Дмитрий Гизлык

НЕЙРОСЕТИ В АЛГОТРЕЙДИНГЕ НА MQL5

MetaQuotes

Содержание

Нейросети в алготрейдинге на MQL5

В эпоху цифровых технологий и искусственного интеллекта алгоритмическая торговля преобразует финансовые рынки, предлагая новые стратегии для достижения прибыли. Книга "Нейросети в алготрейдинге на MQL5" является уникальным руководством, объединяющим глубокие технологические знания с практическими аспектами создания торговых алгоритмов. Эта книга предназначена для трейдеров, разработчиков и финансовых аналитиков, желающих освоить принципы работы нейронных сетей и их применение в алготрейдинге на платформе MetaTrader 5.

Книга разбита 7 глав, которые помогут вам овладеть основами нейронных сетей, и научат вас интегрировать их в ваши собственные торговые роботы на MQL5. Начиная с основных принципов нейронных сетей и заканчивая более сложными архитектурными решениями и механизмами внимания, эта книга предоставляет вам всю необходимую информацию для успешной реализации машинного обучения в вашем алгоритмическом трейдинге.

Вы узнаете, как использовать различные типы нейронных сетей, включая сверточные и рекуррентные модели, и как интегрировать их в среду MQL5. Книга также рассматривает архитектурные решения для повышения сходимости моделей, такие как пакетная нормализация и Dropout.

Кроме того, автор предоставляет практическое руководство по обучению нейронных сетей и внедрение их в ваши торговые стратегии. Вы научитесь создавать торговые советники для проверки эффективности обученных моделей на новых данных, чтобы оценить их потенциал на реальных финансовых рынках.

- Глава 1 вводит вас в мир искусственного интеллекта, обучая основам построения нейронных сетей и их ключевым компонентам, таким как функции активации и методы инициализации весов.
- Глава 2 раскрывает возможности MetaTrader 5, детально описывая, как использовать инструменты платформы для создания мощных алготрейдинговых стратегий.
- Глава 3 посвящена пошаговой разработке вашей первой модели нейронной сети на MQL5, начиная от подготовки данных до реализации и тестирования модели.
- Глава 4 глубоко погружается в изучение базовых типов нейронных слоев, включая сверточные и рекуррентные нейронные сети, их реализацию и тестирование.
- Глава 5 знакомит с механизмами внимания, включая Self-Attention и Multi-Head Self-Attention, представляя передовые методы анализа данных.
- Глава 6 объясняет архитектурные решения для повышения сходимости моделей, такие как пакетная нормализация и Dropout.
- Глава 7 завершает книгу, предлагая методы проверки торговых стратегий с использованием разработанных моделей нейронных сетей в реальных торговых условиях на MetaTrader 5.

Благодаря книге "Нейросети в алготрейдинге на MQL5" вы получите комплексные знания и практические навыки для создания собственных торговых роботов, способных анализировать рынок и принимать решения с помощью передовых технологий машинного обучения.

 Примеры из книги "Нейросети в алготрейдинге на MQL5"

 Примеры из книги также доступны в публичном проекте \MQL5\Shared Projects\NeuroBook

Введение

Вся история человечества — это создание и совершенствование орудий труда. С того момента, как древний человек взял первую палку в руки, орудия физического труда постоянно совершенствуются. Наряду с постоянными улучшениями орудий физического труда, человек также развивает орудия интеллектуального труда.

От первой цифровой механической вычислительной машины, построенной Вильгельмом Шикардо в 1623 году, мир вычислительных машин эволюционировал до современных компьютеров, которые благодаря разработанным алгоритмам позволяют перейти от простых вычислений к решению других более интеллектуальных задач. Появились алгоритмы искусственного интеллекта, который с каждым днем охватывает все большие аспекты нашей жизнедеятельности. Все чаще новостные ленты пестрят сообщениями «Нейросеть научили...».

Первое определение искусственного интеллекта сформулировал Джон Маккарти в 1956 году:

> «Искусственный интеллект — свойство интеллектуальных систем выполнять творческие функции, которые традиционно считаются прерогативой человека; наука и технология создания интеллектуальных машин, особенно интеллектуальных компьютерных программ».

Параллельно развивалось и искусство биржевой торговли. В попытках угадать будущее движение биржевых инструментов трейдеры тщательно изучали графики в поисках закономерностей ценовых движений, вырабатывали торговые правила и создавали целые торговые стратегии.

Появление электронных вычислительных машин не обошло стороной и эту сферу деятельности человека. Использование ЭВМ позволило обрабатывать больше информации за меньшее время. В результате анализ ценовых движений биржевых инструментов стал более детальным и глубоким.

Дальнейшее развитие привело к появлению программ, способных действовать согласно заложенным торговым стратегиям и совершать сделки 24 часа в сутки без участия человека.

На страницах этой книги мы попытаемся совместить две вышеуказанные сферы деятельности. В данном контексте наиболее уместным будет другое определение искусственного интеллекта, которое сформулировали Андреас Каплан и Майкл Хенлайн:

> «Искусственный интеллект — это способность системы правильно интерпретировать внешние данные, извлекать уроки из таких данных и использовать полученные знания для достижения конкретных целей и задач при помощи гибкой адаптации».

Именно это свойство мы будем эксплуатировать. Мы рассмотрим базовые принципы и основы искусственного интеллекта, а затем возьмем широко используемый терминал *MetaTrader 5* и продемонстрируем его возможности в построении различных алгоритмов интеллектуальных программ.

На реальных данных мы проверим способность реализованных алгоритмов к выявлению закономерностей, ведь понимание закономерностей дает возможность определить наиболее вероятный вектор развития предстоящих событий.

Мы не рассматриваем данную книгу как учебное пособие по изучению алгоритмов искусственного интеллекта. В книге даны лишь базовые понятия и принципы без глубокого погружения в математические особенности вычислений и построения алгоритмов.

Данный труд будет более интересен для практиков. В книге будут даны примеры использования различных алгоритмов для решения реального кейса, будут представлены результаты обучения нейронных сетей, построенных на различных архитектурах с использованием разных алгоритмов.

Хочу обратить внимание всех читателей на то, что биржевая торговля сопряжена с высоким риском. Ответственность за совершение любых торговых операций лежит на читателе. В книге рассматриваются инструменты, а не готовые торговые решения. Для использования предложенных инструментов в реальной торговле необходимо проведение дополнительной работы по построению торгового робота и/или индикаторов для принятия решений трейдером, а также их тщательное тестирование.

1. Основные принципы построения искусственного интеллекта

Познание мира и самого себя в нем — неотъемлемая часть существования человека. Размышления о природе сознания издавна поднимались философами. Учеными нейрофизиологами и психологами были созданы теории о принципах и механизмах работы человеческого мозга. Как и в ряде других наук, подсмотренные у природы процессы легли в основу создания интеллектуальных машин.

Основным составным кирпичиком в структуре человеческого мозга является нейрон. Точное количество нейронов в нервной системе человека доподлинно неизвестно, но оценки говорят о приблизительно 100 млрд. Нейроны, состоящие из **тела клетки**, **дендритов** и **аксона**, соединяются между собой, формируя сложную сеть, где места соединений называют синапсами.

Описанные процессы и структуры послужили основой для создания искусственных нейронных сетей. В 1943 году была опубликована работа Уоррена Мак-Каллока и Вальтера Питтса «Логическое исчисление идей, относящихся к нервной активности», в которой были предложены и описаны две теории нейронных сетей: с петлями и без. Предложенные теории были большим шагом в понимании взаимодействия нейронов и в последующем легли в основу принципов построения взаимодействия нейронов в искусственных нейронных сетях. В 1949 году вышла книга Дональда Хебба «Организация поведения: нейропсихологическая теория». Эта работа закладывает основы обучения нейронных связей.

Указанные выше труды рассматривали процессы в мозге человека и получили свое развитие в работах Фрэнка Розенблатта. Разработанная им в 1957 году математическая модель перцептрона легла в основу первого в мире нейрокомпьютера «Марк-1», созданного им же в 1960 году. Следует отметить, что различные варианты перцептрона успешно используются и в наши дни для решения ряда задач.

Но обо всем по порядку. В этой главе мы рассмотрим математические модели нейрона и перцептрона:

- Нейрон и принципы построения нейронных сетей. Этот раздел подробно излагает строение нейрона и основные концепции, лежащие в основе искусственных нейронных сетей, а также их важность для понимания интеллектуальных систем.
- Функции активации являются неотъемлемой частью нейронных сетей, определяя, как нейрон должен реагировать на входящие сигналы. Этот раздел посвящен различным типам функций активации и их роли в процессе обучения нейросетей.

- Методы инициализации весовых коэффициентов нейронной сети являются критическим этапом в подготовке сети к обучению, влияющий на ее способность к обучению и сходимости.
- Обучение нейронной сети разбирается через призму ключевых компонентов: функций потерь, метода обратного распространения градиента ошибки и методов оптимизации, которые вместе формируют основу для эффективного обучения сетей.
- Приемы повышения сходимости нейронных сетей, включая регуляризацию, Dropout и нормализацию, детализируют стратегии улучшения производительности и стабильности нейросетей в процессе их обучения.
- Искусственный интеллект в трейдинге знаменует собой практическое применение обсуждаемых технологий, исследуя, как искусственный интеллект и машинное обучение могут быть использованы для анализа финансовых рынков и принятия торговых решений.

Таким образом, глава представляет собой комплексное изложение основ искусственного интеллекта и нейронных сетей, их структуры, механизмов работы и применения в современном мире, в частности, в алготрейдинге.

1.1 Нейрон и принципы построения нейронных сетей

В работе «Логическое исчисление идей, относящихся к нервной активности» Уоррен Мак-Каллок и Вальтер Питтс предложили математическую модель нейрона и описали базовые принципы организации нейронных сетей. Математическая модель искусственного нейрона включает два этапа вычислений. По аналогии с нейроном человека в математической модели искусственного нейрона дендриты представлены вектором числовых значений X, который подается на вход искусственного нейрона. Зависимость значения нейрона от каждого конкретного входа определяется вектором весовых коэффициентов W. Первый этап вычислений модели искусственного нейрона реализован в виде произведения вектора исходных сигналов на вектор весовых коэффициентов, что с математической точки зрения дает взвешенную сумму исходных данных.

$$S = \sum_{i=1}^{n} w_i x_i,$$

где:

- n — количество элементов во входной последовательности;
- w_i — весовой коэффициент i-го элемента последовательности;
- x_i — i-ый элемент входной последовательности.

Весовые коэффициенты определяют чувствительность нейрона к изменениям конкретного входного значения и могут быть как положительными, так и отрицательными. Тем самым имитируется работа возбуждающих и тормозящих сигналов. Значения весовых коэффициентов, удовлетворяющие решению конкретной задачи, подбираются в процессе обучения нейронной сети.

Как уже говорилось ранее, на аксоне нейрона появляется сигнал только после накопления критического значения в теле клетки. В математической модели искусственного нейрона этот этап реализован путем введения некой функции активации.

$$OUT = f(W, X) = f\left(\sum_{i=1}^{n} w_i x_i\right)$$

И здесь возможны вариации. В первых моделях использовалась простая функция сравнения взвешенной суммы входящих значений с неким пороговым значением. Такой подход имитировал природу биологического нейрона, который может быть возбужден или находиться в покое. График такой функции активации нейрона будет с резким перепадом значения в точке порогового значения.

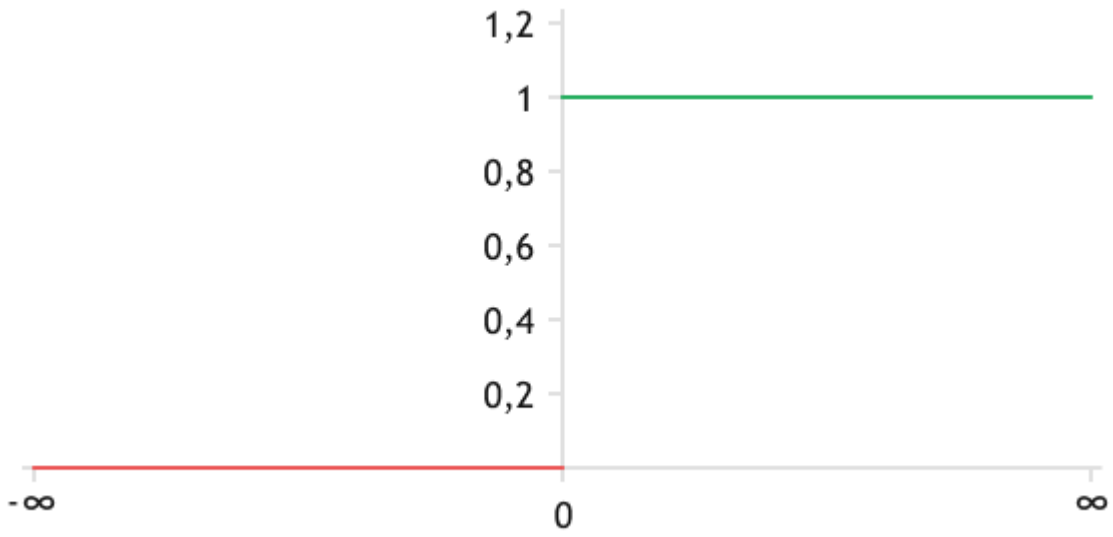


График пороговой функции активации нейрона.

В 1960 году в свет вышла работа Бернарда Видроу (Bernard Widrow) и Маршиан Хофф (Marcian Hoff) «Adaptive switching circuits», в которой была представлена машина адаптивной линейной классификации Adaline. Данный труд показал, что использование непрерывных функций активации нейрона позволит решать больший круг задач с меньшей погрешностью. С тех пор и до нашего времени в качестве функций активации нейрона широко используются различные сигмовидные функции. В таком варианте получается более сглаженный график функции математической модели нейрона.

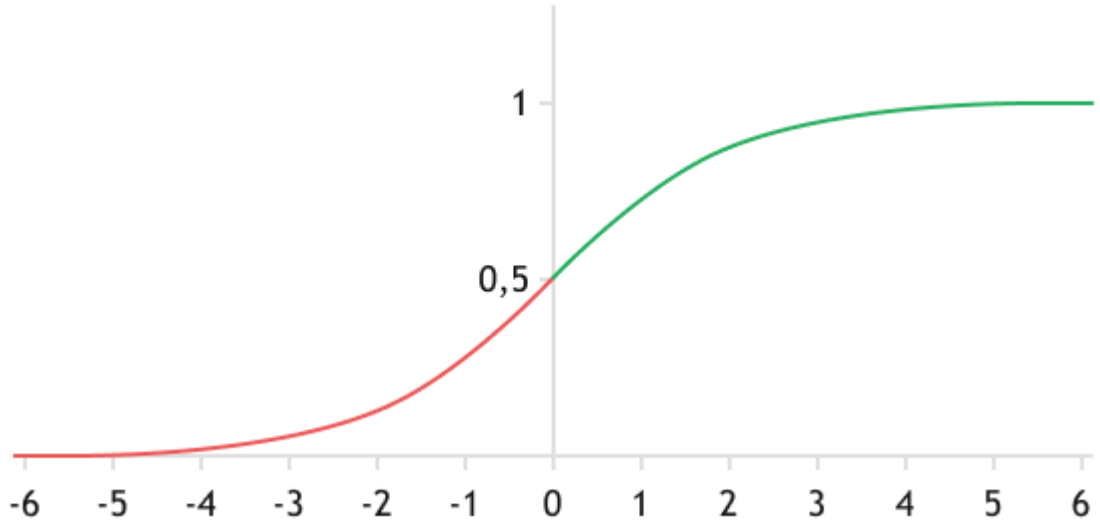


График логистической функции (Сигмоида)

Различные варианты функций активации, их преимущества и недостатки обсудим в следующей главе книги. В общем виде математическую модель искусственного нейрона схематично можно изобразить в нижеследующем виде.

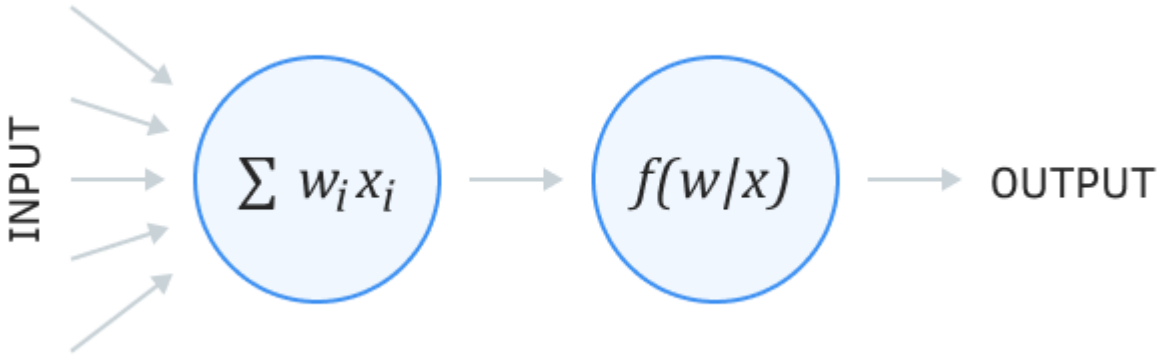


Схема математической модели нейрона

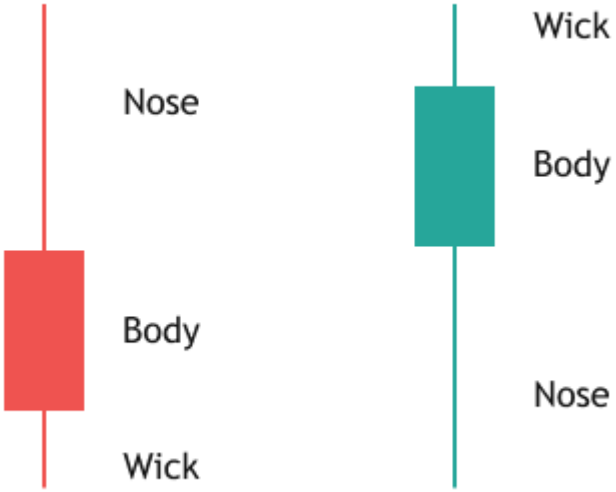
Рассмотренная математическая модель нейрона позволяет дать логический ответ *Истина* или *Ложь* на основании анализа исходных данных. Рассмотрим действие модели на примере поиска свечной модели «Пин-бар».

Согласно классической модели Пин-бар, размер «Носа» свечи должен, как минимум, в 2,5 раза превышать размер тела и второй тени. Математически это можно представить так:

$$Nose > 2.5(Body + Wick)$$

или

$$1 * Nose - 2.5 * Body - 2.5 * Wick > 0$$



Пин-бар

В соответствии с математической моделью нейрона, на вход нейрона мы будет подавать три значения: размеры носа, тела и тени свечи. Весовыми коэффициентами будут 1, −2,5 и −2,5, соответственно. Сразу надо сказать, что веса мы не будем считать при конструировании нейронной сети. Они будут подбираться в процессе ее обучения.

Функцией активации будет логическое сравнение взвешенной суммы с нулем. Если взвешенная сумма входных значений больше нуля, то свечной паттерн найден и нейрон будет активирован. На выходе нейрона будет 1. Если взвешенная сумма меньше нуля, то паттерн не найден. Нейрон остается деактивирован и на выходе нейрона будет 0.

Теперь у нас есть нейрон, который будет реагировать на свечной паттерн Пин-бар. Но обратите внимание, что в бычьем паттерне носом будет нижняя тень, а в медвежьем — верхняя. То есть если мы подаем на вход нейрона вектор значений, содержащий последовательно верхнюю тень, тело и нижнюю тень свечи, то для определения паттерна нам требуется два нейрона: один будет определять бычью модель, а второй медвежью.

Значит ли это, что нам потребуется создавать программу для каждого паттерна отдельно? Нет. Мы объедим их в единую модель нейронной сети.

В нейронной сети все нейроны группируются в последовательные слои. По расположению и назначению нейронные слои подразделяются на входной, скрытый и выходной. Входной и выходной слои всегда по одному, а вот количество скрытых слоев может быть различно, в зависимости от сложности поставленной задачи.

Количество нейронов во входном слое соответствует количеству исходных данных. В нашем примере их три: верхняя тень, тело, нижняя тень.

Скрытый слой в нашем случае состоит из двух нейронов, определяющих бычий и медвежий паттерн. Количество скрытых слоев и нейронов в них задается при проектировании нейронной сети и определяется ее архитектором в зависимости от сложности решаемой задачи.

Количество скрытых слоев определяет, каким образом пространство входных данных делится на подклассы. Нейронная сеть с одним скрытым слоем делит пространство входных данных гиперплоскостью. Наличие двух скрытых слоев позволяет сформировать в пространстве входных

данных выпуклую область. Третий скрытый слой дает возможность формировать практически любые области в пространстве.

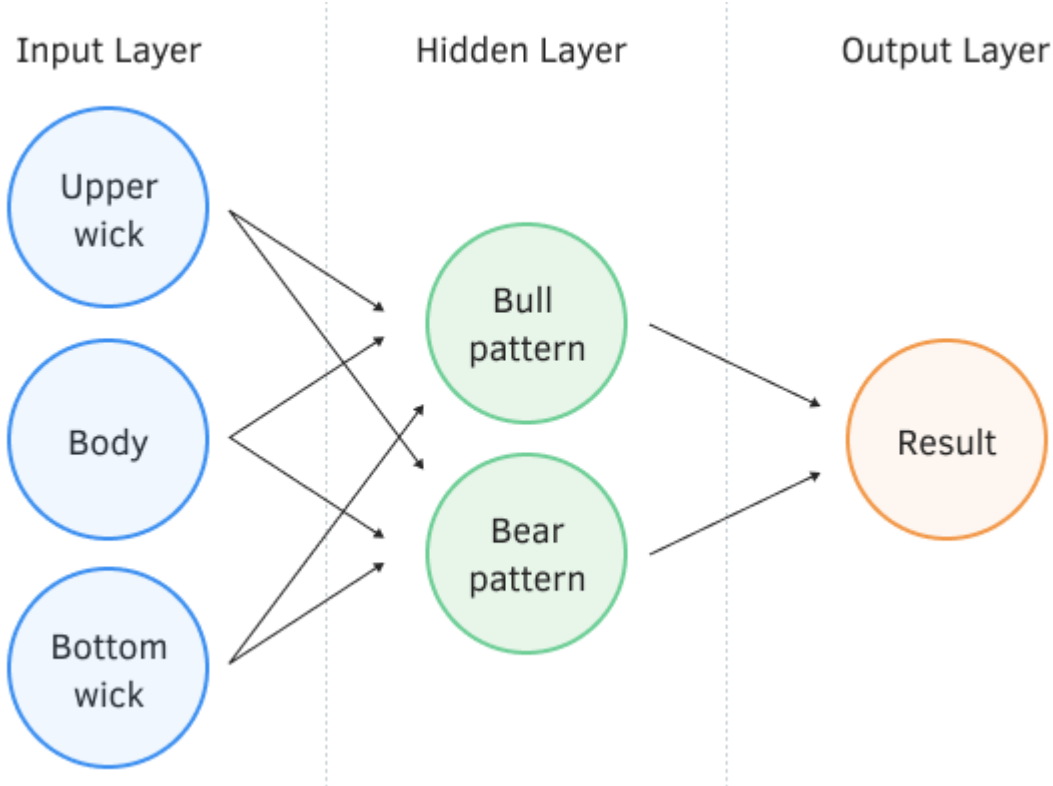
Количество нейронов в срытом слое определяется количеством искомых признаков на каждом уровне.

Количество нейронов в выходном слое определяется архитектором нейронной сети в зависимости от возможных вариантов решений поставленной задачи. Для решения бинарных задач и задач регрессии бывает достаточно одного нейрона. Для задач классификации количество нейронов будет соответствовать конечному количеству классов.

Исключением является бинарная классификация, когда все объекты делятся на два класса. В таком случае достаточно одного нейрона, так как вероятность отнесения объекта ко второму классу P_2 равна разнице между единицей и вероятностью отнесения объекта к первому классу P_1.

$$P_2 = 1 - P_1$$

В нашем примере выходной слой будет содержать только один нейрон, который выдаст результат, открывать сделку или нет и в каком направлении. Для этого бычьему паттерну установим вес 1, а медвежьему — вес −1. В результате сигналом на покупку будет 1, на продажу — значение −1. Ноль будет означать отсутствие торгового сигнала.



Модель перцептрона

Такая модель нейронной сети была предложена Фрэнком Розенблаттом в 1957 году и получила название *Перцептрон*. Данная модель является одной из первых моделей искусственных нейронных сетей. Она способна выстраивать ассоциативные связи между входными данными и результирующим действием. В реальной жизни это можно сопоставить с реакцией человека на сигнал светофора.

Конечно, перцептрон не лишен недостатков, есть ряд ограничений при его использовании. Но за годы исследований были достигнуты хорошие результаты применения перцептрона в задачах классификации и аппроксимации. К тому же были разработаны механизмы обучения перцептрона, с которыми мы познакомимся чуть позже.

1.2 Функции активации

Наверное, одной из самых сложных задач, которая становится перед архитектором нейронной сети, является выбор функции активации нейронов. Ведь именно функция активации создает нелинейность в нейронной сети. Во многом от выбора функции активации зависит процесс обучения нейронной сети и финальный результат в целом.

На сегодняшний день разработан и применяется целый ряд функций активации. Каждая из них обладает своими достоинствами. Но, к сожалению, все они имеют свои недостатки. Чтобы мы могли правильно применять их достоинства и бороться или смириться с их недостатками, я предлагаю рассмотреть и обсудить некоторые из них.

Пороговая (ступенчатая) функция активации

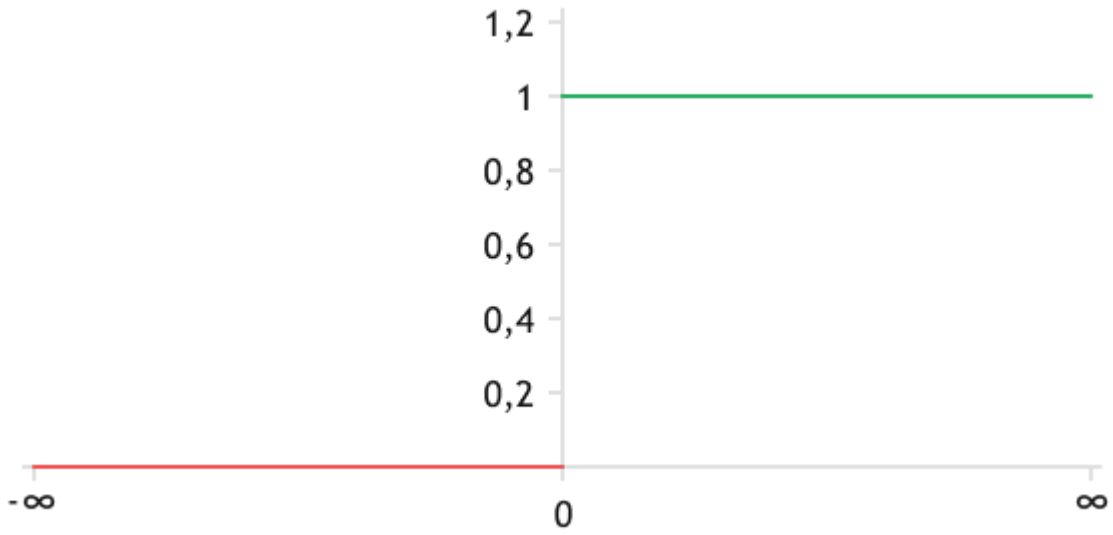
Пороговая функция активации, наверное, была применена одной из первых. И это не удивительно, ведь она повторяет действие биологического нейрона:

- возможно только два состояния (активирован или нет);
- нейрон активируется при достижении порогового значения θ.

Математически данную функцию активации можно выразить формулой:

$$f(x) = \begin{cases} 1, & x >= \theta \\ 0, & x < \theta \end{cases}$$

При θ=0 функция имеет нижеследующий график.



Пороговая (ступенчатая) функция активации

Данная функция активации легка в понимании, но основным ее недостатком является сложность или даже невозможность обучения нейронной сети. Дело в том, что в алгоритмах обучения

нейронных сетей используется производная первого порядка. А производная рассматриваемой функции равна нулю на всем протяжении, за исключением $x=\theta$ (в данной точке она не определена).

Реализовать данную функцию в виде программного кода *MQL5* довольно легко. Константа ***theta*** определяет уровень, при достижении которого нейрон будет активирован. При вызове функции активации в параметрах передадим предварительно посчитанную взвешенную сумму исходных данных. Внутри функции сравним полученное в параметрах значение с уровнем активации *theta* и вернем значение активации нейрона.

```
const double theta = 0;
//———
double ActStep(double x)
  {
   return (x >= theta ? 1 : 0);
  }
```

На *Python* реализация так же легка.

```
theta = 0
def ActStep (x):
  return 1 if x >= theta else 0
```

Линейная функции активации

Линейная функция активации задается линейной функцией:

$$f(x) = ax + b,$$

где:

- ***a*** — определяет угол наклона линии;
- ***b*** — смещение линии по вертикале.

Как частный случай линейной функции активации, при a=1 и b=0 функция имеет вид $f(x) = x$.

Функция может генерировать значения в диапазоне от $-\infty$ до $+\infty$ и дифференцируема на всем протяжении. Производная функции постоянна и равна a, что облегчает процесс обучения нейронной сети. Указанные свойства позволяют широко использовать данную функцию активации при решении задач регрессии.

Следует отметить, что вычисление взвешенной суммы входов нейрона является линейной функцией. Применение линейной функции активации дает линейную функцию всего нейрона и нейронной сети. Данное свойство не позволяет использовать линейную функцию активации для решения нелинейных задач.

В то же время, создавая нелинейность на скрытых слоях нейронной сети с помощью других функций активации, мы можем использовать линейную функцию активации в нейронах выходного слоя своей модели. Такой прием позволяет решать нелинейные задачи регрессии.

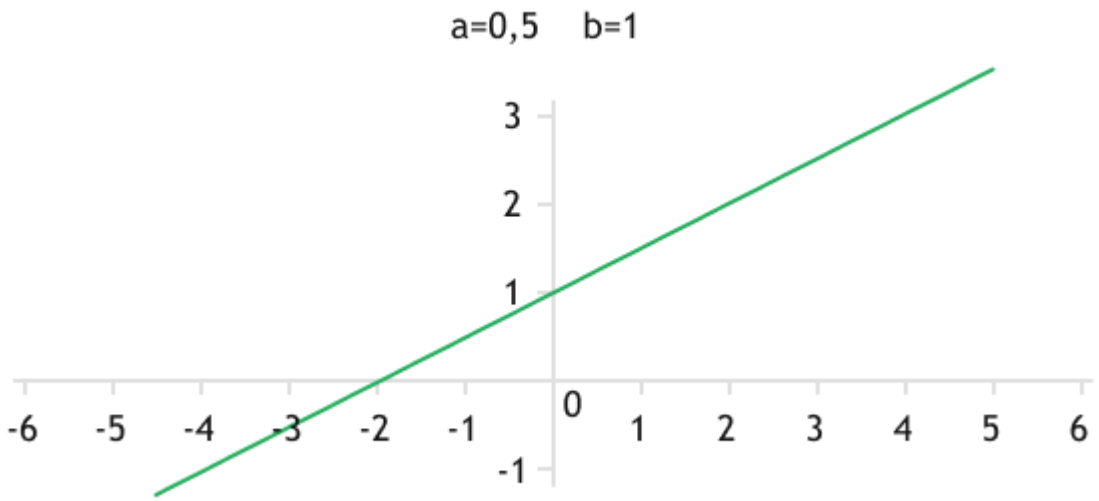


График линейной функции

Реализация линейной функции активации в виде программного кода *MQL5* требует создания двух констант: ***a*** и ***b***. По аналогии с реализацией предыдущей функции активации, при вызове функции в параметрах передадим предварительно посчитанную взвешенную сумму исходных данных. Внутри функции реализация расчетной части умещается в одну строчку.

```
const double a = 1.0;
const double b = 0.0;
//———
double ActLinear(double x)
  {
   return (a * x + b);
  }
```

На *Python* реализация аналогична.

```
a = 1.0
b = 0.0
def ActLinear (x):
  return a * x + b
```

Логистическая функция активации (Сигмоида)

Логистическая функция активации, наверное, самая распространенная *S*-образная функция. Значения функции находятся в диапазоне от 0 до 1, они асимметричны относительно точки [0, 0.5]. График функции напоминает пороговую функцию, но с плавным переходом между состояниями.

Математическая формула функции имеет вид:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Данная функция позволяет нормализовать выходные значений функции в диапазоне [0, 1]. Благодаря этому свойству использование логистической функции вводит понятие вероятности в практику нейронных сетей. Это свойство широко эксплуатируется в нейронах выходного слоя

при решении задач классификации, когда количество нейронов выходного слоя равно количеству классов, а отнесение объекта к тому или иному классу определяется по максимальной вероятности (максимальному значению выходного нейрона).

Функция дифференцируема на всем промежутке допустимых значений. Значение производной легко рассчитывается через значение функции по формуле:

$$\frac{df(x)}{dx} = f(x) * (1 - f(x))$$

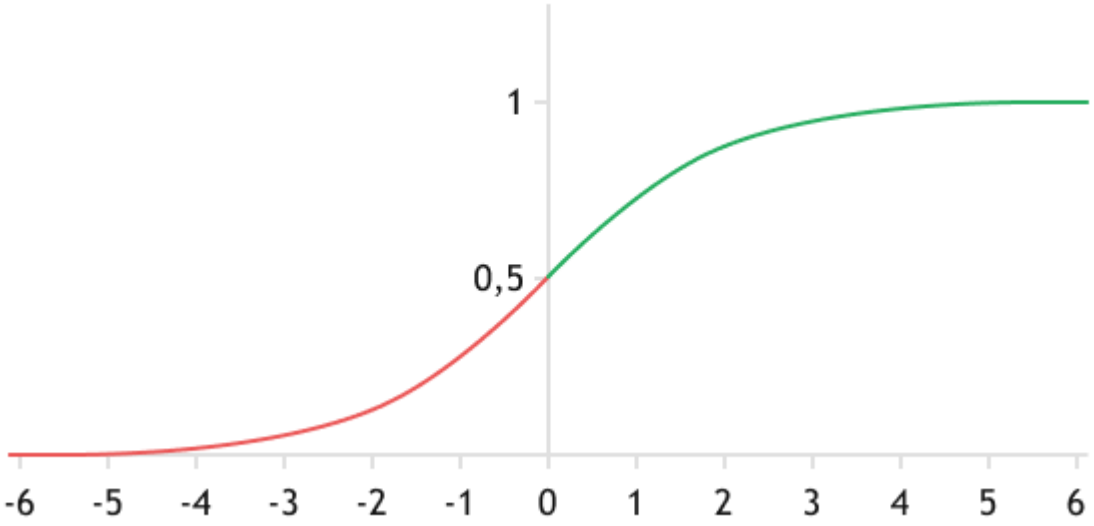


График логистической функции (Сигмоида)

Иногда в практике применения нейронных сетей можно встретить немного измененную логистическую функцию:

$$f(x) = \frac{a}{1 + e^{-x}} - b,$$

где:

- ***a*** — растягивает диапазон значений функции от 0 до a;
- ***b*** — по аналогии с линейной функцией смещает результирующее значения.

Производная такой функции также вычисляется через значение функции по формуле:

$$\frac{df(x)}{dx} = f(x) * \left(1 - \frac{f(x)}{a}\right)$$

В практике наиболее часто применяют $b = \frac{1}{2}a$, чтобы график функции был асимметричен относительно начала координат.

Все вышеизложенные свойства добавляют популярности использованию логистической функции в качестве функции активации нейрона.

Но и она не лишена недостатков. При входных значения меньше −6 и больше 6 значение функции прижимается к границам диапазона значений функции, а производная стремится к

нулю. Как следствие, градиент ошибки также стремится к нулю. Это приводит к снижению скорости обучения нейронной сети, а порой и вовсе делает сеть практически необучаемой.

Ниже предлагаю рассмотреть реализацию наиболее общего варианта логистической функции с двумя константами ***a*** и ***b***. Экспоненту вычислим с помощью функции `exp()`.

```
const double a = 1.0;
const double b = 0.0;
//———
double ActSigmoid(double x)
  {
   return (a / (1 + exp(-x)) - b);
  }
```

При реализации на Python перед использованием функции экспоненты необходимо импортировать библиотеку *math*, в которой собраны основные математические функции. В остальном алгоритм и реализация функции аналогичны реализации на *MQL5*.

```
import math
a = 1.0
b = 0.0
def ActSigmoid (x):
  return a / (1 + math.exp(-x)) - b
```

Гиперболический тангенс (*tanh*)

Альтернативой логистической функции активации является гиперболический тангенс (*Tanh*). Он так же, как и логистическая функция, имеет *S*-образный график, а значения функции нормализованы. Но они принадлежат диапазону от −1 до 1, и изменение состояния нейрона осуществляется в 2 раза быстрее. График функции также асимметричен, но в отличие от логистической функции центр асимметрии находится в центре координат.

$$f(x) = tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

Функция дифференцируема на всем промежутке допустимых значений. Значение производной легко считается через значение функции по формуле:

$$\frac{df(x)}{dx} = (1 + f(x))(1 - f(x)) = 1 - (f(x))^2$$

Функция гиперболического тангенса является альтернативой логистической функции, которая довольно часто сходится быстрее.

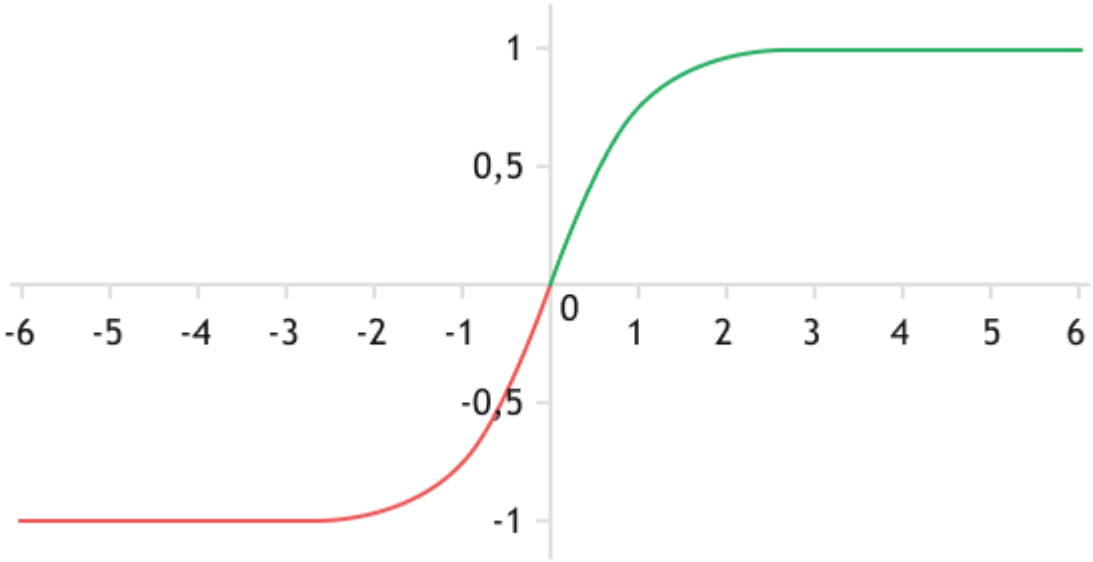


График функции гиперболического тангенса (TANH)

Но и она не лишена главного недостатка логистической функции: при насыщении функции, когда значения функции приближаются к границам диапазона значений, производная функции стремится к нулю. Как следствие, градиент ошибки стремится к нулю.

Функция гиперболического тангенса уже реализована в используемых нами языках программирования, и для ее вызова достаточно вызвать функцию `tanh()`.

```
double ActTanh(double x)
  {
   return tanh(x);
  }
```

И аналогичная реализации на Python.

```
import math
def ActTanh (x):
  return math.tanh(x)
```

Выпрямленный линейный блок (ReLU)

Еще одна широко используемая функция активации нейронов — *ReLU* (выпрямленный линейный блок). При входящих значениях больше нуля функция возвращает само значение, подобно линейной функции активации. При значениях меньше или равном нулю функция всегда возвращает 0. Математически данная функция выражается формулой:

$$f(x) = \begin{cases} x, & x > 0 \\ 0, & x <= 0 \end{cases} = max(0,x)$$

График функции представляет собой нечто среднее между пороговой и линейной функцией.

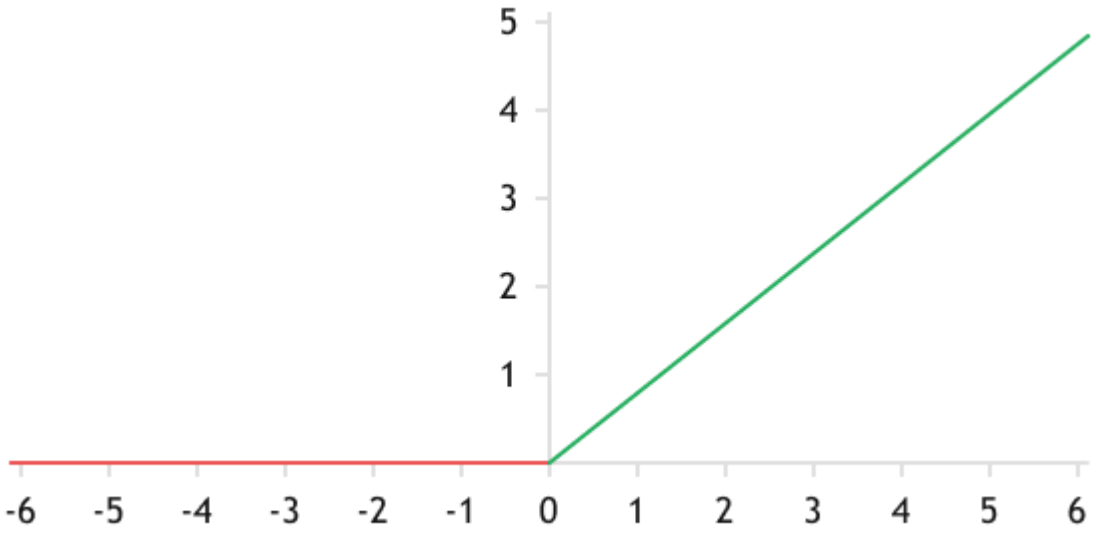


График функции ReLU

ReLU, наверное, одна из наиболее распространенных функций активации на данный момент. Такое распространение она получила благодаря своим свойствам:

- Как и пороговая функция, работает по принципу биологического нейрона, активируется только после достижения порогового значения (0). В отличие от пороговой функции, при активации нейрон возвращает не константу, а переменное значение.
- Область значений лежит в диапазоне от 0 до $+\infty$, что позволяет использовать функцию при решении задач регрессии.
- При значении функции больше нуля ее производная равна единице.
- При расчете функции не требуются сложные вычисления, что ускоряет процесс обучения.

В литературе приводятся примеры, когда нейронные сети с *ReLU* обучаются до 6 раз быстрее сетей с использованием *TANH*.

Тем не менее, использование *ReLU* также не лишено недостатков. Когда взвешенная сумма входов меньше нуля, производная функции равна нулю. В таком случае нейрон не обучается и не передает градиент ошибки на предшествующие слои нейронной сети. В процессе обучения существует вероятность получить такой набор весовых коэффициентов, при котором нейрон будет деактивирован на протяжение всего цикла обучения. Такой эффект получил название «мертвых нейронов».

Субъективно наличие мертвых нейронов можно отследить по росту скорости обучения: чем сильнее возрастает скорость обучения с каждым проходом, тем больше сеть содержит мертвых нейронов.

Для минимизации эффекта мертвых нейронов при использовании *ReLU* было предложено несколько вариаций данной функции, но все они сводились к одному — применению некоего коэффициента *a* для взвешенной суммы меньше нуля.

$$f(x) = \begin{cases} x, & x > 0 \\ ax, & x <= 0 \end{cases}$$

LReLU	Leaky ReLU — ReLU с утечкой	a = 0,01
PReLU	Parametric ReLU — параметрическая ReLU	Параметр a подбирается в процессе обучения нейронной сети
RReLU	Randomized ReLU — рандомизированный ReLU	Параметр a задается случайным образом при создании нейронной сети

Классический вариант *ReLU* удобно реализовать с помощью функции `max()`. Реализация ее вариантов потребует создания константы или переменной ***a***. Вариант инициализации будет зависеть от выбранной функции (*LReLU* / *PReLU* / *RReLU*). А внутри нашей функции активации создадим логическое разветвление, в зависимости от значение получаемого параметра.

```
const double a = 0.01;
//——
double ActPReLU(double x)
  {
   return (x >= 0 ? x : a * x);
  }
```

На Python реализация аналогична.

```
a = 0.01
def ActPReLU (x):
  return x if x >= 0 else a * x
```

Softmax

Если выше рассмотренные функции рассчитывались на данных исключительно отдельно взятого нейрона, то функция *Softmax* применяется ко всем нейронам отдельного слоя сети (как правило, последнего). Наряду с сигмоидой, вводится понятие вероятности в нейронные сети. Диапазон значений функции лежит между 0 и 1, а сумма всех выходных значений нейронов взятого слоя равна 1.

Математическая формула функции имеет вид:

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Функция дифференцируема на всем промежутке значений, и ее производная легко вычисляется через значение функции:

$$\frac{df(x_i)}{dx_i} = \begin{cases} i = j & f(x_i) * (1 - f(x_i)) \\ i \neq j & -f(x_j) f(x_i) \end{cases}$$

Функция широко применяется в последнем слое нейронной сети при решении задач классификации. Считается, что выходное значение нейрона, нормализованное функцией

Softmax, показывает вероятность отнесения объекта к соответствующему классу классификатора.

Стоит отметить, что вычисление *Softmax* трудозатратно, поэтому его применение оправдано в последнем слое нейронных сетей многоклассовой классификации.

Реализация функции *Softmax* на *MQL5* будет немного сложнее рассмотренных выше примеров. Это связано с обработкой нейронов всего слоя. Следовательно, в параметрах функция будет получать не отдельное значение, а целый массив данных.

Надо обратить внимание, что в *MQL5* массивы, в отличие от переменных, передаются в параметры функций указателями на элементы памяти, а не значениями.

Наша функция в параметрах примет указатели на два массива данных *X* и *Y* и в завершение операций вернет логический результат. Непосредственно результаты операций будут в массиве *Y*.

В теле функции сначала проверяем размер массива исходных данных *X*. Полученный массив должен быть не нулевой длины. Затем изменяем размер массива для записи результатов *Y*. При неудачном выполнении какой-либо из операций выходим из функции с результатом *false*.

```
bool SoftMax(double& X[], double& Y[])
  {
   uint total = X.Size();
   if(total == 0)
      return false;
   if(ArrayResize(Y, total) <= 0)
      return false;
```

Далее организуем два цикла. В первом посчитаем экспоненты для каждого элемента полученного массива данных и суммируем полученные значения.

```
//--- Расчет экспоненты для каждого элемента массива
   double sum = 0;
   for(uint i = 0; i < total; i++)
      sum += Y[i] = exp(X[i]);
```

Во втором цикле нормализуем значения массива, созданного в первом цикле. Перед выходом из функции вернем полученные значения.

```
//--- Нормализация данных в массиве
   for(uint i = 0; i < total; i++)
      Y[i] /= sum;
//---
   return true;
  }
```

На *Python* реализация выглядит намного проще, т.к. функция *Softmax* уже реализована в библиотеке *Scipy*.

```
from scipy.special import softmax
def ActSoftMax (X):
  return softmax(X)
```

Swish

В октябре 2017 года команда исследователей из Google Brain провела работу по автоматическому поиску функций активации. Результаты этой работы были представлены в статье Searching for Activation Functions. В статье приведены результаты тестирования целого ряда функций в сравнении с ReLU. Наилучшие показатели были достигнуты в нейросетях с функцией активации Swish. Только замена ReLU на Swish (без переобучения) позволила улучшить показатели нейронных сетей.

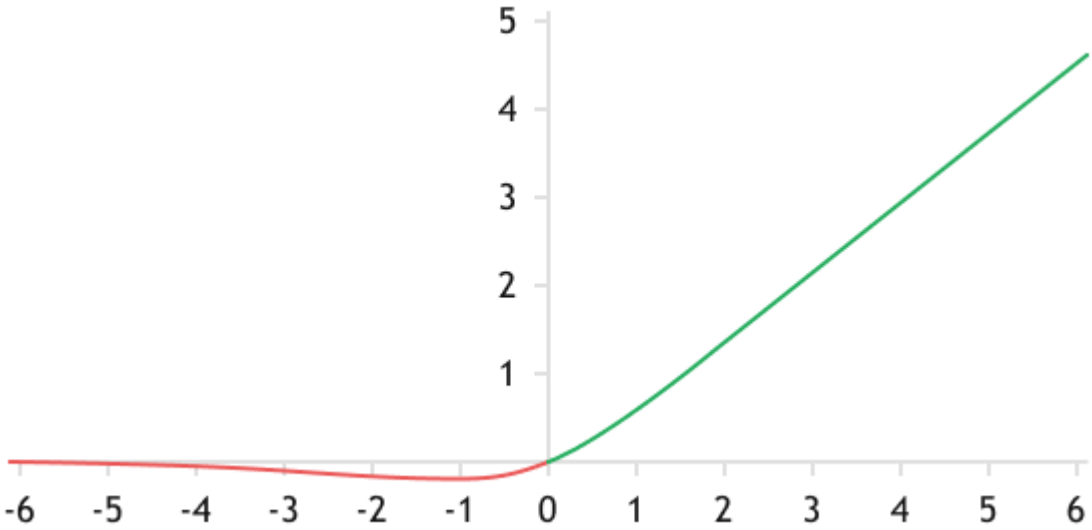


График функции Swish

Математическая формула функции имеет вид:

$$f(x) = x * sigmoid(\beta x) = \frac{x}{1 + e^{-\beta x}}$$

Параметр β влияет на нелинейность функции и может быть принят константой при проектировании сети или подбираться в процессе обучения. При β=0 функция сводится к масштабируемой линейно функции.

$$f(x) = \frac{x}{1 + e^{-x*0}} = \frac{x}{1 + e^{0}} = \frac{x}{2}$$

При β=1 график функции Swish приближается к ReLU. Но в отличие от последней, функция дифференцируема на всем промежутке значений.

Функция дифференцируема на всем протяжении, и ее производная вычисляется через значение функции. Но в отличие от сигмоиды, для вычисления производной требуется еще и входящее значение. Математическая формула производной имеет вид:

$$\frac{df(x)}{dx} = \beta f(x) + \frac{f(x)(1 - \beta f(x))}{x}$$

Реализация функции в программном коде *MQL5* аналогична представленной выше Сигмоиде. Параметр a заменяется полученным значением взвешенной суммы и добавляется параметр нелинейности β.

```
const double b=1.0;
//———
double ActSwish(double x)
  {
   return (x / (1 + exp(-b * x)));
  }
```

Аналогична реализация на *Python*.

```
import math
b=1.0
def ActSwish (x):
  return x / (1 + math.exp(-b * x))
```

Стоит отметить, что это далеко не полный список возможных функций активации. Возможны как вариации к приведенным выше функциям, так и использование иных функций. Выбор функции активации и пороговых значений лежит на архитекторе нейронной сети. Далеко не всегда все нейроны в сети имеют одинаковую функцию активации. В практике широко используются нейронные сети, в которых функция активации меняется от слоя к слою. Такие сети называют ***гетерогенными***.

Позже будет показано, что для реализации нейронных сетей гораздо удобнее использовать векторные и матричные операции, которые реализованы в *MQL5*. В частности это касается и функций активации, ведь в функционале матричных и векторных операций создана функция ***Activation***, которая позволяет одной строкой кода вычислять функции активации для целого массива данных. В нашем случае — для всего нейронного слоя. Реализация функции имеет нижеследующий вид.

```
bool vector::Activation(
  vector&                   vect_out,      // вектор для получения значений
  ENUM_ACTIVATION_FUNCTION  activation     // тип функции
   );


bool matrix::Activation(
  matrix&                   matrix_out,    // матрица для получения значений
  ENUM_ACTIVATION_FUNCTION  activation     // тип функции
   );
```

В параметрах функция принимает указатель на вектор или матрицу (в зависимости от источника данных) для записи результатов и тип используемой функции активации. Стоит отметить, что спектр функций активации в векторных/матричных операциях *MQL5* гораздо шире описанных выше. Их полный список приводится в таблице.

Идентификатор	Описание
AF_ELU	Экспоненциальная линейная единица
AF_EXP	Экспоненциальная

Идентификатор	Описание
AF_GELU	Линейная единица ошибки Гаусса
AF_HARD_SIGMOID	Жесткий сигмоид
AF_LINEAR	Линейная
AF_LRELU	Линейный выпрямитель с «утечкой» (Leaky ReLU)
AF_RELU	Усеченное линейное преобразование ReLU
AF_SELU	Масштабированная экспоненциальная линейная функция (Scaled ELU)
AF_SIGMOID	Сигмоид
AF_SOFTMAX	Softmax
AF_SOFTPLUS	Softplus
AF_SOFTSIGN	Softsign
AF_SWISH	Swish-функция
AF_TANH	Гиперболический тангенс
AF_TRELU	Линейный выпрямитель с порогом

1.3 Методы инициализации весовых коэффициентов нейронной сети

При создании нейронной сети перед первым запуском ее обучения нам предстоит неким образом задать начальные весовые коэффициенты. Эта на первый взгляд простая задача имеет большое значение для последующего обучения нейронной сети и в целом имеет большое влияние на результат всей работы.

Дело в том, что наиболее часто используемый для обучения нейронных сетей метод градиентного спуска не способен отличить локальные минимумы функции от ее глобального минимума. На практике применяются различные варианты решений для минимизации этой проблемы, и о них мы поговорим чуть позже. Однако вопрос остается открытым.

Второй момент заключается в том, что метод градиентного спуска — это итерационный процесс. Следовательно, общее время обучения нейронной сети напрямую зависит от того, насколько далеко от конечной точки мы будем в начале.

Кроме того, не будем забывать о законах математики и особенностях функций активации, которые мы обсуждали в предыдущем разделе этой книги.

Инициализация весов единым значением

Наверное, первое, что приходит в голову, это взять какую-нибудь константу (0 или 1) и инициализировать все веса одним значением. К сожалению, это далеко не лучший вариант. И это связано с законами математики.

Использование нуля в качестве синаптического коэффициента во многих случаях губительно для нейронных сетей. В таком случае взвешенная сумма исходных данных будет равна нулю. Как мы знаем из предыдущего раздела, многие варианты функции активации в подобном случае возвращают 0, а нейрон остается деактивированным. Следовательно, никакой сигнал дальше по нейронной сети не проходит.

$$S_0 = \sum_{i=1}^{n} 0 * x_i = 0$$

Производная такой функции по x_i будет равна нулю. Следовательно, в процессе обучения нейронной сети градиент ошибки через такой нейрон тоже не будет передан на предшествующие слои, что парализует процесс обучения.

$$\frac{dS_0}{dx_i} = 0$$

> Использование 0 для инициализации синаптических (весовых) коэффициентов приводит к получению необучаемой нейронной сети, которая в большинстве случаев будет генерировать 0 (зависит от функции активации) вне зависимости от получаемых исходных данных.

Использование константы отличной от нуля в качестве весового коэффициента также имеет недостатки. На входной слой нейронной сети подается набор исходных данных. Все нейроны последующего слоя одинаково работают с этим набором данных. В рамках одного нейрона, по законам математики, в формуле вычисления взвешенной суммы константу можно вынести за скобки. В результате на первом этапе мы получаем масштабирование суммы исходных значений. В процессе обучения возможно изменение значений весовых коэффициентов. Но это касается только первого слоя нейронов, получающего на вход исходные данные.

$$S_{const} = \sum_{i=1}^{n} const * x_i = const \sum_{i=1}^{n} x_i$$

Если посмотреть на нейронный слой в целом. То все нейроны одного слоя получают на вход один набор данных. В результате использования одного коэффициента все нейроны генерируют один и тот же сигнал. Как следствие, все нейроны одного слоя работают синхронно как один нейрон. И это приводит к тому, что на всех входах всех нейронов последующего слоя одно и тоже значение. Так происходит от слоя к слою по всей нейронной сети.

Применяемые алгоритмы обучения, не позволяют выделить отдельный нейрон среди большого количества одинаковых значений. Поэтому в процессе обучения все весовые коэффициенты будут изменяться синхронно. И каждый слой, кроме первого после входного, получит свой коэффициент, единый для всего слоя, что приводит к линейному масштабированию результатов, полученных на одном нейроне.

> Инициализация синаптических коэффициентов единым числом, не равным нулю, ведет к вырождению нейронной сети до одного нейрона.

Инициализация весовых коэффициентов случайными значениями

Если мы не можем инициализировать нейронную сеть одним числом, попробуем провести инициализацию случайными значениями. Для максимальной эффективности не будем забывать

о рассказанном выше. Нам нужно сделать так, чтобы среди синаптических коэффициентов не было двух одинаковых. В этом нам поможет непрерывное равномерное распределение.

Как показала практика, такой подход дает результаты. Но, к сожалению, не всегда. Из-за случайности выбора весовых коэффициентов порой приходится несколько раз инициализировать нейронную сеть, прежде чем будет достигнут желаемый результат. Большое влияние оказывает диапазон разброса значений весовых коэффициентов. Если разрыв между минимальным и максимальным значением окажется достаточно большой, то это приведет к выделению одних нейронов и полному игнорированию других.

Кроме того, в глубоких нейронных сетях существует риск так называемых «взрыва градиента» и «затухания градиента».

Взрыв градиента проявляется при использовании весовых коэффициентов больше единицы. В этом случае при умножении исходных данных на коэффициенты больше единицы взвешенная сумма постоянно растет и с каждым слоем возрастает в экспоненциальной прогрессии. При этом генерация большого числа на выходе часто приводит к большой ошибке.

В процессе обучения мы будет использовать градиент ошибки для корректировки весовых коэффициентов. Для того чтобы провести градиент ошибки от выходного слоя до каждого нейрона нашей сети, нам потребуется умножить полученную ошибку на весовые коэффициенты. В результате градиент ошибки так же, как и взвешенная сумма, будет расти экспоненциально при продвижении по слоям нейронной сети.

Как следствие, в какой-то момент мы получим число, превышающее наши технические возможности по записи значений, и не сможем далее обучать и использовать сеть.

При выборе значений весовых коэффициентов близких к нулю происходит обратная ситуация. Постоянное умножение исходных данных на весовые коэффициенты меньше единицы ведет к уменьшению взвешенной суммы весовых коэффициентов. Этот процесс идет в экспоненциальной прогрессии с ростом количества слоев нейронной сети.

Как следствие, в процессе обучения можем столкнуться с ситуацией, когда градиент малой ошибки при прохождении по слоям будет меньше технически возможной точности. Для наших нейронов градиент ошибки превратится в ноль, и они не будут обучаться.

На момент написания книги общей практикой считается инициализация нейронов по методу Xavier, предложенному в 2010 году. Ксавье Глорот (Xavier Glorot) и Йошуа Бенжио (Yoshua Bengio) предложили проводить инициализацию нейронной сети случайными числами из непрерывного нормального распределения с центром в точке 0 и дисперсией (δ^2), равной 1/n.

Такой подход позволяет генерировать такие синаптические коэффициенты, при которых средняя из активаций нейронов будет равна нулю, а их дисперсия будет одинаковой для всех слоев нейронной сети. Наиболее актуальна инициализация Xavier при использовании гиперболического тангенса (*tanh*) в качестве функции активации.

Теоретическое обоснование такого похода было дано в статье "Understanding the difficulty of training deep feedforward neural networks".

Инициализация по методу Xavier дает хорошие результаты при использовании сигмовидных функций активации. Но при использовании ReLU в качестве функции активации она не так эффективна. Это связано с особенностями самой ReLU.

$$ReLU(x_i) = max(0, x_i)$$

Так как ReLU пропускает только положительные значения взвешенной суммы, а отрицательные обнуляет, то по теории вероятности половина нейронов большую часть времени будет деактивирована. Следовательно, нейроны последующего слоя получат только половину информации, а взвешенная сумма входов на них будет меньше. С ростом количества слоев нейронной сети эффект будет усиливаться: все меньше нейронов будут набирать пороговое значение, и все больше информации будет теряться по пути прохождения через нейронную сеть.

Решение было предложено Каймин Хе (Kaiming He) в феврале 2015 года в статье "Delving Deep into Rectifiers: Surpassing Human—Level Performance on ImageNet Classification". В статье предложено веса для нейронов с активацией ReLU инициализировать из непрерывного нормального распределения с дисперсией (δ^2), равной 2/n. А при использовании PReLU в качестве активации дисперсия распределения должна составлять $2/((1+a^2)*n)$. Данный метод инициализации синаптических весов получил название «Хе-инициализация».

Инициализация случайной ортогональной матрицей

В декабре 2013 года Эндрю Сакс (Andrew M. Saxe) в статье "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks" представил трехслойную нейронную сеть в виде матричного умножения и таким образом показал соответствие нейронной сети и сингулярного разложения. Матрица синаптических весов первого слоя представляется ортогональной матрицей, векторы которых являются координатами исходных данных в некоем *n*-мерном пространстве.

Так как векторы ортогональной матрицы являются ортонормированными, то и создаваемые ими проекции исходных данных являются абсолютно независимыми. Такой подход позволяет заранее подготовить нейросеть таким образом, что каждый нейрон будет обучаться распознавать свой признак во входных данных независимо от обучения других нейронов, находящихся в том же слое.

Несмотря на всю привлекательность метода он используется не так уж часто. Это связано прежде всего со сложностью генерации ортогональных матриц. Преимущества же метода проявляются с ростом количества слоев нейронной сети. Поэтому инициализацию ортогональными матрицами на практике можно встретить в глубоких нейронных сетях, когда инициализация случайными значениями не дает результата.

Использование предварительно обученных нейронных сетей

Наверно этот метод сложно назвать инициализацией, но все чаще можно встретить его практическое применение. Суть метода заключается в том, что для решения задачи используют нейронную сеть, обученную на тех же или похожих данных, но решающую другие задачи. Из ранее обученной нейронной сети берут ряд нижних слоев, которые уже обучены выделять признаки из исходных данных, и добавляют несколько новых слоев нейронов, которые будут решать поставленную задачу на основе уже выделенных признаков.

На начальном этапе блокируют обучение предварительно обученных слоев и обучают новые слои. Если в процессе обучения не удается получить желаемый результат, снимают блокировку обучения с заимствованных нейронных слоев и проводят дообучение нейронной сети.

Разновидностью данного метода является подход, когда вначале создают нейронную сеть с несколькими слоями и проводят ее обучение на выделение различных признаков из исходных данных. Это могут быть алгоритмы обучения без учителя по разделению данных на классы или алгоритмы-автоэнкодеры. В последних нейронная сеть сначала выделяет признаки из исходных данных, а потом на основании выделенных признаков пытается вернуть исходные данные.

После предварительного обучения берутся слои нейронов, ответственных за выделение признаков, и к ним добавляются слои нейронов для решения поставленной задачи.

При создании глубоких сетей такой подход поможет обучить нейронную сеть быстрее, чем при прямом обучении большой нейронной сети сразу. Это связано с тем, что при одном проходе обучения малой нейронной сети требуется выполнить меньше операций, чем при обучении глубокой нейронной сети. При этом малые нейронные сети менее подвержены риску взрыва или затухания градиента.

В практической части книге мы еще вернемся к процессу инициализации нейронных сетей и на практике оценим преимущества и недостатки каждого из методов.

1.4 Обучение нейронной сети

Мы уже познакомились со строением искусственного нейрона, узнали об организации обмена данными между нейронами и о принципах построения нейронных сетей. Также мы научились инициализировать синаптические коэффициенты. Следующим шагом будет обучение нейронной сети.

Том Митчелл (Tom Mitchell) предложил следующее определение машинному обучению:

> «Компьютерная программа учится на опыте E в отношении некоторого класса задач T и показателя производительности P, если ее производительность при выполнении задач в T, измеренная с помощью P, улучшается с опытом E».

Принято выделять три основных подхода к обучению нейронных сетей:

- Обучение с учителем;
- Обучение без учителя;
- Обучение с подкреплением.

Алгоритмы *обучения с учителем* на практике дают наилучшие результаты, но они требуют большой подготовительной работы. Сам принцип обучения с учителем подразумевает наличие правильных ответов. Как учитель, на каждой итерации обучения мы будем направлять нейронную сеть, показывая ей правильные результаты, тем самым призывая нейронную сеть запомнить, что верно, а что ложь.

При таком подходе внутри нейронной сети настраиваются связи корреляции исходных данных с правильными ответами. В идеале нейронная сеть должна научиться выделять существенные признаки из набора исходных данных. Обобщая набор выделенных признаков, она должна определять принадлежность объекта к тому или иному классу (задачи классификации) или указать наиболее вероятное развитие событий (задачи регрессии).

Сложность такого подхода заключается в необходимости проведения огромной подготовительной работы. Данный подход требует сопоставление правильных ответов к каждому набору исходных данных из обучающей выборки. Не всегда эту работу можно автоматизировать, приходится привлекать человеческие ресурсы. В то же время использование рабочей силы по подготовке обучающей выборки и правильных ответов повышает риск наличия ошибок в выборке и, как следствие, неправильной настройки нейронной сети.

Еще один риск такого подхода — переобучение нейронной сети. Это явление обычно проявляется при обучении глубоких сетей на малом наборе исходных данных. В таком случае нейронная сеть в состоянии «запомнить» все пары наборов исходных данных с правильными

ответами. При этом она утратит всякую способность к обобщению данных. В результате мы получим нейронную сеть с прекрасными результатами на тренировочном наборе данных и абсолютно случайными ответами на тестовой выборке и при ее эксплуатации на реальных данных.

Для снижения риска переобучения нейронных сетей используют различные методы регуляризации, нормализации и дропаута, о которых мы поговорим позже.

Также не стоит забывать о возможности столкнуться с задачей, в которой нет однозначного правильного ответа для представленных наборов данных. В подобных случаях используются другие подходы к обучению нейронных сетей.

Обучение без учителя применяется при отсутствии правильных ответов для обучающей выборки. Алгоритмы обучения без учителя позволяют выделять отдельные признаки объектов исходных данных. Сравнивая выделенные признаки, алгоритмы проводят кластеризацию исходных данных, объединяя наиболее похожие объекты в некие классы. Количество таких классов задается в гиперпараметрах нейронной сети.

За экономию затрат на подготовительном этапе приходится платить качеством распознавания объектов и более узким кругом решаемых задач.

С ростом объема исходных данных для обучения широко используются алгоритмы обучения без учителя для предварительного обучения нейронных сетей. Вначале создается нейронная сеть и обучается без учителя на большой выборке. Это позволяет научить нейронную сеть выделять отдельные признаки из набора исходных данных и разделить большой объем данных на отдельные классы объектов.

Затем к предварительно обученной сети добавляются нейронные слои принятия решений (чаще всего полносвязные нейронные слои перцептрона) и проводится дообучение нейронной сети алгоритмами обучения с учителем.

Такой подход позволяет обучить нейронную сеть на большом объеме исходных данных, что помогает снизить до минимума риск переобучения нейронной сети. При этом, так как обучение на основном массиве данных проходит без учителя, мы можем дообучить глубокую нейронную сеть на сравнительно небольшом наборе пар исходных данных с правильными ответами. Это снижает ресурсы, требуемые для проведения подготовительной работы при обучении с учителем.

Отдельным подходом к обучению нейронных сетей можно назвать *обучение с подкреплением*. Данный подход применяется для решения оптимизационных задач, требующих построения стратегии. Наилучшие результаты демонстрируются при обучении нейронных сетей компьютерным и логическим играм, для чего и был разработан этот метод. Он применим для длительных конечных процессов, когда в течении процесса от нейронной сети требуется ряд решений в зависимости от состояния окружающей среды, а суммарный результат принятых решений будет ясен только в конце процесса. К примеру, выигрыш или проигрыш в игре.

Суть метода заключается в присваивании некой награды или штрафа за каждое действие. В процессе обучения определяется стратегия с максимальной наградой.

В данной книге больше внимания будет уделено обучению с учителем, которое на практике показывает наилучшие результаты обучения и применим для решения регрессионных задач, к которым относится и прогнозирование временных рядов.

1.4.1 Функции потерь

Приступая к обучению, необходимо определиться с методами определения качества обучения сети. Обучение нейронной сети — это итерационный процесс. На каждой итерации нам необходимо определить, насколько расчеты нейронной сети точны. В случае обучения с учителем — на сколько они отличаются от эталона. Только зная отклонение, мы сможем понять, насколько и в какую сторону нам нужно корректировать синаптические коэффициенты.

Следовательно, нам нужна некая метрика, которая непредвзято и с математической точностью покажет погрешность работы нейронной сети.

На первый взгляд, это довольно тривиальная задача — сравнить два числа (расчетное значение нейронной сети и целевое). Но, как правило, на выходе нейронной сети мы получаем не одно значение, а целый вектор. Для решения этой задачи обратимся к математической статистике. Введем некую *функцию потерь*, которая зависит от расчетного значения (*y'*) и эталонного (*y*).

$$L(y, y')$$

Данная функция должна определить отклонение расчетного значения от эталонного (ошибку). Если рассматривать расчетное и эталонное значения как точки в пространстве, то ошибка является расстоянием между этими точками. Следовательно, функция потерь при всех допустимых значениях должна быть непрерывной и неотрицательной.

В идеальном состоянии расчетное и эталонное значения совпадают, расстояние между точками равно нулю. Следовательно, функция должна быть выпуклой вниз с минимумом в $L(y,y')=0$.

В книге Л. Б. Клебанова "Robust and Non-Robust Models in Statistics" описано четыре свойства, которыми должна обладать функция потерь:

- Полнота информации;
- Отсутствие условия рандомизации;
- Условие симметризации;
- Состояние Рао-Блэквелла (статистические оценки параметров могут быть улучшены).

В книге приводится довольно много математических теорем и их доказательств. Демонстрируется взаимосвязь между выбором функции потерь и статистической оценкой. Как следствие, некоторые статистические проблемы могут быть решены при правильном выборе функции потерь.

Средняя абсолютная ошибка (MAE)

Одну из первых функций потерь — среднюю абсолютную ошибку (в английской аббревиатуре MAE) — представил французский математик 18-го века Пьер-Симон де Лаплас. Он предложил для оценки отклонения использовать модуль разницы между эталонным и расчетным значениями.

$$L(y, y') = \frac{1}{n}\sum_{i=1}^{n}\left|y_i - y_i'\right|$$

Функция имеет график, симметричный относительно нуля, линейный до и после нуля.

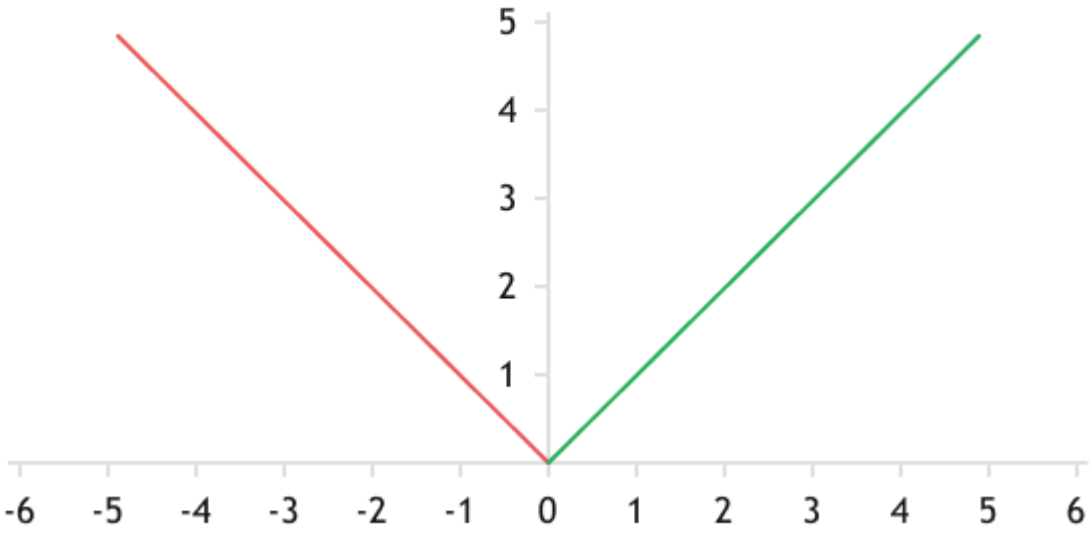


График функции среднего абсолютного отклонения

Использование среднего абсолютного отклонения дает линейное приближение аналитической функции к обучающей выборке на всей дистанции ошибки.

Посмотрим на вариант реализации данной функции в программном коде *MQL5*. Для расчета отклонений функция должна получить на вход два вектора данных: расчетных и эталонных значений. Эти данные будем передавать в параметрах функции.

В начале метода сравним размеры полученных массивов. В идеале размеры массивов должны быть не меньше нуля. Если проверка не пройдена, то выйдем из функции с результатом максимально возможной ошибки *DBL_MAX*.

```
double MAE(double &calculated[], double &target[])
  {
   double result = DBL_MAX;
//---
   if(calculated.Sizel() < target.Sizel() || target.Sizel() <= 0)
      return result;
```

После успешного прохождения проверок организуем цикл для суммирования абсолютных значений отклонений. В заключение разделим полученную сумму на количество эталонных значений.

```
//---
   result = 0;
   int total = target.Size();
   for(int i = 0; i < total; i++)
      result += MathAbs(calculated[i] - target[i]);
   result /= total;
//---
   return result;
  }
```

Среднеквадратичная ошибка (MSE)

Немецкий математик 19-го века Карл Фридрих Гаусс предложил в формуле среднего абсолютного отклонения вместо модуля отклонения использовать его квадрат. Функция получила название среднеквадратичного отклонения.

$$L(y,y') = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - y_i'\right)^2$$

Благодаря квадрату отклонения функция ошибки приобретает вид параболы.

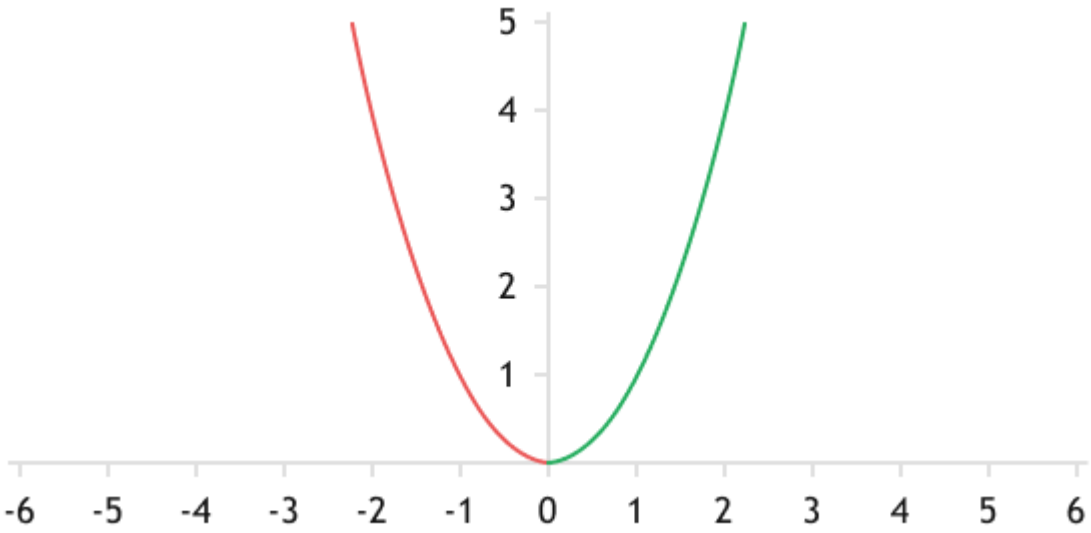


График функции среднеквадратичного отклонения

При использовании среднеквадратичного отклонения скорость компенсации ошибки тем выше, чем больше сама ошибка. При снижении ошибки падает и скорость ее компенсации. В случае с нейронными сетями это позволяет быстрее сходиться нейронной сети при больших ошибках и проводить более тонкую настройку при малых ошибках.

Но есть и обратная сторона медали: указанное выше свойство делает функцию чувствительной к шумовым явлением, так как редкое, большое отклонение вызывает смещение функции.

В настоящее время использование среднеквадратичной ошибки в качестве функции потерь широко используется при решении задач регрессии.

Алгоритм реализации *MSE* на *MQL5* аналогичен реализации *MAE*. Отличие лишь в теле цикла, где вычисляется сумма квадратов отклонений вместо их абсолютных значений.

```
double MSE(double &calculated[], double &target[])
  {
   double result = DBL_MAX;
//---
   if(calculated.Size() < target.Size() || target.Size() <= 0)
      return result;
//---
   result = 0;
   int total = target.Size();
   for(int i = 0; i < total; i++)
      result += MathPow(calculated[i] - target[i], 2);
   result /= total;
//---
   return result;
  }
```

Кросс-энтропия

Для решения задач классификации в качестве функции потерь чаще всего используют функцию кросс-энтропии.

Энтропия — это мера неопределенности в распределении.

Применение энтропии переводит расчеты из плоскости абсолютных значений в область вероятностей. Кросс-энтропия определяет сходство вероятностей проявления событий двух распределений и вычисляется по формуле:

$$L(y, y') = -\sum_{i=1}^{n} p(y_i) * log(p(y_i')),$$

где:

- $p(y_i)$ — вероятность появления i-го события в эталонном распределении;
- $p(y_i')$ — вероятность появления i-го события в расчетном распределении.

Так как мы исследуем вероятности появления событий, то значение вероятностей появления события всегда лежит в диапазоне значений от 0 до 1. Значение логарифма в данном диапазоне отрицательное, поэтому добавление знака минуса перед функцией переводит значение функции в положительную область и делает функцию строго убывающей. Для наглядности график логарифмической функции представлен ниже.

В процессе обучения для событий эталонного распределения при появлении события его вероятность равна единице. Вероятность появления отсутствующего события равна нулю. Исходя из графика функции, наибольшую ошибку будет генерировать событие, произошедшее в эталонном распределении и не предсказанное аналитической функцией. Таким образом, мы будем стимулировать нейронную сеть к предсказанию ожидаемых событий.

Именно применение вероятностной модели делает эту функцию наиболее привлекательной для решения задач классификации.

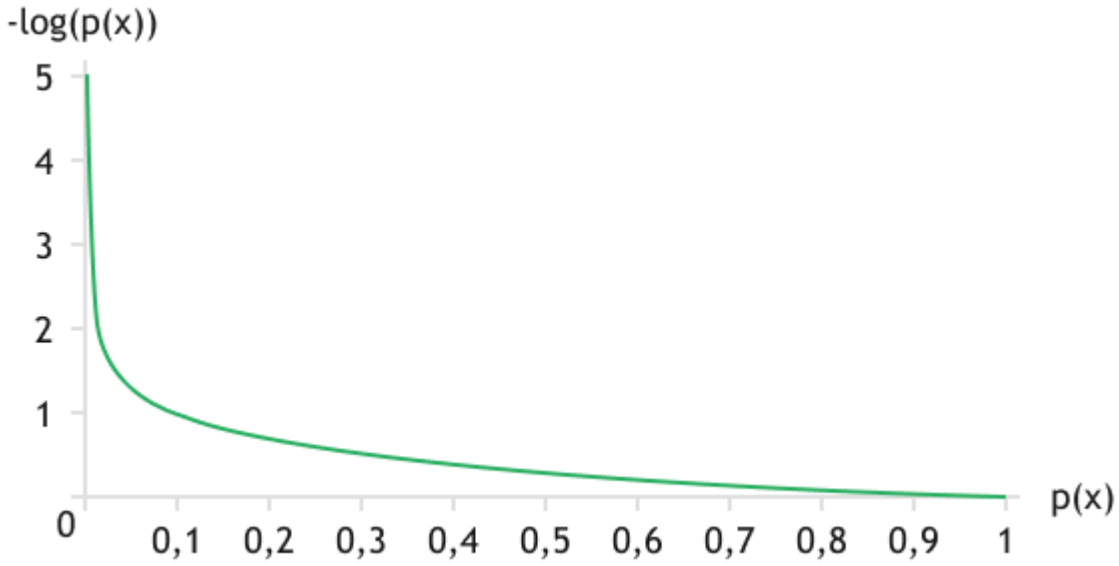


График функции логарифма

Вариант реализации данной функции представлен ниже. Алгоритм реализации аналогичен двум предыдущим функциям.

```
double LogLoss(double &calculated[], double &target[])
  {
   double result = DBL_MAX;
//---
   if(calculated.Size() < target.Size() || target.Size() <= 0)
      return result;
//---
   result = 0;
   int total = target.Size();
   for(int i = 0; i < total; i++)
      result -= target[i] * MathLog(calculated[i]);
//---
   return result;
  }
```

Выше представлено описание лишь трех наиболее часто используемых функций потерь. Но на самом деле их число значительно больше. И тут, как и в случае функций активации, нам помогут векторные и матричные операции, реализованные в языке *MQL5*, среди которых реализована функция *Loss*. Данная функция позволяет одной строкой кода вычислить функцию потерь между двумя векторами/матрицами одинакового размера. Функция вызывается для вектора или матрицы расчетных значений. В параметрах функции передаются вектор/матрица эталонных значений и тип функции потерь.

```
double vector::Loss(
  const vector&       vect_true,    // вектор истинных значений
  ENUM_LOSS_FUNCTION  loss          // тип функции потерь
   );

double matrix::Loss(
  const matrix&       matrix_true,  // матрица истинных значений
  ENUM_LOSS_FUNCTION  loss          // тип функции потерь
```

```
);
```

Разработчики компании MetaQuotes реализовали 14 различных функций потерь. Их список представлен в таблице ниже.

Идентификатор	Описание
LOSS_MSE	Среднеквадратичная ошибка
LOSS_MAE	Средняя абсолютная ошибка
LOSS_CCE	Категориальная кросс-энтропия
LOSS_BCE	Бинарная кросс-энтропия
LOSS_MAPE	Средняя абсолютная ошибка в процентах
LOSS_MSLE	Среднеквадратичная логарифмическая ошибка
LOSS_KLD	Дивергенция Кульбака-Лейблера
LOSS_COSINE	Косинусное сходство/близость
LOSS_POISSON	Функция потерь Пуассона
LOSS_HINGE	Кусочно-линейная функция потерь (Hinge loss)
LOSS_SQ_HINGE	Квадратичная кусочно-линейная функция потерь
LOSS_CAT_HINGE	Категориальная кусочно-линейная функция потерь
LOSS_LOG_COSH	Логарифм гиперболического косинуса
LOSS_HUBER	Функция потерь Хьубера

1.4.2 Метод обратного распространения градиента ошибки

После того, как мы определились с функцией потерь, можно перейти к обучению нейронной сети. Сам процесс обучения сводится к итерационному подбору параметров нейронной сети (синаптических коэффициентов), при которых значение функции потерь нейронной сети будет минимально.

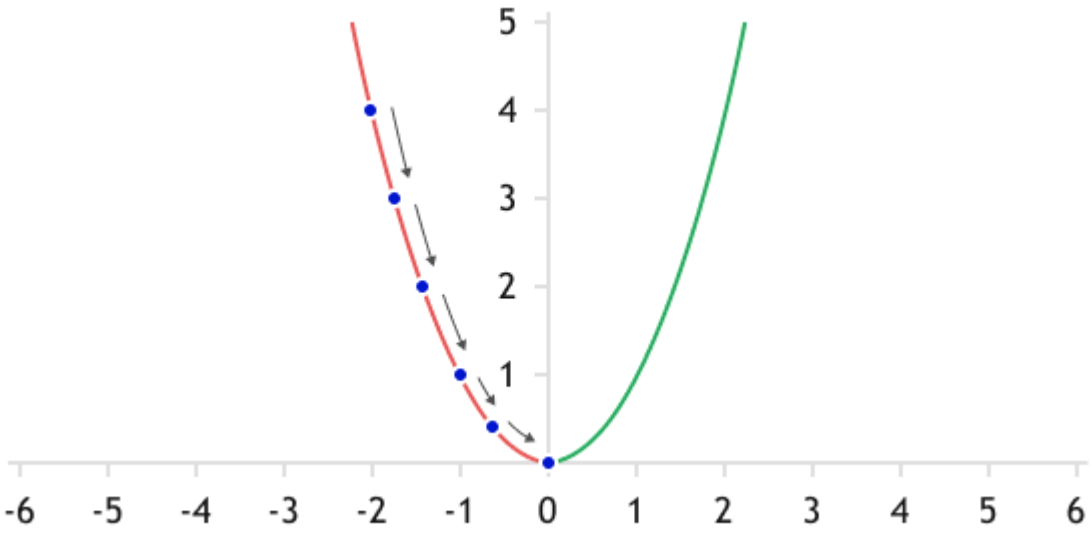
Из прошлого раздела мы узнали, что функция потерь подбирается выпуклой вниз. Следовательно, начиная обучение из любой точки на графике функции потерь, мы должны двигаться в сторону минимизации ошибки. Для сложных функций, таких как нейронная сеть, наиболее удобным способом является метод градиентного спуска.

Градиентом функции многих переменных (которой является нейронная сеть) называется вектор, состоящий из частных производных функции по ее аргументам. Из курса математики мы знаем, что производная функции характеризует скорость изменения функции в данной точке.

Следовательно, градиент указывает направление наискорейшего роста функции. Двигаясь в направлении антиградиента (обратном к градиенту), мы будем спускаться с максимальной скоростью к минимуму функции.

Наш алгоритм действий будет следующим:

1. Инициализируем весовые коэффициенты нейронной сети одним из способов, описанных ранее.
2. Вычисляем прогнозные данные на обучающей выборке.
3. С помощью функции потерь вычисляем погрешность вычислений нейронной сети.
4. Определяем градиент функции потерь в полученной точке.
5. Корректируем синаптические коэффициенты нейронной сети в сторону антиградиента.



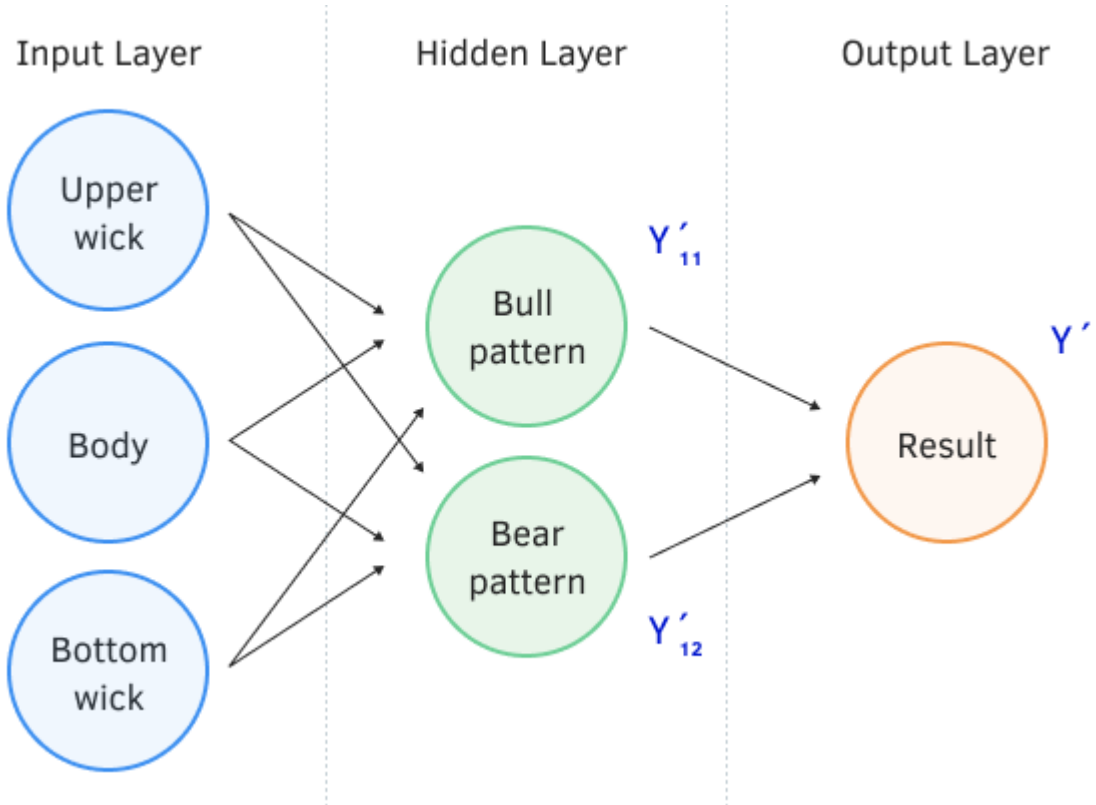
Градиентный спуск

Так как чаще всего будет использоваться нелинейная функция потерь, то и в каждой точке на графике функции потерь направление вектора антиградиента будет меняться. Поэтому, снижаться к минимуму функции потерь мы будем постепенно, с каждой итерацией все ближе и ближе к минимуму.

На первый взгляд, алгоритм довольно прост и логичен. Но как нам технически реализовать пункт 5 нашего алгоритма в случае многослойной нейронной сети?

Данный вопрос решается с применением алгоритма обратного распространения градиента ошибки, который состоит из двух блоков:

1. *Прямой проход*. Пункт 2 из нашего алгоритма выше. Во время прямого прохода на вход нейронной сети подается набор данных из обучающей выборки и производится их обработка в нейронной сети последовательно от входного до выходного слоя. При этом сохраняются промежуточные значения на каждом нейроне.



Прямой проход нейронной сети

2. *Обратный проход* включает пункты 3–5 нашего алгоритма.

В этом месте стоит немного вспомнить математику. Мы говорим о частных производных функции, но при этом хотим обучить нейронную сеть, которая состоит из большого количества нейронов. При этом каждый нейрон представляет собой сложную функцию, и чтобы обновить весовые коэффициенты нейронной сети, нам необходимо посчитать частные производные сложной функции нашей нейронной сети по каждому весовому коэффициенту.

По правилам математики, производная сложной функции равна произведению частной производной внешней функции на частную производную внутренней функции.

$$\frac{dF(G(x))}{dx} = \frac{dF(G(x))}{dG(x)}\frac{dG(x)}{dx}$$

Воспользуемся этим правилом и найдем частные производные функции потерь L по весовому коэффициенту выходного нейрона w_i и по i-му входному значению x_i.

$$\frac{dL(A(S(X,W)))}{dw_i} = \frac{dL(A(S(X,W)))}{dA(S(X,W))}\frac{dA(S(X,W))}{dS(X,W)}\frac{dS(X,W)}{dw_i},$$
$$\frac{dL(A(S(X,W)))}{dx_i} = \frac{dL(A(S(X,W)))}{dA(S(X,W))}\frac{dA(S(X,W))}{dS(X,W)}\frac{dS(X,W)}{dx_i},$$

где:

- ***L*** — функция потерь;
- ***A*** — функция активации нейрона;

- S — взвешенная сумма исходных данных;
- X — вектор исходных данных;
- W — вектор весовых коэффициентов;
- w_i — *i*-ый весовой коэффициент, для которого рассчитывается производная:
- x_i — *i*-ый элемент вектора исходных данных.

Первое, что можно заметить в представленных выше формулах, это полное совпадение первых двух множителей. Т.е. при расчете частных производных по весовым коэффициентам и исходным данным нам достаточно один раз посчитать градиент ошибки перед функцией активации, и уже используя это значение, посчитать частные производные для всех элементов векторов весовых коэффициентов и исходных данных.

Аналогичным методом определим частную производную по весовому коэффициенту одного из нейронов скрытого слоя, предшествовавшего выходному нейронному слою. Для этого в предыдущей формуле вектор исходных данных заменим функцией нейрона скрытого слоя. Вектор весовых коэффициентов обернется в скалярное значение соответствующего весового коэффициента.

$$\frac{dL(A(S(A_h(S_h(X_h,W_h)),w_i)))}{dw_h}=\frac{dL(A(S(A_h(S_h(X_h,W_h)),w_i)))}{dA(S(A_h(S_h(X_h,W_h)),w_i))}\frac{dA(S(A_h(S_h(X_h,W_h)),w_i))}{dS(A_h(S_h(X_h,W_h)),w_i)}\dots\frac{dS_h(X_h,W_h)}{dw_h},$$

где:

- A_h — функция активации нейрона скрытого слоя;
- S_h — взвешенная сумма исходных данных нейрона скрытого слоя;
- X_h — вектор исходных данных для нейрона скрытого слоя;
- W_h — вектор весовых коэффициентов нейрона скрытого слоя;
- w_h — весовой коэффициент скрытого слоя, для которого рассчитывается производная.

Обратите внимание, что если в последней формуле мы вернем X вместо функции нейрона скрытого слоя, то в первых множителях функции увидим представленную выше функцию частной производной по *i*-му входному значению.

$$X=A_h(S_h(X_h,W_h)),$$

$$\frac{dL(A(S(A_h(S_h(X_h,W_h)),w_i)))}{dw_h}=\frac{dL(A(S(X,W)))}{dA(S(X,W))}\frac{dA(S(X,W))}{dS(X,W)}\frac{dS(X,W))}{dx_i}\dots\frac{dS_h(X_h,W_h)}{dw_h}.$$

Отсюда,

$$\frac{dL(A(S(A_h(S_h(X_h,W_h)),w_i)))}{dw_h}=\frac{dL(A(S(X,W)))}{dx_i}\frac{dA_h(S_h(X_h,W_h))}{dS_h(X_h,W_h)}\frac{dS_h(X_h,W_h)}{dw_h}.$$

Аналогичные формулы можно привести для каждого нейрона нашей сети. Таким образом, мы можем один раз посчитать производную и градиент ошибки на выходе нейрона, а далее распространить градиент ошибки на все связанные нейроны предыдущего слоя.

Следуя этой логике, вначале определяем отклонения от эталонного значения с помощью функции потерь. Функция потерь может быть любая, удовлетворяющая описанным в предыдущем разделе требованиям.

$$L(Y, Y'),$$

где:

- ***Y*** — вектор эталонных значений;
- ***Y'*** — вектор значений на выходе нейронной сети.

Далее мы определяем, как должны измениться состояния нейронов выходного слоя, чтобы наша функция потерь принимала минимальное значение. С математической точки зрения мы определяем градиент ошибки на каждом нейроне выходного слоя путем вычисления частной производной функции потерь по каждому параметру.

$$grad_i^{out} = \frac{\delta L(Y, Y')}{\delta y_i'} * L(Y, Y')$$

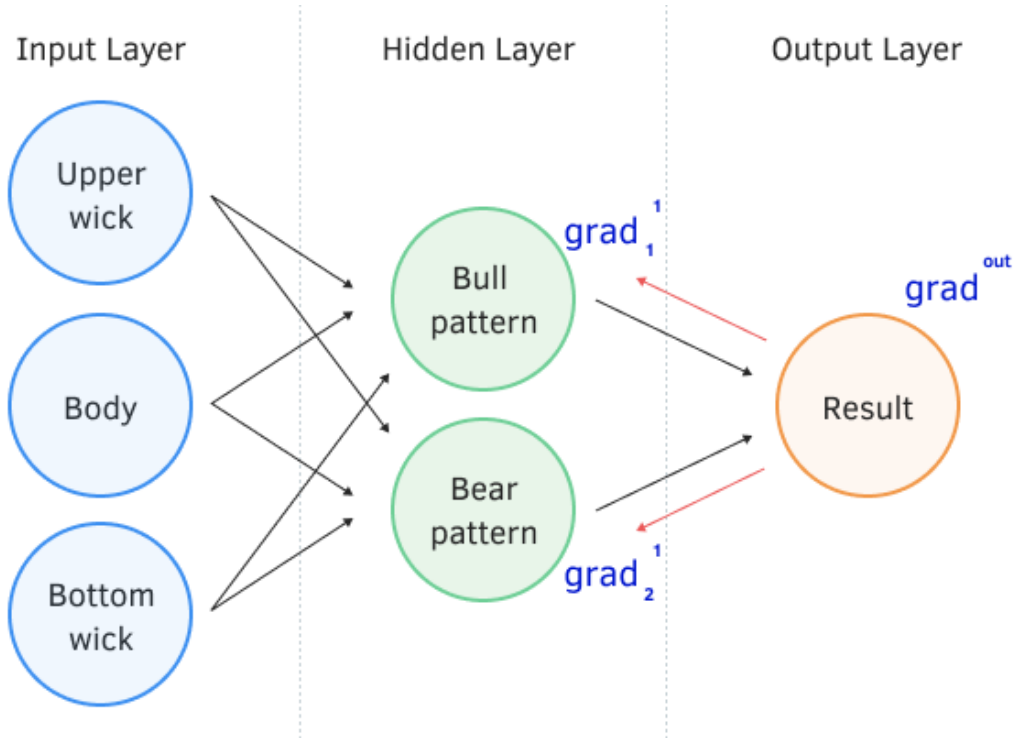
Затем мы «спускаем» градиент ошибки от выходного нейронного слоя к входному, проводя его последовательно через все скрытые нейронные слои нашей сети. Таким образом, мы как бы приводим до каждого нейрона его эталонное значение на данном шаге обучения.

$$grad_i^{j-1} = \sum_{k=1}^{n} \frac{\delta A_{kj}(S_{kj}(W_{kj}Y'_{j-1}))}{\delta y'_{ij-1}} grad_k^j,$$

где:

- $grad_i^{j-1}$ — градиент на выходе *i*-го нейрона *j*—1 слоя;
- A_{kj} — функция активации *k*-го нейрона на *j*-ом слое;
- S_{kj} — взвешенная сумма входящих данных *k*-го нейрона на *j*-ом слое;
- W_{kj} — вектор синаптических коэффициентов *k*-го нейрона на *j*-ом слое.

После получения градиентов ошибки на выходе каждого нейрона можно приступать к корректировке синаптических коэффициентов нейронов. Для этого еще раз последовательно пройдемся через все слои нейронной сети. При этом на каждом слое будем перебирать все нейроны и для каждого нейрона будем обновлять все синаптические связи.



Обратный проход нейронной сети

О способах обновления весовых коэффициентов поговорим в следующей главе.

После обновления весовых коэффициентов возвращаемся к пункту 2 нашего алгоритма. Цикл повторяется до нахождения минимума функции. Определить достижение минимума можно по нулевым частным производным. В общем виде это будет заметно по отсутствию изменения ошибки на выходе нейронной сети после очередного цикла обновления весовых коэффициентов, т.к. при нулевых производных останавливается процесс обучения нейронной сети.

1.4.3 Методы оптимизации нейронных сетей

Мы продолжаем двигаться дальше по пути изучения базовых принципов обучения нейронных сетей. В предыдущих главах мы уже рассмотрели варианты функций потерь и алгоритм обратного распространения градиента ошибки, который позволяет определить влияние каждого нейрона на общий результат и направление изменения выходного значения каждого нейрона для минимизации общей ошибки на выходе.

Напомню формулу математической модели нейрона.

$$OUT = f(W, X) = f\left(\sum_{i=1}^{n} w_i x_i\right),$$

где:

- f — функция активации;
- n — количество элементов во входной последовательности;
- w_i — весовой коэффициент i-го элемента последовательности;
- x_i — i-ый элемент входной последовательности.

Градиентный спуск и стохастический градиентный спуск

В приведенной выше формуле видно, что выходное значение нейрона зависит от функции активации, входной последовательности и вектора весовых коэффициентов. Формула активации задается при конструировании нейронной сети и является неизменной. На входную последовательность нейрон влияние не оказывает. Следовательно, в процессе обучения изменить значение на выходе нейрона мы можем, только подобрав оптимальные значения весовых коэффициентов.

Скорость изменения выходного значения нейрона при изменении отдельно взятого весового коэффициента равна частной производной функции математической модели нейрона по данному весовому коэффициенту. Отсюда, чтобы получить дельту изменения отдельно взятого весового коэффициента, нужно умножить ошибку на выходе нейрона в текущей ситуации на частную производную математической модели нейрона по данному весовому коэффициенту.

Следует отметить, что функция математической модели нейрона чаще всего нелинейна. И, следовательно, частная производная не является постоянной на всем допустимом интервале значений. Поэтому в формулу обновления весовых коэффициентов вводится параметр «коэффициент обучения», который определяет скорость обучения.

Описанный выше подход называется ***градиентным спуском***. В общем случае его можно выразить формулой:

$$w_{il}^{j} = w_{il}^{j} - \alpha \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^{j}} grad_{i}^{j},$$

где:

- w_{il}^{j} — *l*-ый весовой коэффициент на *i*-ом нейроне *j*-го слоя;
- α — обучающий коэффициент, определяющий скорость обучения;
- $grad_{i}^{j}$ — градиент на выходе *i*-го нейрона *j*-го слоя.

Обучающий коэффициент α является гиперпараметром, который подбирается в процессе валидации нейронной сети. Он выбирается из диапазона от 0 до 1. При этом обучающий коэффициент не может быть равен крайним значениям.

При α=0 не будет никакого обучения, так как при умножении любого градиента на ноль мы всегда получим ноль для обновления весовых коэффициентов.

При α=1 или близких к этому значению нас ждет другая проблема. Если при движении в сторону минимума ошибки значение частной производной снижается, то использование достаточно большого шага перекинет нас через точку минимума. А в худшем случае ошибка на выходе нейронной сети еще и увеличится. Кроме того, большой обучающий коэффициент способствует максимальной адаптации сети к текущей ситуации. При этом теряется способность к обобщению. Такое обучение не сможет выявить ключевые признаки и адекватно работать «в полевых условиях».

С небольшими нейронными сетями и маленькими наборами данных метод легко работает. Но нейронные сети становятся все больше, обучающие выборки растут. Для того чтобы сделать одну итерацию обучения, нужно провести прямой проход по всей выборке и сохранить информацию о состоянии всех нейронов для всех выборок — эта информация понадобится для

проведения обратного прохода. Следовательно, нам потребуется дополнительное выделение памяти в размере количество нейронов * количество наборов данных.

Решение было найдено в применении ***стохастического градиентного спуска***. Метод сохраняет алгоритм стандартного градиентного спуска, но обновление весовых коэффициентов проводится для каждого случайно взятого набора данных.

На каждой итерации из обучающей выборки случайным образом выбираем один набор данных. Далее осуществляем прямой и обратный проход и обновляем весовые коэффициенты. Затем «перемешиваем» обучающую выборку и выбираем следующий набор данных случайным образом.

Итерации повторяем до получения приемлемой ошибки на выходе нейронной сети.

Стохастический градиентный спуск обладает меньшей сходимостью, по сравнению со стандартным градиентном спуском. И, как правило, для достижения приемлемой ошибки требуется больше итераций. Но при стохастическом градиентном спуске отдельно взятая итерация требует меньше времени, поскольку осуществляется на случайно взятом наборе данных, а не на всей выборке, как при стандартном градиентном спуске. В целом процесс обучения нейронной сети осуществляется с меньшими временными и ресурсными затратами.

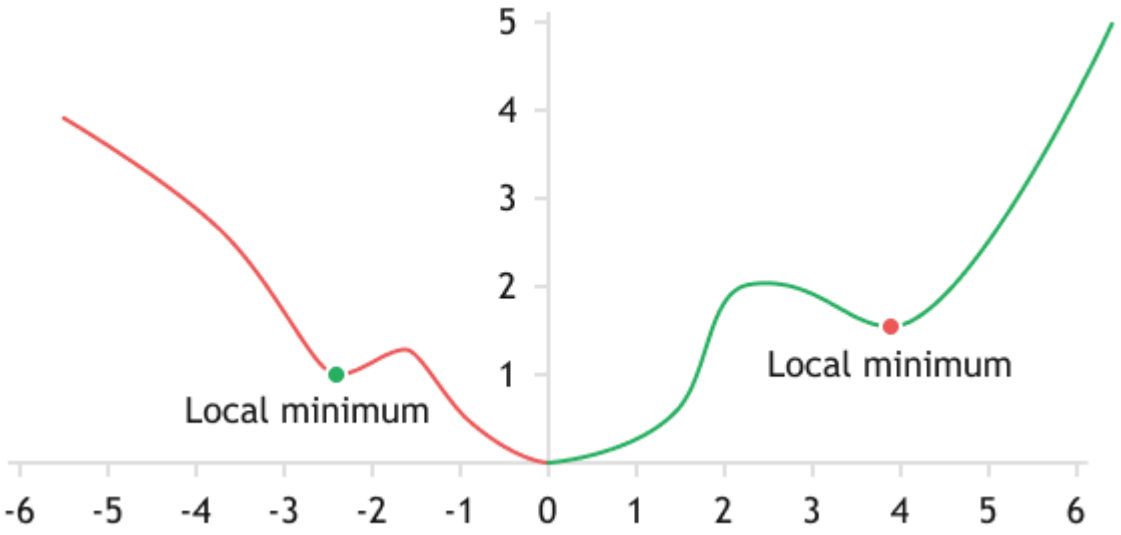
Ниже приведен пример реализации функции обновления весовых коэффициентов по методу градиентного спуска. В параметрах функция получает два указателя на массивы данных (текущих весовых коэффициентов и градиентов к ним) и обучающий коэффициент. Вначале проверяем соответствие размеров массивов. Затем организуем цикл, в котором для каждого элемента массива весов рассчитываем новое значение по приведенной выше формуле. Полученное значение сохраняем в соответствующей ячейке массива.

```
bool SGDUpdate(double &m_cWeights[],
               double &m_cGradients[],
               double learningRate)
  {
   if(m_cWeights.Size() > m_cGradients.Size()  ||
      m_cWeights.Size()<= 0)
      return false;
//---
   for(int i = 0; i < m_cWeights.Size(); i++)
      m_cWeights[i] -= learningRate * m_cGradients[i];
   return true;
  }
```

Метод накопления импульса (Momentum)

Наверное, основным недостатком методов градиентного спуска является невозможность разделения локальных и глобального минимумов. В процессе обучения всегда остается большой риск остановиться в локальном минимуме, так и не достигнув желаемого уровня точности работы нейронной сети.

Тщательный подбор коэффициента обучения, эксперименты с различными вариантами инициализации весовых коэффициентов и несколько итераций обучения не всегда позволяют добиться желаемых результатов.



Локальные минимумы на графике сложной функции

Один из вариантов решения был заимствован из физики природных явлений. Если положить мячик в небольшое углубление или ямку, то он будет лежать без движения на ее дне. Но стоит нам тот же мячик спустить по некой наклонной поверхности в эту ямку, то он с легкостью выскочит из нее и покатится дальше. Такое поведение мячика объясняется импульсом, накопленным при спуске по наклонной поверхности.

Аналогично мячику, было предложено добавить накопление импульса при обучении весовых коэффициентов, и затем добавлять этот импульс в формулу обновления весовых коэффициентов методом градиентного спуска.

Импульсы накапливаются по каждому отдельному весовому коэффициенту. При длительном обновлении отдельно взятого весового коэффициента в одном направлении его импульс будет накапливаться и, как следствие, он будет двигаться с большей скоростью к заветной цели. Благодаря накопленной энергии мы сможем преодолевать локальные минимумы, подобно мячику, пущенному по наклонной поверхности.

К сожалению, есть и обратная сторона медали. Накапливая импульс, мы проскочим не только локальный минимум, но и глобальный. При неограниченном накоплении импульса значение нашей ошибки будет двигаться подобно маятнику по графику функции потерь. По аналогии с силой трения в природе, добавим коэффициент β в диапазоне от 0 до 1 (не включая граничные точки), который и будет выполнять роль силы трения. Данный коэффициент характеризует скорость затухания импульса. Чем ближе β к "1", тем дольше сохраняется импульс.

Все описанное выше можно записать в виде двух математических формул:

$$\Delta_t = \alpha \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^j} grad_i^j + \beta \Delta_{t-1}$$

$$w_{il}^j = w_{il}^j - \Delta_t$$

где:

- Δ_t — изменение весового коэффициента на текущем шаге;

- Δ_{t-1} — изменение весового коэффициента на предыдущей итерации обучения;
- β — коэффициент затухания импульса.

В результате мы получили неплохой алгоритм для борьбы с локальными минимума. Но, к сожалению, и он не является панацеей. Для преодоления локального минимума нужно накопить достаточный импульс. Для этого инициализация должна быть на достаточной дистанции от локального минимума. При решении практических задач мы не знаем, где в действительности находятся локальные и глобальный минимумы. Да и инициализация весовых коэффициентов случайным образом может забросить нас куда угодно.

К тому же применение данного метода требует дополнительного объема памяти для хранения последнего импульса каждого нейрона и дополнительных трудозатрат на выполнение добавленных вычислений.

Все же метод накопления импульса (Momentum) используется на практике и демонстрирует лучшую сходимость, по сравнению со стохастическим градиентным спуском.

При реализации метода импульсного накопления нам потребуется добавить в параметры функции коэффициент затухания и еще один массив для хранения накопленного импульса по каждому весовому коэффициенту. Логика построения функции остается прежней: сначала проверяем корректность входных данных, а затем в цикле обновляем весовые коэффициенты. При обновлении весов, как и в приведенных формулах, вычислим сначала величину изменения синаптического коэффициента с учетом накопленного импульса, а затем его новое значение. Полученные значения сохраняем в соответствующие ячейки массивов весовых коэффициентов и моментов.

```
bool MomentumUpdate(double &m_cWeights[],
                    double &m_cGradients[],
                    double &m_cMomentum[],
                    double learningRate,
                    double beta)
  {
   if(m_cWeights.Size() > m_cGradients.Size() ||
      m_cWeights.Size() > m_cMomentum.Size()  ||
      m_cWeights.Size() <= 0)
      return false;

//---
   for(int i = 0; i < m_cWeights.Size(); i++)
     {
      m_cMomenum[i] = learningRate * m_cGradients[i] +
                      beta * m_cMomenum[i];
      m_cWeights[i] -= m_cMomenum[i];
     }
   return true;
  }
```

Метод адаптивного градиента (AdaGrad)

В обоих рассмотренных выше методах присутствует гиперпараметр скорости обучения. Важно понимать, что от выбора данного параметра во многом зависит и весь процесс обучения нейронной сети. Задание слишком большой скорости обучения может привести к постоянному росту ошибки вместо ее уменьшения. Использование малой скорости обучения приведет к

увеличению длительности процесса обучения и повысит вероятность застрять в локальном минимуме, даже при использовании метода накопления импульса.

Поэтому при валидации архитектуры нейронной сети очень много времени уделяется именно подбору правильного коэффициента скорости обучения. Но и при этом всегда сложно подобрать правильную скорость обучения. К тому же всегда хочется обучить нейронную сеть при минимальных затратах времени и ресурсов.

Существует практика постепенного понижения коэффициента в процессе обучения. Обучение сети начинается с относительно большим коэффициентом, который позволяет максимально быстро спуститься до некоторого уровня ошибки. После понижения коэффициента обучения проводится более тонкая подстройка весовых коэффициентов нейронной сети для понижения общей ошибки. Итераций с понижением коэффициента может быть несколько, но с каждым понижением весового коэффициента уменьшается эффективность данной итерации.

Здесь стоит обратить внимание еще и на тот момент, что мы используем одну скорость обучения для всех нейронных слоев и нейронов. Но не все признаки и нейроны вносят одинаковый вклад в финальный результат нейронной сети, поэтому наш коэффициент скорости обучения должен быть довольно универсальным.

Думаю, не стоит говорить, что универсальность враг лучшего: при создании любого универсального продукта или подбора значения (как в нашем случае) приходится идти на компромиссы, чтобы максимально удовлетворить все предъявляемые требования. А эти требования часто бывают противоречивыми.

Казалось бы, в таком случае следует предложить индивидуальные коэффициенты обучения для нейронов. Но решение этой задачи ручным трудом практически невыполнимо. В 2011 году был предложен метод адаптивного градиента AdaGrad. Предложенный метод является вариацией от рассмотренного выше градиентного спуска и не исключает использование коэффициента скорости обучения. При этом авторы метода предлагают накапливать сумму квадратов градиентов за все предшествующие итерации и при обновлении весовых коэффициентов делить обучающий коэффициент на квадратный корень из накопленной суммы квадратов градиентов.

$$G_{ilt}^{j} = G_{ilt-1}^{j} + (grad_{ilt}^{j})^{2}$$

$$w_{il}^{j} = w_{il}^{j} - \frac{\alpha}{\sqrt{G_{ilt}^{j} + \varepsilon}} * \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^{j}} grad_{il}^{j},$$

где:

- G_t, G_{t-1} — сумма квадратов градиентов на текущем и предыдущем шаге, соответственно;
- ε — некое малое положительное число, чтобы исключить деление на ноль.

Таким образом мы получаем индивидуальный для каждого нейрона и постоянно уменьшающийся коэффициент скорости обучения. Как всегда, за это нам приходится платить дополнительными ресурсами на вычисления и выделением дополнительного объема памяти для хранения сумм квадратов градиентов.

Функция реализации метода AdaGrad очень похожа на функцию обновления весовых коэффициентов методом накопленного импульса. В ней мы отказываемся от использования коэффициента затухания, но по-прежнему используем массив моментов. Только теперь мы будем

в него складывать накопленную сумму квадратов градиентов. Также изменения коснулись и вычисления нового значения весового коэффициента. Полный код функции приведен ниже.

```
bool AdaGradUpdate(double &m_cWeights[],
                   double &m_cGradients[],
                   double &m_cMomentum[],
                   double learningRate)
  {
   if(m_cWeights.Size() > m_cGradients.Size() ||
      m_cWeights.Sizel() > m_cMomentum.Size()  ||
      m_cWeights.Size() <= 0)
      return false;

//---
   for(int i = 0; i < m_cWeights.Size(); i++)
     {
      double G = m_cMomenum[i] + MathPow(m_cGradients[i], 2);
      m_cWeights[i] -= learningRate / (MathSqrt(G) + 1.0e-10) *
                 m_cGradients[i];
      m_cMomentum[i] = G;
     }
   return true;
  }
```

В приведенной выше формуле можно заметить и основную проблему данного метода. Мы постоянно накапливаем сумму квадратов градиентов. Как следствие, на достаточно длинной обучающей выборке наши скорости обучения быстро будут стремиться к нулю. Это приведет к параличу нейронной сети и невозможности дальнейшего обучения.

Выход из сложившейся ситуации был предложен в методе RMSProp.

Метод RMSProp

Метод обновления весовых коэффициентов RMSProp является логическим продолжением метода AdaGrad. В нем сохраняется идея автоматического регулирования скорости обучения в зависимости от частоты обновления и величины градиентов, поступающих на нейрон, но при этом решается основная проблема рассмотренного выше метода — паралич обучения на больших тренировочных выборках.

Как и AdaGrad, метод RMSProp эксплуатирует сумму квадратов градиентов, но в RMSProp применяется экспоненциально сглаженная средняя квадратов градиентов.

$$REMS(G_{il}^{j})_{t} = \gamma(grad_{il(t-k)}^{j})^{2} + (1-\gamma)REMS(G_{il}^{j})_{t-1}$$

$$w_{il}^{j} = w_{il}^{j} - \frac{\alpha}{\sqrt{REMS(G_{il}^{j}) + \varepsilon}} * \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^{j}} grad_{il}^{j},$$

где:

- ***REMS(G)$_t$, REMS(G)$_{t-1}$*** — экспоненциальная средняя квадратов градиентов на текущей и предыдущей итерации;

- γ — коэффициент экспоненциального сглаживания.

Использование экспоненциально сглаженной средней квадратов градиентов позволяет избежать снижения скорости обучения нейронов до нуля. При этом каждый весовой коэффициент получит индивидуальную скорость обучения, зависящую от поступающих градиентов. С ростом градиентов скорость обучения будет постепенно снижаться, а при снижении градиентов коэффициент скорости обучения будет нарастать. Это позволит в первом случае ограничить максимальную скорость обучения, а во втором — обновлять коэффициенты даже при малых градиентах ошибки.

Следует обратить внимание, что использование квадрата градиента позволяет работать данному методу и в том случае, когда на нейрон поступают разнонаправленные градиенты. Если из-за большого обучающего коэффициента в процессе обучения мы будем перескакивать через минимум и на следующей итерации двигаться в противоположном направлении, накопленный квадрат градиентов будет постепенно снижать коэффициент обучения, тем самым позволяя нам спуститься ближе к минимуму ошибки.

Реализация данного подхода практически полностью повторяет реализацию метода адаптивного градиента. Мы лишь заменим расчет суммы квадратов градиентов на их экспоненциальную среднюю. Для этого нам потребуется дополнительный параметр γ.

```
bool RMSPropUpdate(double &m_cWeights[],
                   double &m_cGradients[],
                   double &m_cMomentum[],
                   double learningRate,
                   double gamma)
  {
   if(m_cWeights.Size() > m_cGradients.Size() ||
      m_cWeights.Size() > m_cMomentum.Size()  ||
      m_cWeights.Size() <= 0)
      return false;

//---
   for(int i = 0; i < m_cWeights.Size(); i++)
     {
      double R = (1-gamma) * m_cMomenum[i] +
                 gamma * MathPow(m_cGradients[i], 2);
      m_cWeights[i] -= learningRate / (MathSqrt(R) + 1.0e-10) *
                 m_cGradients[i];
      m_cMomentum[i] = R;
     }
   return true;
  }
```

Метод Adadelta

В методах AdaGrad и RMSProp мы дали индивидуальный коэффициент обучения каждому нейрону, но по-прежнему оставили гиперпараметр скорости обучения в числителе формулы. Создатели метода Adadelta пошли немного дальше и предложили полностью отказаться от данного гиперпараметра. В математической формуле метода Adadelta его место заняла экспоненциальная средняя изменений весовых коэффициентов за предыдущие итерации.

$$REMS(\delta w_{il}^{j})_{t} = \gamma_{w}(\delta w_{il(t-k)}^{j})^{2} + (1-\gamma_{w})REMS(\delta w_{il}^{j})_{t-1}$$

$$REMS(G_{il}^{j})_{t} = \gamma_{G}(grad_{il(t-k)}^{j})^{2} + (1-\gamma_{G})REMS(G_{il}^{j})_{t-1}$$

$$w_{il}^{j} = w_{il}^{j} - \frac{REMS(\delta w_{il}^{j})}{REMS(G_{il}^{j}) + \varepsilon} * \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^{j}} grad_{il}^{j},$$

где:

- ***REMS(δw)$_t$*** , ***REMS(δw)$_{t-1}$*** — экспоненциальная средняя квадратов изменений весовых коэффициентов на текущей и предыдущей итерации.

В практике применения данного метода могут встречаться как одинаковые коэффициенты экспоненциального сглаживания квадратов дельт весовых коэффициентов и градиентов, так и индивидуальные. Решение принимает архитектор нейронной сети.

Ниже дается пример реализации метода средствами *MQL5*. Логика построения алгоритма полностью повторяет представленные выше функции. Изменения коснулись лишь вычислений, являющихся особенностями метода: отказ от коэффициента обучения и ввод дополнительного коэффициента усреднения, а вместе с ним и еще одного массива данных.

```
bool AdadeltaUpdate(double &m_cWeights[],
                    double &m_cGradients[],
                    double &m_cMomentumW[],
                    double &m_cMomentumG[],
                    double gammaW, double gammaG)
  {
   if(m_cWeights.Size() > m_cGradients.Size() ||
      m_cWeights.Size() > m_cMomentumW.Size() ||
      m_cWeights.Size() > m_cMomentumG.Size()  ||
      m_cWeights.Size() <= 0)
      return false;

//---
   for(int i = 0; i < m_cWeights.Size(); i++)
     {
      double W = (1-gammaW) * m_cMomenumW[i] +
                 gammaW * MathPow(m_cWeights[i], 2);
      double G = (1-gammaG) * m_cMomenumG[i] +
                 gammaG * MathPow(m_cGradients[i], 2);
      m_cWeights.At(i) -= MathSqrt(W) / (MathSqrt(G) + 1.0e-10) *
                 m_cGradients[i];
      m_cMomentumW[i] = W;
      m_cMomentumG[i] = G;
     }
   return true;
  }
```

Метод адаптивной оценки моментов

В 2014 году Diederik P. Kingma и Jimmy Lei Ba предложили метод адаптивной оценки моментов Adam. По словам авторов, метод объединяет преимущества методов AdaGrad и RMSProp и хорошо работает при онлайн-обучении. Данный метод показывает стабильно хорошие результаты на

разных выборках и в последнее время рекомендуется к применению по умолчанию в различных пакетах.

В основе метода лежит расчет экспоненциальной средней градиента *m* и экспоненциального среднего квадратов градиента *v*. Каждая экспоненциальная средняя имеет свой гиперпараметр *β*, определяющий период усреднения.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) grad_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) grad_t^2$$

Авторы предлагают использовать по умолчанию β_1 на уровне 0,9 и β_2 на уровне 0,999. При этом m_0 и v_0 принимают нулевые значения. С такими параметрами формулы, представленные выше, в начале обучения возвращают значения близкие к нулю. Как следствие, мы получаем низкую скорость обучения на начальном этапе. Для ускорения обучения авторы предложили скорректировать полученные моменты.

$$\hat{m} = \frac{m}{1 - \beta_1}$$

$$\hat{v} = \frac{v}{1 - \beta_2}$$

Обновление параметров осуществляется путем корректировки на отношение скорректированного момента градиента *m* к корню квадратному из скорректированного момента квадрата градиента *v*. Для исключения деления на ноль в знаменатель добавляют близкую к нулю константу *ε*. Полученное отношение корректируется на коэффициент обучения *α*, который в данном случае выступает верхней границей шага обучения. По умолчанию авторы предлагают использовать *α* на уровне 0,001.

$$w_t = w_{t-1} - \alpha \frac{\hat{m}}{\sqrt{\hat{v}} + \varepsilon}$$

Реализация метода *Adam* немного сложнее представленных выше, но в целом выдержана в той же логике. Изменения видны только в теле цикла обновления весовых коэффициентов.

```
bool AdamUpdate(double &m_cWeights[],
                double &m_cGradients[],
                double &m_cMomentumM[],
                double &m_cMomentumV[],
                double learningRate,
                double beta1, double beta2)
  {
//---
   if(m_cWeights.Size() > m_cGradients.Size() ||
      m_cWeights.Size() > m_cMomentumM.Size() ||
      m_cWeights.Size() > m_cMomentumV.Size()  ||
      m_cWeights.Size() <= 0)
      return false;
//---
   for(int i = 0; i < m_cWeights.Size(); i++)
     {
      double w = m_cWeights[i];
      double delta = m_cGradients[i];
      double M = beta1 * m_cMomenumM[i] + (1 - beta1) * delta;
      double V = beta2 * m_cMomenumV[i] + (1 - beta2) * MathPow(delta, 2);
      double m = M / (1 - beta1);
      double v = V / (1 - beta2);

      w -= learningRate * m / (MathSqrt(v) + 1.0e-10);
      m_cWeights[i] = w;
      m_cMomenumM[i] = M;
      m_cMomenumV[i] = V;
     }
//---
   return true;
  }
```

1.5 Приемы повышения сходимости нейронных сетей

В предыдущих главах книги мы уже познакомились с базовыми принципами построения и обучения нейронных сетей. Вместе с тем были обозначены и некоторые проблемы, возникающие в процессе обучения нейронных сетей. Мы увидели локальные минимумы, которые могут остановить обучение до достижения желаемых результатов. Упоминали о проблемах угасающих и взрывающихся градиентов. Кроме того, существует проблема совместной адаптации нейронов, переобучение и многие другие, о которых поговорим чуть позже.

Но на пути развития прогресса человечество стремится к совершенству орудий труда и технологий. Это относится и к алгоритмам обучения нейронных сетей. Давайте поговорим о методах, которые позволяют если не решить некоторые проблемы обучения нейронных сетей, то хотя бы минимизировать их влияние на конечный результат обучения.

1.5.1 Регуляризация

В погоне за минимальной погрешностью работы нейронной сети мы часто усложняем нашу модель. И каково же бывает разочарование, когда после длительной и кропотливой работы мы

получаем приемлемую ошибку на обучающей выборке, а при проверке модели на тестовой выборке ошибка просто «взлетает». Подобная ситуация встречается довольно часто. Это «переобучение модели».

Причины данного явления довольно банальны и связаны с неидеальностью, а точнее многогранностью, нашего мира. Дело в том, что и исходные данные, и эталонные результаты для обучающей и тестовой выборки получены не в лабораторных условиях, а взяты из реальной жизни. Следовательно, помимо анализируемых признаков они включают в себя целый ряд неучтенных факторов, которые мы на стадии проектирования в силу различных причин отнесли к так называемому шуму.

В процессе обучения мы ожидаем, что из переданного объема исходных данных модель выделит существенные признаки и построит зависимости между этими признаками и ожидаемым результатом. Но в результате чрезмерного усложнения модели она находит возможность найти несуществующие зависимости между случайными величинами — она «заучивает» обучающую выборку. Как результат, на обучающей выборке мы получаем ошибку близкую к нулю. При этом данные тестовой выборки содержат свои случайные шумовые отклонения, которые не вписываются в концепцию, выученную на обучающей выборке. Это вводит нашу модель в замешательство. В итоге мы получаем разительную разницу погрешности работы нейронной сети на обучающей и тестовой выборке.

Методы регуляризации, рассматриваемые в данном разделе, призваны исключить или минимизировать влияние случайного шума и выделить регулярные признаки в процессе обучения модели. В практике обучения нейронных сетей наиболее часто можно встретить использование двух методов: L1 и L2-регуляризации. Оба они построены на добавлении к функции потерь суммы норм весовых коэффициентов.

L1-регуляризация

L1-регуляризацию часто называют регрессией лассо или Манхэттенской. Суть данного метода заключается в добавлении к функции потерь суммы абсолютных значений весовых коэффициентов.

$$L_{L1}(Y, Y', W) = L(Y, Y') + \lambda \sum_{i=1}^{n} |w_i|,$$

где:

- $L_{L1}(Y, Y', W)$ — функция потерь с L1-регуляризацией;
- $L(Y, Y')$ — одна из рассмотренных ранее функций потерь;
- λ — коэффициент регуляризации (штраф);
- w_i — i-ый весовой коэффициент.

В процессе обучения нейронной сети мы будем минимизировать нашу функцию потерь. В данном случае минимизация функции потерь напрямую зависит от суммы абсолютных значений весовых коэффициентов. Таким образом, в обучение нашей модели мы вводим дополнительное ограничение на подбор весовых коэффициентов, максимально приближенных к нулю.

Частная производная такой функции потерь примет вид:

$$\frac{dL_{L1}(Y, Y', W)}{dw_i} = \frac{dL(Y, Y')}{dw_i} + \lambda sign(w_i)$$

Здесь мы не расписываем производную от самой функции потерь, чтобы выделить влияние непосредственно регуляризации.

Функция *sign(w_i)* возвращает знак при весовом коэффициенте, когда он отличен от нуля и 0, когда весовой коэффициент равен нулю. Так как λ — константа, а в процессе обновления весовых коэффициентов мы постоянно отнимаем значение производной, помноженной на обучающий коэффициент и градиент ошибки, то в процессе обучения нейронной сети модель обнулит признаки, не оказывающие прямое влияние на результат. Тем самым она полностью исключит влияние случайного шума на результат.

L1-регуляризация вводит штраф за большие веса, тем самым позволяя отобрать важные признаки и исключить влияние случайного шума на конечный результат.

L2-регуляризация

L2, или гребневая, регуляризация, как и L1-регуляризация, вводит в функцию потерь штраф за большие весовые коэффициенты. Но при этом используется L2-норма — сумма квадратов весовых коэффициентов. В результате функция потерь будет иметь следующий вид.

$$L_{L2}(Y, Y', W) = L(Y, Y') + \lambda \sum_{i=1}^{n} w_i^2$$

Аналогично L1-регуляризации в процесс обучения модели мы добавляем ограничение на использование весовых коэффициентов, максимально близких к нулю. Но давайте посмотрим на производную нашей функции потерь.

$$\frac{dL_{L1}(Y, Y', W)}{dw_i} = \frac{dL(Y, Y')}{dw_i} + 2\lambda w_i$$

В формуле производной L2-регуляризации штраф λ умножается на весовой коэффициент. Это означает, что в процессе обучения штраф не постоянен, а динамичен, он снижается пропорционально уменьшению весового коэффициента. При этом каждый весовой коэффициент получил индивидуальный штраф за размер весового коэффициента. Поэтому, в отличие от L1-регуляризации, в процессе обучения нейронной сети будут уменьшаться весовые коэффициенты признаков, не оказывающих прямое влияние на результат. Но они никогда не будут равны нулю, конечно если это позволяет предел точности вычислений.

L2-регуляризация вводит штраф за большие веса, тем самым усиливает влияние важных признаков и снижает, но не исключает, влияние случайного шума на конечный результат.

Elastic Net

Как уже было сказано выше, L1-регуляризация упрощает модель, обнуляя весовые коэффициенты при параметрах, не оказывающих прямого влияния на ожидаемый результат работы модели. Применение такого подхода оправдано, когда мы с большой долей уверенности понимаем наличие небольшого количества избыточных признаков, исключение которых может только улучшить работу модели.

Если же мы понимаем, что общий результат складывается из небольших вкладов всех используемых признаков и исключение каких-либо признаков ухудшит результат работы модели, то в таком варианте оправдано использование L2-регуляризации.

Но какой же из методов использовать, когда наша модель получает явно избыточное количество признаков. При этом мы не понимаем влияние отдельных признаков на результат. Возможно, исключение некоторых признаков могло бы упростить нашу модель и улучшить ее работу. В то же время, исключение других признаков отрицательно скажется на работе модели.

В такие моменты применяется эластичная регуляризация (Elastic Net regularization). Данная модель добавляет к функции потерь штрафы по первой и второй норме весовых коэффициентов и объединяет в себе преимущества L1 и L2-регуляризации.

$$L_{Elastic}(Y, Y', W) = L(Y, Y') + \lambda_1 \sum_{i=1}^{n} |w_i| + \lambda_2 \sum_{i=1}^{n} w_i^2$$

Обратите внимание, что в формуле Elastic Net L1 и L2-регуляризации получили собственный коэффициент регуляризации. Таким образом, изменяя коэффициенты регуляризации λ_1 и λ_2, можно управлять моделью регуляризации. При этом, приравняв их к нулю, получим оптимизации модели без регуляризации. При $\lambda_1>0$ и $\lambda_2=0$ имеем L1-регуляризацию в чистом виде, а при $\lambda_1=0$ и $\lambda_2>0$ получаем L2-регуляризацию.

1.5.2 Dropout

Продолжаем изучение методов повышения сходимости нейронных сетей. Рассмотрим технологию Dropout.

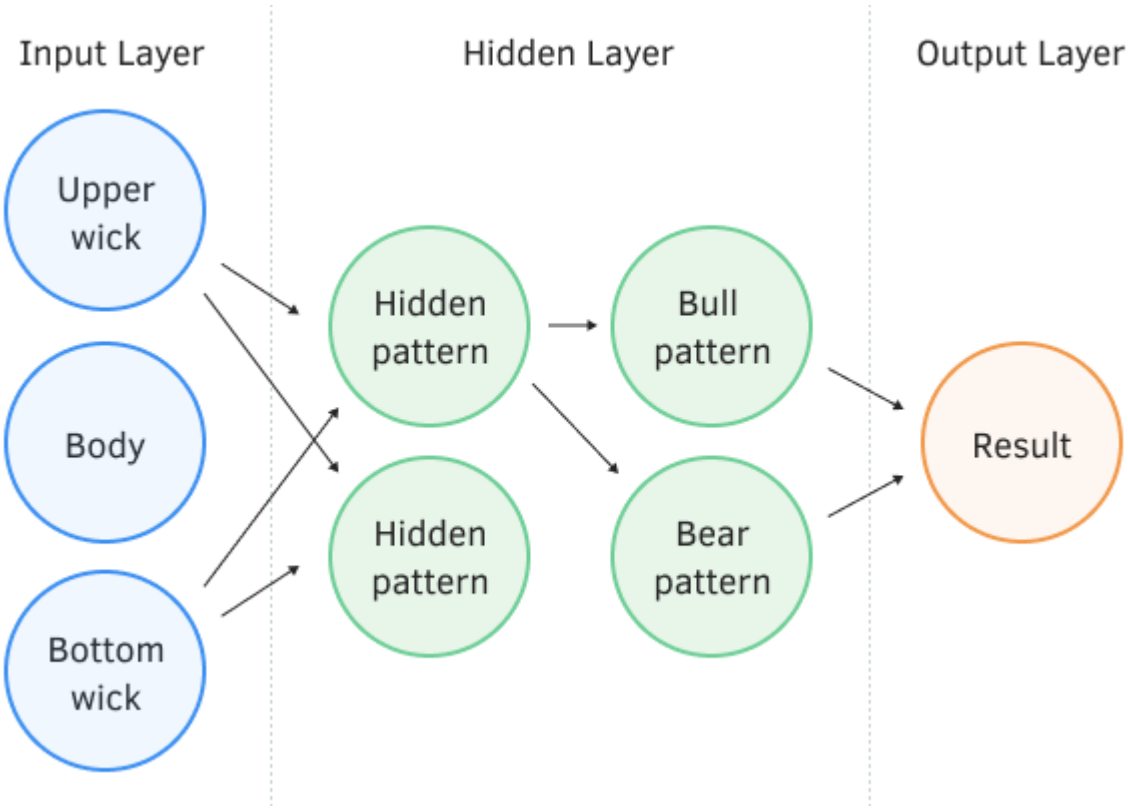
При обучении нейронной сети на вход каждого нейрона подается большое количество признаков, влияние каждого из которых сложно оценить. В результате ошибки одних нейронов сглаживаются правильными значениями других, а на выходе нейронной сети ошибки накапливаются. Обучение останавливается в некоем локальном минимуме с достаточно большой ошибкой, не удовлетворяющей нашим требованиям. Данный эффект был назван совместной адаптацией признаков, когда влияние каждого признака как-бы подстраивается под окружающую среду. Для нас было бы лучше получить обратный эффект, когда среда будет разложена по отдельным признакам, и оценивать отдельно влияние каждого из них.

Для борьбы со сложной совместной адаптацией признаков, в июле 2012 года группа ученных из университета Торонто в работе Improving neural networks by preventing co-adaptation of feature detectors предложила случайным образом исключать часть нейронов в процессе обучения. Снижение количества признаков при обучении повышает значимость каждого, а постоянное изменение количественного и качественного состава признаков снижает риск их совместной адаптации. Такой метод получил название Dropout.

Применение данного метода можно сравнить с деревьями решений, ведь согласитесь, исключая часть нейронов случайным образом, мы на каждой итерации обучения получаем новую нейронную сеть со своими весовыми коэффициентами. По правилам комбинаторики вариативность таких сетей довольно высока.

В то же время в процессе эксплуатации нейронной сети оцениваются все признаки и нейроны. Тем самым мы получаем максимально точную и независимую оценку текущего состояния изучаемой среды.

Авторы решения в своей статье указывают на возможность использования метода и для повышения качества предварительно обученных моделей.



Модель реализации Dropout для перцептрона с двумя скрытыми слоями

Описывая предложенное решение с точки зрения математики, можно сказать, что каждый отдельный нейрон выкидывается из процесса с некой заданной вероятностью *P*. Получается, нейрон будет участвовать в процессе обучения нейронной сети с вероятностью *q=1–P*.

Для определения списка исключаемых нейронов используется генератор псевдослучайных чисел с нормальным распределением. Такой подход позволяет достичь максимально возможного равномерного исключения нейронов. На практике мы будем генерировать вектор бинарных признаков размером, равным входной последовательности. Для используемых признаков в векторе пропишем 1, а для исключаемых элементов поставим 0.

Однако исключение анализируемых признаков несомненно ведет к снижению суммы на входе функции активация нейрона. Для компенсации этого эффекта умножим значение каждого признака на коэффициент *1/q*. Легко заметить, что данный коэффициент будет увеличивать значения, так как вероятность *q* всегда будет в диапазоне от 0 до 1.

$$D_i = \frac{1}{q} m_i x_i,$$

где:

- D_i — элементы вектора результатов Dropout;
- q — вероятность использования нейрона в процессе обучения;
- m_i — элемент вектора маскирования;
- x_i — элементы вектора входной последовательности.

При обратном проходе в процессе обучения градиент ошибки умножается на производную вышеприведенной функции. Как легко заметить, в случае Dropout обратный проход будет аналогичен прямому с использованием вектора маскирования из прямого прохода.

$$\frac{dD_i}{dx_i} = \frac{1}{q} m_i$$

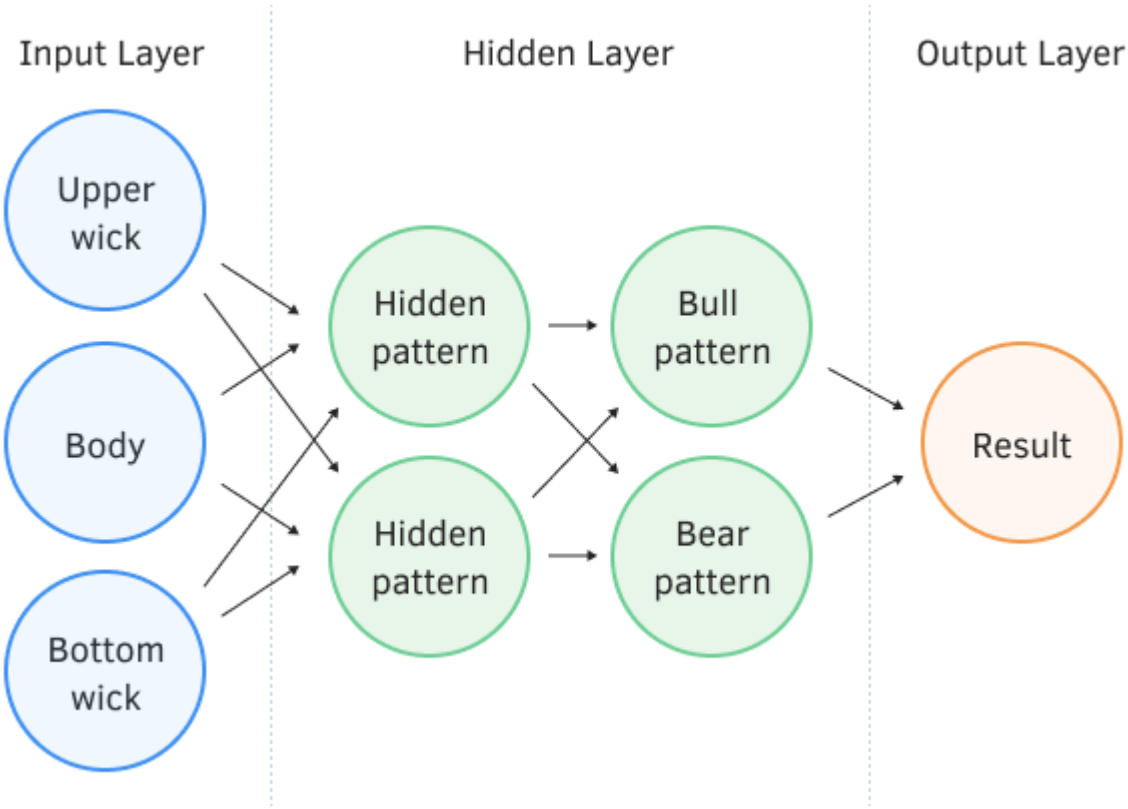
В процессе эксплуатации нейронной сети вектор маскирования заполняется единицами, что позволяет значениям беспрепятственно передаваться в обоих направлениях.

На практике коэффициент *1/q* постоянен на протяжении всего обучения, поэтому мы легко можем посчитать данный коэффициент один раз и записать его вместо единиц в тензор маскирования. Тем самым мы исключим операции пересчета коэффициента и умножения его на 1 маски в каждой итерации обучения.

1.5.3 Пакетная нормализация (Batch normalization)

В практике использования нейронных сетей используются различные подходы к нормализации данных. Все они направлены на удержание данных обучающей выборки и выходных данных скрытых слоев нейронной сети в заданном диапазоне и с определенными статистическими характеристиками выборки, такими как дисперсия и медиана. Почему же это так важно? Ведь мы помним, что в нейронах сети применяются линейные преобразования, которые в процессе обучения смещают выборку в сторону антиградиента.

Рассмотрим полносвязный перцептрон с двумя скрытыми слоями. При прямом проходе каждый слой генерирует некую совокупность данных, которые служат обучающей выборкой для последующего слоя. Результат работы выходного слоя сравнивается с эталонными данными, и на обратном проходе распространяется градиент ошибки от выходного слоя через скрытые слои к исходным данным.



Перцептрон с двумя скрытыми слоями

Получив на каждом нейроне свой градиент ошибки, мы обновляем весовые коэффициенты, подстраивая нашу нейронную сеть под обучающие выборки последнего прямого прохода. И здесь возникает конфликт: мы подстраиваем второй скрытый слой (на рисунке выше Bull и Bear pattern) под выборку данных на выходе первого скрытого слоя (на рисунке Hidden pattern), в то время как, изменив параметры первого скрытого слоя, мы уже изменили массив данных. То есть мы подстраиваем второй скрытый слой под уже несуществующую выборку данных.

Аналогичная ситуация и с выходным слоем, который подстраивается под уже измененный выход второго скрытого слоя. А если еще учесть искажение между первым и вторым скрытыми слоями, то масштабы ошибки увеличиваются. И чем глубже нейронная сеть, тем сильнее проявление этого эффекта. Это явление было названо внутренним ковариационным сдвигом.

В классических нейронных сетях указанная проблема частично решалась уменьшением коэффициента обучения. Небольшие изменения весовых коэффициентов не сильно изменяют распределение выборки на выходе нейронного слоя. Но такой подход не решает проблемы масштабирования с ростом количества слоев нейронной сети и снижает скорость обучения. Еще одна проблема низкого коэффициента обучения — застревание в локальных минимумах (этот вопрос мы уже обсуждали в разделе о методах оптимизации нейронных сетей).

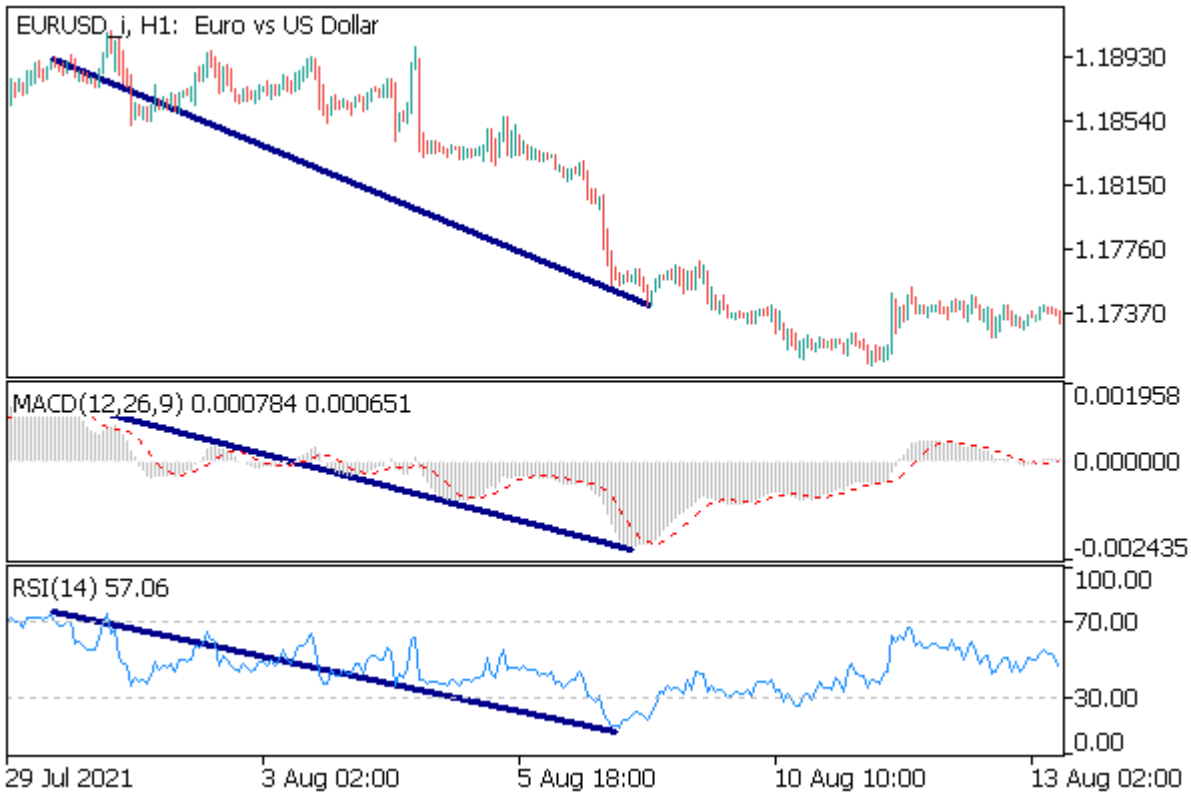
Здесь стоит еще упомянуть и о необходимости нормализации исходных данных. Довольно часто при решении различных задач на входной слой нейронной сети подаются разноплановые исходные данные, которые могут относиться к выборкам с разными распределениями. Некоторые из них могут иметь значения, по модулю значительно превышающие остальные. Такие значения будут оказывать большее влияние на конечный результат работы нейронной сети. В то же время реальное влияние описываемого фактора может быть значительно ниже, а абсолютные значения выборки обусловлены природой показателя.

На графике ниже для примера продемонстрировано отражение одного ценового движения двумя осцилляторами (MACD и RSI). При рассмотрении графиков индикаторов можно заметить корреляцию кривых. При этом числовые значения индикаторов отличаются в сотни тысяч раз. Это связано с тем, что значение индикатора RSI нормализованы по шкале от 0 до 100, а значения индикатора MACD зависят от точности указания цены на графике, поскольку MACD показывает расстояние между двумя скользящими средними.

При построении торговой стратегии мы можем пользоваться любым из этих индикаторов, а можем брать значение двух индикаторов и совершать торговые операции, только когда сигналы индикаторов совпадают. На практике такой подход позволяет отфильтровать часть ложных сигналов и тем самым снизить просадку торговой стратегии. Но прежде чем мы подадим столь различные сигналы на вход нейронной сети, желательно их привести в сопоставимый вид. Это и поможет нам сделать нормализация исходных данных.

Конечно, нормализацию исходных данных мы можем выполнить при подготовке обучающей и тестовой выборок вне нейронной сети. Но такой подход увеличивает подготовительную работу. Да и при практическом использовании такой нейронной сети нам придется все время подготавливать исходные данные по аналогичному алгоритму. Гораздо удобнее возложить эту работу на саму нейронную сеть.

В феврале 2015 года Сергей Иоффе (Sergey Ioffe) и Кристиан Сегеди (Christian Szegedy) в работе Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift предложили метод пакетной нормализации данных (Batch Normalization) для решения проблемы внутреннего ковариационного сдвига. Но этот алгоритм можно использовать и для нормализации исходных данных.



EURUSD H1

Суть метода заключалась в нормализации каждого отдельного нейрона на некоем временном интервале со смещением медианы выборки к нулю и приведением дисперсии выборки к единице.

Алгоритм проведения нормализации следующий. Вначале по выборке данных считается среднее значение.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i,$$

где:

- μ_B — среднее арифметическое признака по выборке;
- m — размер выборки (batch).

Затем считаем дисперсию исходной выборки.

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)$$

Нормализуем данные выборки, приведя выборку к нулевому среднему и единичной дисперсии.

$$\hat{x} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

Обратите внимание, что в знаменателе к дисперсии выборки прибавляется константа ε, небольшое положительное число с целью исключить деление на ноль.

Но как оказалось, такая нормализация может исказить влияние исходных данных. Поэтому авторы метода добавили еще один шаг — масштабирование и смещение. Были введены переменные γ и β, которые обучаются вместе с нейронной сетью методом градиентного спуска.

$$y_i = \gamma\hat{x}_i + \beta \equiv BN_{\gamma\beta}(x_i)$$

Применение данного метода позволяет на каждом шаге обучения получать выборку данных с одинаковым распределением, что на практике делает обучение нейронной сети более стабильным и позволяет увеличить коэффициент обучения. В целом это позволит повысить качество обучения при меньших затратах времени на обучение нейронной сети.

Но в то же время возрастают затраты на хранение дополнительных коэффициентов. Кроме того, для расчета скользящей средней и дисперсии требуется хранение в памяти исторических данных каждого нейрона на весь размер пакета. И тут можно посмотреть в сторону экспоненциальной средней: для расчета EMA достаточно предыдущего значения функции и текущего элемента последовательности.

Эксперименты, проведенные авторами метода, показывают, что применение метода Batch Normalization выступает и в роли регуляризатора. Это позволяет отказаться от использования других методов регуляризации, в частности от рассмотренного ранее Dropout. Более того, есть более поздние работы, в которых показано, что совместное использование Dropout и Batch Normalization отрицательно сказывается на результатах обучения нейронной сети.

В современных архитектурах нейронных сетей предложенный алгоритм нормализации можно встретить в различных вариациях. Авторы предлагают использовать Batch Normalization непосредственно перед нелинейностью (функцией активации). Как одну из вариаций данного алгоритма можно рассматривать метод Layer Normalization, предложенный Джимми Ли Ба (Jimmy Lei Ba), Джами Райн Кирос (Jamie Ryan Kiros) и Джефри Хинтон (Geoffrey E. Hinton) в июле 2016 года в работе Layer Normalization.

1.6 Искусственный интеллект в трейдинге

В предыдущих разделах были представлены базовые принципы и алгоритмы построения нейронных сетей. Однако нас в большей степени интересует возможность практического использования представленных технологий. И, конечно, я не первый, кто задумывался об этом.

Компьютерные технологии уже давно вошли и успешно применяются в трейдинге. Я бы даже сказал, что сейчас трудно представить трейдинг без использования компьютерных технологий. В первую очередь, благодаря интернету и компьютерам трейдеру сейчас не нужно лично присутствовать на торговой площадке. Программа торгового терминала легко устанавливается на любом компьютере, а в последнее время такие программы стали доступны и для мобильных устройств (смартфоны, планшетные ПК). Это позволяет трейдеру анализировать рынок и совершать торговые операции практически из любой точки нашей планеты.

Упомянутые выше торговые терминалы позволяют не только совершать сделки. Они оборудованы всем необходимым для проведения детального анализа рыночной ситуации в режиме реального времени. В них есть и инструменты для построения графических объектов на ценовых графиках инструментов, и целый ряд индикаторов, способных на лету обновляться и отображаться на графике в соответствии с текущей рыночной ситуацией.

Еще одно направление применения компьютерных технологий в тейдинге является алгоритмический трейдинг. Алготрейдинг подразумевает создание компьютерных программ-роботов, которые в рамках заданной торговой стратегии выполняют торговые операции без участия человека. У данного метода есть свои преимущества и недостатки по сравнению с ручной торговлей человеком.

Созданная программа может работать без устали 24 часа в сутки 7 дней подряд, что невозможно для человека. Соответственно, программа не пропустит какой-либо сигнал на вход в позицию или выход из нее.

Робот будет работать четко по заданному алгоритму. Человек же, оценивая рыночную ситуацию, помимо оцениваемых стратегией показателей, учтет свой личный предыдущий опыт и свои субъективные ощущения. Их влияние может быть различно.

В первую очередь, отклонение от стратегии торговли нарушает баланс прибыльных и убыточных операций и на длительном временном интервале скорее всего окажет отрицательное влияние на баланс торгового счета.

С другой стороны, бывает довольно сложно четко описать математическим языком все аспекты торговой стратегии. В таком случае существенную роль будет иметь личный опыт трейдера и его личное ощущение рынка. Программа не обладает такими свойствами, а заложенные программистом допуски могут быть неидеальны.

К положительным сторонам алготрейдинга можно добавить отсутствие психологического фактора у программ. В то время, как именно психологический барьер часто является причиной нарушения торговой стратегии трейдерами, особенно новичками.

С другой стороны, временные ряды изменчивы. Поэтому любая торговая стратегия имеет ограниченный срок службы. Как следствие, с течением времени возникает необходимость адаптации торговых систем к текущим рыночным ситуациям, а классический робот не может оценивать свою результативность и вносить изменения в торговый алгоритм или его параметры без помощи человека.

Что же мы ожидаем от применения искусственного интеллекта и нейронных сетей в частности?

При построении математической модели с использованием нейронной сети мы не прописываем весь торговый алгоритм, как в классическом алготрейдинге. Мы лишь даем обучающую выборку и предлагаем нейронной сети самой найти закономерности и корреляцию между исходными данными и конечным результатом. При этом мы ожидаем, что нейронная сеть помимо очевидных закономерностей уловит и те мелкие колебания, которые позволят увеличить результативность торговой системы.

Создавая обучающую выборку для нейронной сети, мы можем не ограничиваться исходными данными какой-либо одной стратегии. Исходных данных может быть гораздо больше, чем способен оценить человек. При этом финальная математическая модель будет давать сигналы, не вписываемые ни в одну из ожидаемых стратегий. В итоге мы ожидаем получить результативность выше, чем у роботов, построенных по классической схеме алготрейдинга.

И конечно, свойство нейронных сетей к обучению позволяет создать методы оценки результативности работы стратегии и своевременного запуска процесса обучения нейронной сети для адаптации к текущим рыночным условиям.

Таким образом, мы ожидаем снижение отрицательных сторон алготрейдинга при сохранении его положительных аспектов.

2. Возможности MetaTrader 5 для алготрейдинга

Для практической части книги был выбран терминал *MetaTrader 5*. Это современная, постоянно развивающаяся платформа, разработанная компанией *MetaQuotes Ltd*.

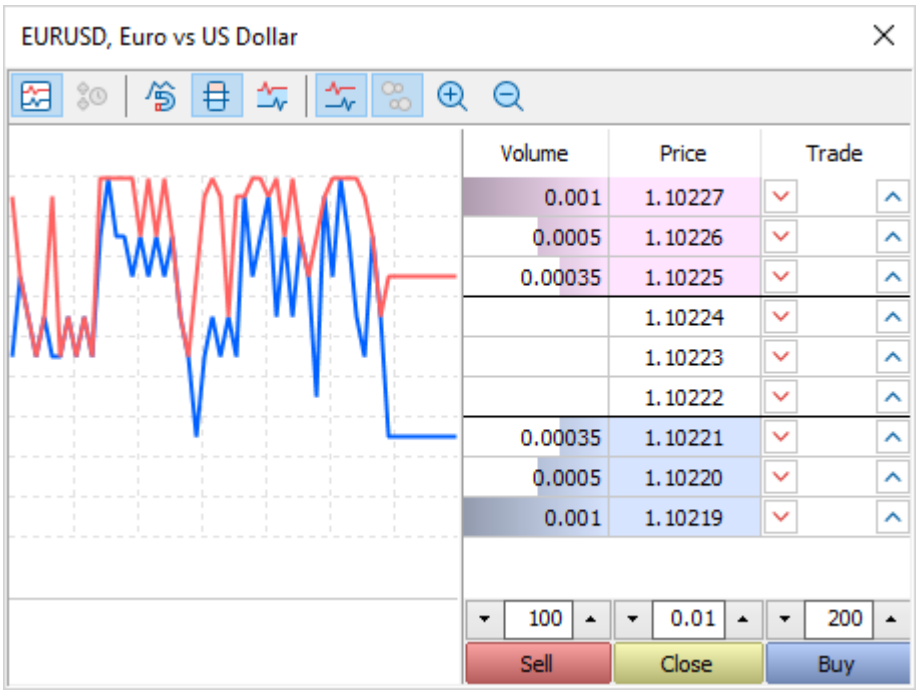
MetaTrader 5

Данная торговая платформа является мультирыночной. Она широко используется трейдерами всего мира для совершения торговых операций на рынке Форекс, фондовой бирже и рынке фьючерсов. *MetaTrader 5* является комплексной программой, которая позволяет совершать торговые операции и предоставляет широкие возможности для проведения детального технического и фундаментального анализа рыночной ситуации.

В платформе представлен расширенный стакан цен с тиковым графиком и лентой сделок. Данный инструмент позволяет не только проводить анализ текущего состояния, но и совершать торговые операции «в одно касание». При этом существует возможность указания уровней стоп-лосса и тейк-профита для выставляемых ордеров, что будет довольно полезно для осуществления скальпинговых торговых стратегий.

Предлагаемый стакан цен позволяет выставлять рыночные и отложенные ордера, а также модифицировать их. Более полная информация о возможностях данного инструмента доступна по ссылке https://www.metatrader5.com/en/terminal/help/trading/depth_of_market.

Кроме того, для проведения технического анализа имеются широкие возможности, позволяющие наносить различные графические объекты прямо на ценовой график. Спектр наносимых объектов довольно широк. Среди них есть как простые линии (вертикальные, горизонтальные и различные наклонные линии тренда), так и различные каналы, уровни Фибоначчи и более сложные фигуры. Существует возможность давать объектам различное цветовое и визуальное оформление, а также добавлять пользовательские названия и описание объектов.



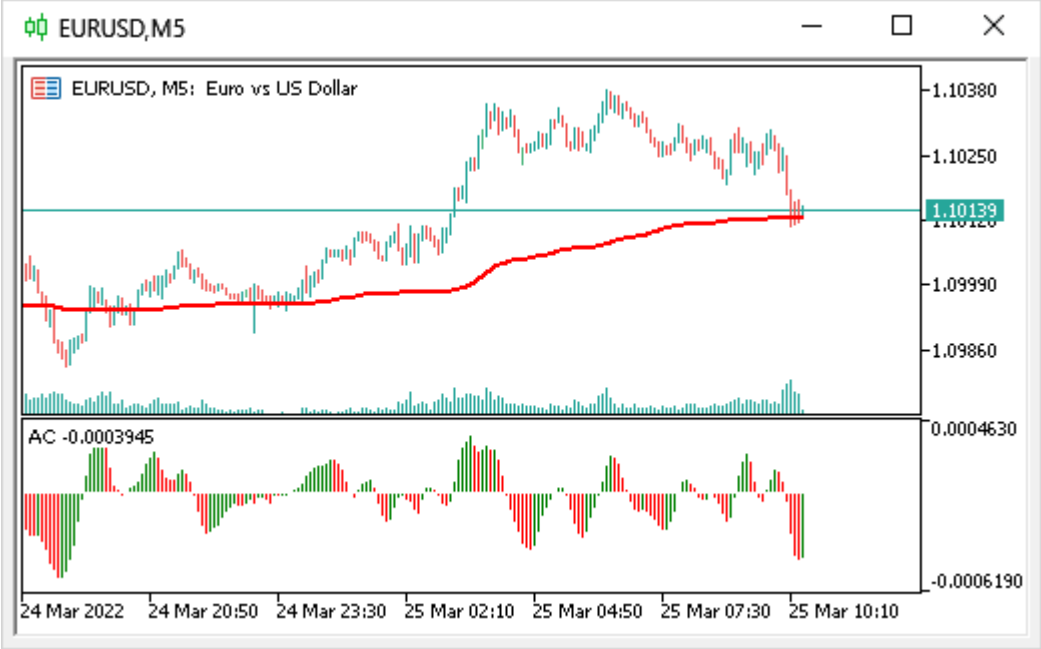
Стакан цен

Объекты технического анализа на графике

Вместе с платформой предлагается большой список осцилляторов, индикаторов объема и тренда, способных удовлетворить требованиям любого пользователя. В то же время, если вам будет недостаточно имеющегося спектра индикаторов, есть возможность создания пользовательского индикатора по вашей собственной формуле. Его вы можете создать самостоятельно или заказать у опытных программистов в сервисе «Фриланс» на сайте mql5.com.

Кроме того, в сервисе «Маркет» данного сайта вы можете купить или скачать бесплатно индикаторы различных сторонних разработчиков, список которых постоянно обновляется и

расширяется. Столь широкий спектр инструментов технического анализа вряд ли сможет предоставить какая-либо другая платформа.



Индикаторы на графике

Возможность анализа каждого инструмента на 21 таймфрейме, от 1 минуты до 1 месяца, позволяет получить полный и развернутый анализ.

Нанесенные на ценовой график индикаторы и графические объекты можно сохранить в шаблоны, которые в последующем можно будет повторно загрузить на график практически в два щелчка мышью.

Для сторонников фундаментального анализа платформа предлагает новостную ленту и календарь финансовых событий с возможностью вывода меток прошедших и предстоящих событий прямо на график инструмента. Это позволяет в будущем отслеживать изменения и анализировать торговые ситуации максимально быстро.

Функция торговли в один клик с графика торгового инструмента помогает трейдеру осуществлять операции максимально быстро и по лучшей цене.

В дополнение платформа дает практически неограниченные возможности для алгоритмической торговли — автоматизированной торговли с помощью роботов. Специалистами компании *MetaQuotes* специально для платформы разработана встроенная среда *MQL5 IDE* (Integrated Development Environment). Данная среда позволяет создавать собственные индикаторы и торговые стратегии, а также проверять и оптимизировать их во встроенном тестере стратегий на исторических данных, собранных из реальных тиков.

Платформа *MetaTrader 5* широко распространена и предлагается к использованию большинством брокеров во всем мире, что позволяет трейдеру выбирать торгового оператора на свое усмотрение.

Торговля в один клик

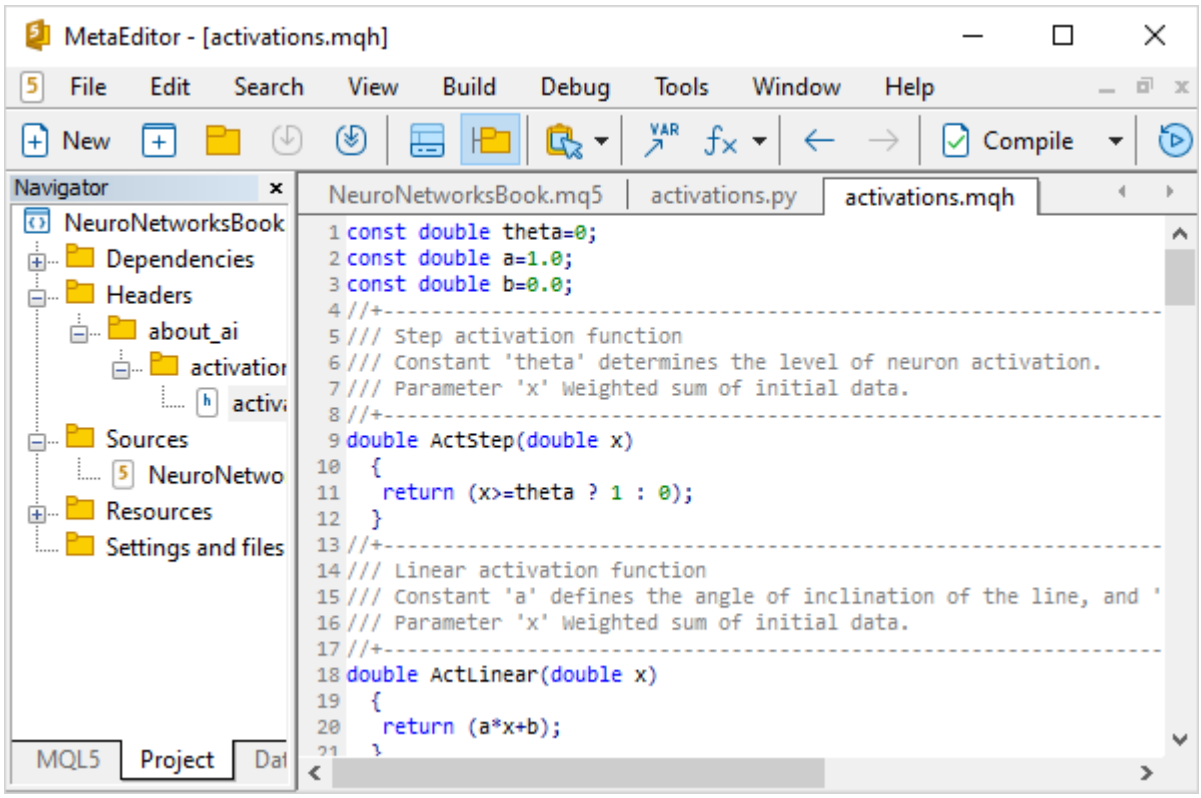
В тематике книги нас, конечно, будет интересовать возможность реализации технологий и алгоритмов нейронных сетей инструментами платформы *MetaTrader 5*. Давайте более детально рассмотрим предлагаемый инструмент.

2.1 Типы программ и особенности их построения

В комплект поставки платформы MetaTrader 5 входит современная среда разработки MetaEditor, которая позволяет создавать различные программы для алгоритмического трейдинга. Для написания программ используется специально разработанный язык программирования MetaQuotes Language 5 (MQL5). Синтаксис языка максимально приближен к C++ с возможностью написания программ в стиле объектно-ориентированного программирования. Это облегчает переход на использование MQL5 большой армии программистов.

Взаимодействие платформы MetaTrader 5 с программами организовано таким образом, что ценовые движения инструментов и изменения торгового счета отслеживается платформой. При наступлении предопределенных изменений платформа генерирует события на открытом в платформе графике инструмента. При наступлении события проверяются пользовательские программы, прикрепленные к графику. Это могут быть электронные советники, индикаторы и скрипты. Для каждого события и типа программ в платформе определены обработчики событий.

Обработчик событий — это специальная функция, определенная языком программирования *MQL5*. Такая функция имеет строго прописанные наименование, тип возвращаемого значения, список и тип параметров. По типу возвращаемого значения и по типам параметров обработчик событий клиентского терминала идентифицирует функции для обработки наступившего события. Если у функции указаны иные, не соответствующие предопределенным, параметры или указан иной тип возвращаемого значения, то такая функция не будет использоваться для обработки события.



Встроенная среда разработки MetaEditor

Каждый тип программ может обрабатывать только определенные события. Таким образом, если обработчик событий не будет соответствовать типу программы, такая функция не будет вызываться терминалом.

Для написания торговых советников язык MQL5 содержит ряд торговых функций и предопределенных обработчиков событий, что позволяет советникам выполнять заложенные в них торговые стратегии. Также предоставляется возможность написания собственных индикаторов технического анализа, скриптов, сервисов и библиотек включаемых функций.

Каждый вид программ предназначен для выполнения своих конкретных задач и имеет особенности построения.

Электронные советники (эксперты)

Наверное, во главе алгоритмической торговли стоят электронные советники (торговые роботы) — программы, способные самостоятельно анализировать рынок и совершать торговые операции на основе запрограммированной стратегии и ее правил совершения торговых операций.

Технически в MetaTrader 5 советник привязан к определенному графику, на котором запущен. При этом он обрабатывает предопределенные события только своего графика. Наступление каждого из событий запускает соответствующий функционал торговой стратегии. Такими событиями могут быть запуск и деинициализация программы, срабатывание таймера, приход нового тика, события графика и пользовательские события.

В то же время торговая стратегия советника может включать как анализ других таймфреймов текущего инструмента, так и анализ любого инструмента в терминале на любом таймфрейме. Это позволяет строить мультивалютные и мультитаймфреймовые стратегии.

Кроме того, у советников есть техническая возможность для получения данных с любого установленного в терминале технического индикатора. Это значительно расширяет возможности для построения различных торговых стратегий.

Каждое предопределенное событие вызывает соответствующую функцию советника, в которой прописан программный код обработки событий.

Сразу после запуска советника терминал генерирует событие ***Init***, которое запускает функцию ***OnInit***. В теле данной функции происходит инициализация глобальных переменных и объектов. При необходимости запускается таймер. Функция не имеет входных параметров, но по результатам отработки возвращает целочисленное значение кода возврата. Отличный от нуля код возврата сигнализирует о неудачной инициализации — в этом случае терминал генерирует событие завершения программы ***Deinit***.

```
//+------------------------------------------------------------------+
//| Expert initialization function                                   |
//+------------------------------------------------------------------+
int OnInit()
  {
//---

//--- create timer
   EventSetTimer(60);
//---
   return(INIT_SUCCEEDED);
  }
```

При завершении программы терминал *MetaTrader 5* генерирует событие ***Deinit,*** которое запускает выполнение функции ***OnDeinit***. Функция имеет один входной целочисленный параметр, в который передается код причины завершения работы программы. В теле функции при необходимости осуществляется удаление глобальных переменных, классов и графических объектов, сохранение данных в файловые ресурсы, закрытие таймера, запущенного при инициализации программы, и прочие операции, необходимые для нормального завершения работы программы и очистки следов ее работы в терминале.

```
//+------------------------------------------------------------------+
//| Expert deinitialization function                                 |
//+------------------------------------------------------------------+
void OnDeinit(const int reason)
  {
//---

//--- destroy timer
   EventKillTimer();
  }
```

При поступлении нового тика по инструменту, на графике которого запущен советник, генерируется событие ***NewTick***. При этом происходит запуск функции ***OnTick***. Это событие генерируется только для экспертов, поэтому в других программах функция ***OnTick*** не будет запускаться. Конечно, указанную функцию всегда можно вызвать принудительно из любого места программы, но это уже не будет отработка события ***NewTick***.

Функция ***OnTick*** не имеет входных параметров и не возвращает какой-либо код. Основное назначение функции — запустить на исполнение обработчик ценовых колебаний в советнике для оценки изменения рыночной ситуации и проверки правил заложенной стратегии на необходимость совершения каких-либо торговых операций. Иногда, по правилам торговой стратегии, эксперт должен выполнять операции не при каждом ценовом движении, а к примеру, при открытии новой свечи. В таких случаях в функцию ***OnTick*** добавляют проверку на наступление ожидаемого события.

```
//+------------------------------------------------------------------+
//| Expert tick function                                             |
//+------------------------------------------------------------------+
void OnTick()
  {
//---

  }
```

Если же алгоритм электронного советника не требует обработки каждого ценового движения, а строится на основе выполнения цикличных операций с неким временным интервалом, даже если в это время не происходят ценовые движения, в таких случаях очень полезным бывает использование таймера.

Для этого при инициализации программы в функции *OnInit* необходимо инициализировать таймер с помощью функции *EventSetTimer*. В параметрах функции указывается период задержки таймера в секундах. После этого терминал будет генерировать событие ***Timer*** для графика, и при этом будет запускаться на выполнение функция ***OnTimer*** советника.

```
//+------------------------------------------------------------------+
//| Timer function                                                   |
//+------------------------------------------------------------------+
void OnTimer()
  {
//---

  }
```

При использовании в коде программы таймера нужно обязательно его выгрузить из памяти терминала при завершении работы программы в функции *OnDeinit*. Для этого используется функция *EventKillTimer.* Данная функция не имеет параметров. Надо сказать, что платформой предусмотрено использование только одного таймера на графике.

В пределах одного советника при необходимости можно использовать и функцию OnTick, и функцию OnTimer.

Среди советников отдельно можно выделить подкласс полуавтоматических советников и торговых панелей. Такие программы не способны вести самостоятельную торговлю без участия человека. Напротив, программы данного вида осуществляют торговые операции по команде трейдера. Сами же программы призваны облегчить работу трейдера, взять на себя некие рутинные операции. Это может быть управление капиталом, установка уровней стоп-лосса и тейк-профита, сопровождение позиции и многое другое.

Для организации взаимодействия программы с пользователем в советниках обрабатываются события группы ***ChartEvent***. События данной группы запускают выполнение функции OnChartEvent, которая принимает от терминала четыре параметра:

- *id* — идентификатор события,
- *lparam* — параметр события типа *long*,
- *dparam* — параметр события типа *double*,
- *sparam* — параметр события типа *string*.

```
//+------------------------------------------------------------------+
//| ChartEvent function                                              |
//+------------------------------------------------------------------+
void OnChartEvent(const int id,
                  const long &lparam,
                  const double &dparam,
                  const string &sparam)
  {
//---

  }
```

Событие может генерироваться для советников и технических индикаторов. При этом для каждого типа события входные параметры функции имеют определенные значения, необходимые для обработки события.

Технические индикаторы

Другой тип возможных программ — пользовательские индикаторы. Это программы, которые выполняют аналитические функции для помощи трейдеру в проведении технического анализа рыночной ситуации. В процессе своей работы индикаторы обрабатывают каждое ценовое движение на графике своего торгового инструмента. Они могут выводить различные графические объекты, тем самым генерируя сигналы для последующего анализа трейдером.

Как и советники, пользовательские индикаторы в своих расчетах могут использовать данные других индикаторов, инструментов и таймфреймов. Но при этом индикаторы не могут совершать торговые операции. Таким образом, область использования индикаторов ограничивается рамками технического анализа.

Наравне с советниками, технические индикаторы имеют обработчики событий *Init*, *Timer* и *ChartEvent*. Построение функций для обработки указанных событий аналогично соответствующим функциям электронных советников, но вместо события *NewTick* для индикаторов генерируется событие *Calculate*. Данное событие обрабатывается функцией *OnCalculate*. В зависимости от области применения индикатора существует два вида функции *OnCalculate*:

- краткая

```
//+------------------------------------------------------------------+
//| Custom indicator iteration function                              |
//+------------------------------------------------------------------+
int OnCalculate (const int rates_total,
                 const int prev_calculated,
                 const int begin,
                 const double& price[]
  {
//---

//--- return value of prev_calculated for the next call
   return(rates_total);
  }
```

- полная

```
//+------------------------------------------------------------------+
//| Custom indicator iteration function                              |
//+------------------------------------------------------------------+
int OnCalculate(const int rates_total,
                const int prev_calculated,
                const datetime &time[],
                const double &open[],
                const double &high[],
                const double &low[],
                const double &close[],
                const long &tick_volume[],
                const long &volume[],
                const int &spread[])
  {
//---

//--- return value of prev_calculated for the next call
   return(rates_total);
  }
```

В пределах одного индикатора можно использовать только один из вариантов функции.

Оба варианта функции *OnCalculate* имеют параметры:

- *rates_total* — число элементов в таймсериях,
- *prev_calculated* — число пересчитанных элементов таймсерии при предыдущем запуске функции.

Использование параметра *prev_calculated* позволяет реализовать алгоритмы, при которых индикатор не пересчитывает ранее посчитанные исторические значения. Тем самым снижается количество итераций при обработке каждого нового ценового колебания.

Работа индикаторов в терминале *MetaTrader 5* организована следующим образом. В параметр *prev_calculated* передается значение, которое функция вернула при предыдущем запуске. Поэтому в общем случае в конце удачного завершения функции достаточно вернуть значение параметра *rates_total*. При возникновении ошибок в процессе работы функции, можно вернуть текущее значение *prev_calculated*. В таком варианте при следующем запуске функция начнет

пересчет с текущего места. Если вернуть 0, при следующем запуске индикатор будет пересчитан на всей истории, как при первом запуске.

При кратком определении функция имеет только один входной массив таймсерии (*price*) и параметр сдвига значимых значений относительно начала таймсерии (*begin*). В таком варианте расчет значений индикатора строится на основе данных одной таймсерии. Какая именно таймсерия будет использоваться, задается трейдером при запуске технического индикатора. Это может быть как любая из ценовых таймсерий, так и значения буфера другого индикатора.

При использовании полного варианта функции *OnCalculate*, в параметрах функция получает все ценовые таймсерии. В таком случае у пользователя нет возможности выбора таймсерии при запуске индикатора. Если необходимо использовать буфер данных другого индикатора, это нужно явно прописывать в программном коде индикатора.

В *MetaTrader 5* существует ограничение в возможности запуска только одного электронного советника для каждого графика. Если нужно запустить два и более советников в одном терминале, следует открыть отдельный график для каждого советника. Для индикаторов же нет такого ограничения — *MetaTrader 5* позволяет параллельно на одном графике торгового инструмента использовать встроенные и пользовательские индикаторы в различных вариациях. При этом индикатор может выводить данные как на сам ценовой график инструмента, так и в подокнах.

Скрипты

Советники и пользовательские индикаторы после их запуска остаются в памяти терминала до моментах принудительного их закрытия трейдером. В процессе наступления тех или иных событий терминал запускает определенный функционал советников и индикаторов. Для выполнения каких-либо разовых операций предусмотрены скрипты. Это отдельный тип программ, которые не обрабатывают никаких событий, кроме события своего запуска.

Сразу после запуска они выполняют заданный функционал и выгружаются из памяти терминала. Наравне с советниками скрипты способны совершать торговые операции, при этом на графике инструмента одновременно нельзя запустить более одного скрипта.

В теле скрипта существует только один обработчик событий ***OnStart***, который запускается сразу после запуска программы. Функция *OnStart* не получает параметров и не возвращает никаких кодов.

```
//+------------------------------------------------------------------+
//| Script program start function                                    |
//+------------------------------------------------------------------+
void OnStart()
  {
//---

  }
```

Отдельный тип программ — сервисы. В отличие от вышеописанных типов программ, сервис не требует привязки к определенному ценовому графику торгового инструмента. Как и скрипты, сервисы не обрабатывают никакие события, кроме собственного запуска, но при этом они сами способны генерировать пользовательские события и отправлять их на графики для дальнейшей обработки в советниках.

Кроме того, в среде разработки *MetaEditor* существует возможность создания библиотек и включаемых файлов. Эти файлы предназначены для хранения и распространения часто используемых блоков программ. Библиотеки являются скомпилированными файлами и предоставляют отдельные функции для экспорта в другие выполняемые программы. Сам же код выполняемых функций скрыт. Включаемые файлы, в отличие от библиотек, являются файлами с открытым кодом. В плане производительности предпочтительнее использовать включаемые файлы, но они не обеспечивают тайну кода при распространении.

Кроме обработчиков событий все программы могут содержать другие функции и классы, которые должны будут вызываться из функций обработчиков событий. Также они могут иметь внешние входные параметры, задаваемые пользователем при запуске программы.

Следует обратить внимание еще на один технический аспект. *MetaTrader 5* позиционируется как платформа с многопоточными вычислениями. При этом на каждый график торгового инструмента выделяется три потока: по одному для советника и скрипта, и один для индикаторов. Все индикаторы, загруженные на один график торгового инструмента, работают в одном потоке вместе с работой самого графика, поэтому не рекомендуется выполнять в индикаторах сложные расчеты.

Следовательно, для построения своей нейронной сети мы можем использовать советники или скрипты.

2.2 Инструменты статистического анализа и нечеткой логики

В среде разработки *MetaEditor* для обмена часто используемыми блоками кода выделен отдельный тип включаемых файлов с расширением *mqh*. Компания *MetaQuotes* в поставке *MetaTrader 5* предлагает огромную Стандартную библиотеку, в которой представлены классы и методы для реализации самых различных задач. В том числе, доступны классы для анализа данных с использованием математической статистики и нечеткой логики.

Библиотека математической статистики предлагает функционал для работы с основными статистическими распределениями. В ней имеется боле двадцати распределений, и для каждого представлено по пять функций:

1. Расчет плотности распределения.
2. Расчет вероятностей.
3. Расчет квантилей распределения.
4. Генерация случайных чисел с заданным распределением.
5. Расчет теоретических моментов распределения.

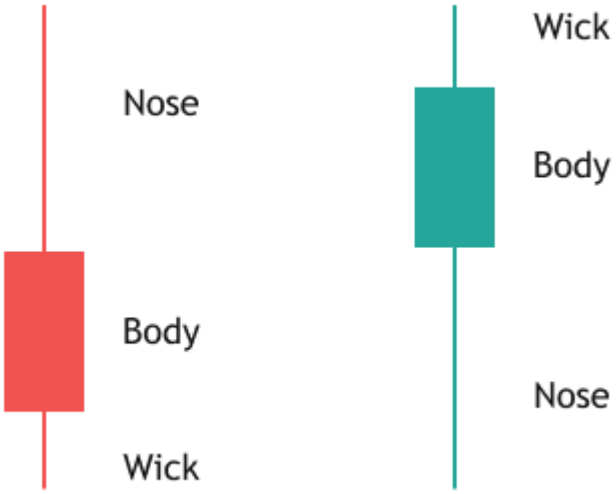
Библиотека также позволяет рассчитать статистические характеристики заданного массива данных. С помощью этой библиотеки можно легко провести статистический анализ выборки из исторических данных анализируемого инструмента. Также можно сравнить статистические показатели нескольких инструментов и посмотреть на динамику статистических показателей одного инструмента на исторических данных различных временных интервалов.

Кроме того, можно провести разносторонний и комплексный анализ, а его результаты заложить в основу построения своей торговой системы.

Прежде чем говорить о возможностях библиотеки нечеткой логики, хочу немного рассказать о самом направлении. Понятие нечеткой логики (fuzzy logic) было предложено американским

ученым Лотфи Заде (Lotfi Zadeh) в 1965 году. Данное нововведение позволяет добавить в расчеты некую долю субъективизма, присущего в реальной жизни. Ведь согласитесь, при описании тех или иных объектов и процессов мы часто используем нечеткие и приблизительные рассуждения.

Мы часто слышим фразу «слово вырвано из контекста». Это говорит о том, что трактовка слова и его использование в речи сильно зависит от контекста. Так же сложно описать отдельно взятую свечу на графике. Мы можем сказать какого она цвета, упомянуть о наличии теней. Но согласитесь, с таким описанием мы сможем разделить все свечи на два класса по их цвету. В теории нечеткой логики это будет два множества.



Свечи

Для дальнейшего описания нам понадобится вводить дополнительные понятия и измерения. Мы можем сравнить свечу с соседними свечами, вычислить некое среднее или взять некий эталон и сравнить с ним. При этом мы опять получим не совсем точное описание. Отклонение от нашего эталона или среднего может быть различным, как и влияние фактора может сильно меняться от размера этого отклонения. Применение нечеткой логики позволяет решить данную проблему благодаря введению «размытых» границ множеств.

В истории развития нечетких систем выделяют три этапа:

1. 1960–70-е годы — развитие теоретических аспектов нечеткой логики и нечетких множеств.
2. 1970–80-е годы — первые практические результаты в области управления нечетких систем.
3. С 1980-х годов и до нашего времени — создание различных пакетов программ для построения нечетких систем значительно расширяет область применения нечеткой логики.

Рассмотрим основные понятия теории нечетких множеств.

Прежде всего это ***нечеткое множество*** — множество значений, объединяемых некими правилами.

Математическое описание этих правил объединяется в ***функцию принадлежности***, которая является характеристикой нечеткого множества и обозначается $MF_C(x)$ — степень принадлежности значения x нечеткому множеству C.

Множество значений исходных данных, удовлетворяющих функции принадлежности, называется ***терм-множеством***.

Совокупность нечетких множеств и их правил объединяются в ***нечеткую модель (систему)***.

Результаты работы нечеткой модели определяются из совокупности нечетких множеств с использованием системы нечетких логических выводов. В библиотеке нечеткой логики MQL5 реализованы системы нечеткого логического вывода Мамдани и Сугено.

Для понимания различий между привычным математическим описанием и функцией принадлежности нечеткой логики рассмотрим пример описания свечи Доджи (свечи без тела). Такие свечи часто выступают предвестниками изменения тенденции, так как появляются в зоне равновесия спроса и предложения.

На практике редко встретишь свечу с нулевым размером тела, где цена открытия равна цене закрытия с математической точностью. Поэтому при определении свечи Доджи используются некие допуски. К примеру, примем допущение, что свечой Доджи будет любая свеча с телом не более 5 пунктов.

С таким допуском при использовании привычной логики свечи с телом в 1 пункт и 4 пункта будут отнесены к классу Доджи и для используемой стратегии будут иметь одинаковое значение. В то же время свеча с телом в 6 пунктов уже не попадет в категорию Доджи и будет проигнорирована стратегией. Почему же отклонение в 3 пункта в первом случае (4 − 1 = 3) не имеет значения, а меньшее отклонение в 2 пункта (6 − 4 = 2) во втором случае вносит кардинальную разность. Применение нечеткой логики позволяет сглаживать эти углы и учитывать отклонения в обоих случаях.

На рисунке ниже приведен график отнесения свечи к классу (множеству) Доджи в зависимости от длины тела свечи. Красной линией представлена классическая математическая логика с принятым выше допуском, а зеленая линия отражает правило нечеткой логики. Как видно из графика, использование правил нечеткой логики позволит нам принимать решения в зависимости от уровня силы поступающего сигнала. К примеру, если тело свечи больше и приближается к границам нечеткого множества, мы можем снижать риск на операцию или вовсе игнорировать такой сигнал.

Математически функцию принадлежности свечи к нечеткому множеству Доджи можно представить в виде:

$$MF_{Doji}(Body) = \begin{cases} \frac{a - Body}{a}, & ABS(Body) <= a \\ 0, & ABS(Body) > a \end{cases}$$

$$Body = Close - Open$$

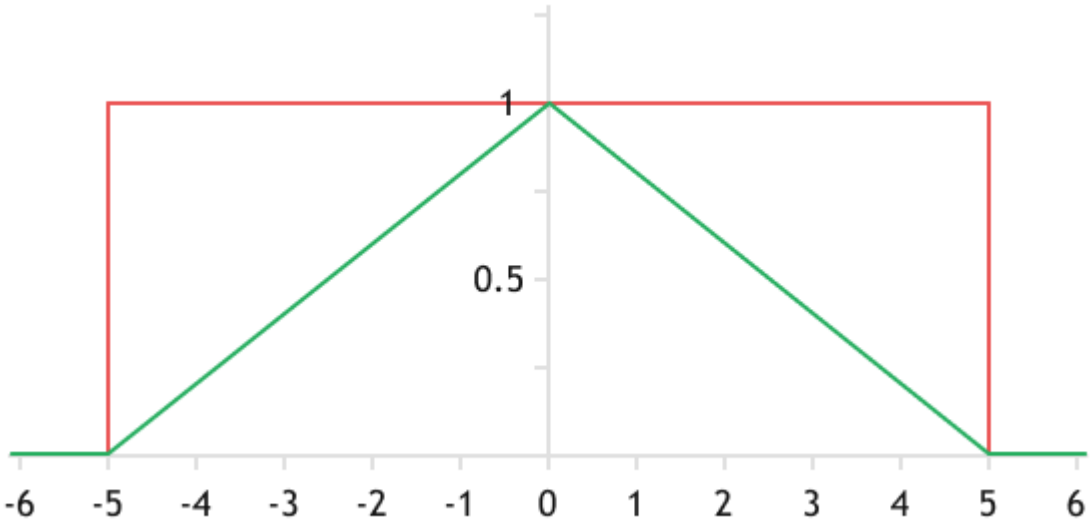


График отнесения свечи к классу Доджи (красный — математическая логика, зеленый — нечёткая логика)

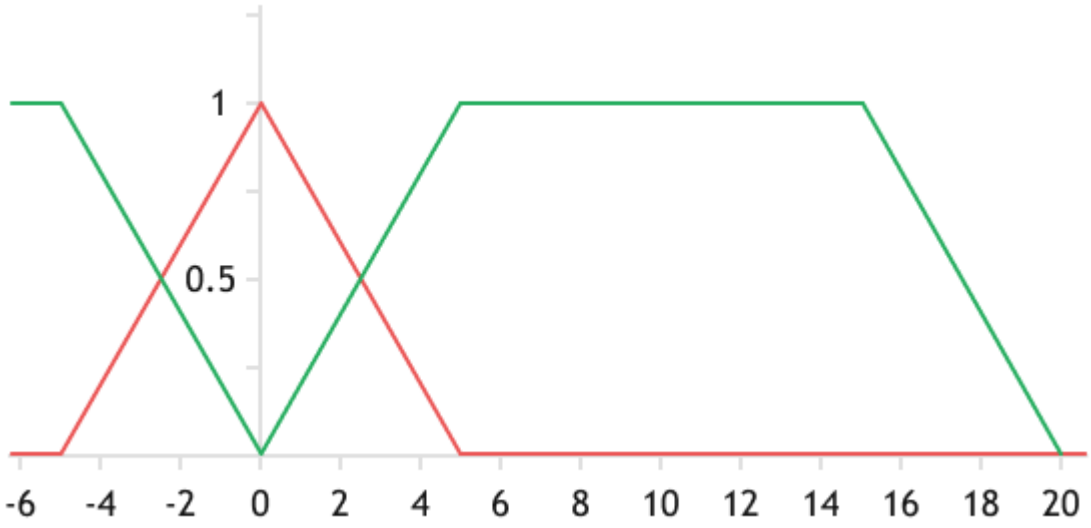
В данном случае мы получили частный случай симметричной треугольной функции активации. Для ее определения, по существу, нам хватило одного параметра a — границы диапазона в 5 пунктов. Центр распределения находится в точке 0. В общем же случае для определения треугольной функции принадлежности потребуется три параметра: нижняя граница, центр и верхняя граница нечеткого множества.

Существуют и другие функции принадлежности, но наибольшее распространение получили вышеупомянутая треугольная, трапецеидальная и гауссова функции принадлежности. При этом треугольная и трапецеидальная функции могут быть симметричными (когда левая и правая зоны размытости границ равны) и несимметричными.

График трапецеидальной функции отличается от графика треугольной функции наличием плато в верхней части. Для определения такой функции требуются четыре точки, указывающие верхние и нижние границы левой и правой зон размытости. В промежутке между зонами размытости функция принимает значение 1, а слева и справа от этих зон — 0. К примеру, введем правило для определения размера тела среднестатистической свечи с размером тела от 5 до 15 пунктов и зоной размытости границ в 5 пунктов. Математическая запись такого правила примет вид:

$$MF_{Midl}(Body) = \begin{cases} 1 - \frac{b - ABS(Body)}{b - a}, & a <= ABS(Body) <= b \quad (0 <= ABS(Body) <= 5) \\ 1, & b <= ABS(Body) <= c \quad (5 <= ABS(Body) <= 15) \\ 1 - \frac{ABS(Body) - c}{d - c}, & c <= ABS(Body) <= d \quad (15 <= ABS(Body) <= 20) \\ 0, & ABS(Body) \notin [a; d] \quad (ABS(Body) \notin [0; 20]) \end{cases}$$

Таким образом, мы определили уже второе правило для тела свечи. Совокупность правил для одной переменной принято показывать на одном графике. На графике ниже представлен красный треугольный терм для определения свечи Доджи и зеленый трапецеидальный терм средней свечи.



Совокупность терм-множеств Доджи (красный) и среднестатистическая свеча (зеленый).

Обратите внимание, что на графике нет резкого разделения между Доджи и средней свечой в точке 5 пунктов, как это было бы при пороговом делении на классы. Вместо этого мы имеем пересечение линий в районе 2,5 пункта. При этом функция принадлежности примет значение около 0,5. Это означает, что свеча с телом в 2,5 пункта в равной степени относится к нечетким множествам Доджи и средней свечи. В таком случае для определения управляющего воздействия следует обратить внимание на вторичные факторы.

Продолжая подобные итерации, мы можем описать правила для свечи с большим телом, а также добавить правила для теней свечи. После проведения работы по определению правил описания свечей и их составляющих мы с легкостью сможем описывать различные свечные паттерны. К примеру, мы с помощью инструментов нечеткой логики довольно просто можем описать пин-бар по определению: свеча с длинной одной тенью, небольшим телом и маленькой или отсутствующей второй тенью.

Обратите внимание, что использование правил нечеткой логики позволяет нам перейти от четких значений к неким абстрактным определениям и приближенным рассуждениям, присущим человеческой логике. Поэтому в теории нечеткой логики вводятся понятия лингвистической и нечеткой переменных.

Лингвистическая переменная имеет:

- *название*, в приведенных выше примерах это «Тело свечи»;
- *множество своих значений*, называемых базовым *терм-множеством*. В нашем случае это свечи «Доджи», «Средняя (обычная)» и «Большая»;
- *множество допустимых значений;*
- *синтаксическое правило*, описывающее термы с помощью слов естественного языка;
- *семантическое правило*, определяющее соответствие между значениями лингвистической переменной и нечетким множеством допустимых значений.

В общей практике размытость границ нечетких множеств позволяет учесть природный симбиоз влияния различных сил в зонах их пересечения. Также это позволяет учесть тот факт, что влияние силы ослабевает с ростом расстояния от источника воздействия.

Наличие нечетких правил — это важная, но не единственная часть в составлении модели. Процесс построения нечеткой модели можно разделить на три условных этапа:

1. Выбор исходных данных.
2. Определение базы знаний (набора правил).
3. Определение метода нечеткого логического вывода.

Вполне естественно, что весь процесс построения зависит от первого этапа: от определения множества исходных данных зависит как в целом возможность их классификации, так и количество возможных классов (термов). Следовательно, набор правил (а также их наполнение) для определения нечетких множеств тоже определяется исходя из множества допустимых значений и поставленной задачи. Следует отметить, что даже для одного набора исходных данных базовое терм-множество и набор правил могут варьироваться в зависимости от поставленной задачи.

Очень часто на параметры правил для определения нечетких множеств сильное влияние оказывают субъективные знания и опыт архитектора модели. Поэтому в последнее время широкое распространение получила практика гибридных моделей. В них подбор параметров правил осуществляется нейронной сетью в процессе ее обучения на обучающей выборке.

На основании созданной базы знаний в модели определяется система нечеткого логического вывода. Нечетким логическим выводом называется получение нечеткого множества, соответствующего текущим значениям входов, с использованием нечетких правил и нечетких операций.

Для нечетких множеств, как и для обычных, разработан ряд логических операций. Основными из них являются объединение (нечеткое «ИЛИ») и пересечение (нечеткое «И»). Создан общий подход к выполнению нечетких операций пересечения, объединения и дополнения.

Для построения процесса осуществления нечеткого логического вывода библиотека MQL5 предлагает реализацию двух основных методов: Мамдани (Mamdani) и Сугено (Sugeno).

При использовании метода Мамдани значение выходной переменной задается нечетким термом. Нечеткое правило данного метода можно описать в нижеследующем виде:

$$if\,(X_1\ is\ a_1) \wedge (X_2\ is\ a_2) \wedge .... \wedge (X_n\ is\ a_n)\ then\ (Y\ is\ d)(W),$$

где:

- ***X*** — вектор входных переменных,
- ***Y*** — выходная переменная,
- ***a*** — вектор исходных данных,
- ***d*** — значение выходной переменной,
- ***W*** — весовой коэффициент правила.

В методе Сугено, в отличие от Мамдани, значение выходной переменной задается не нечетким множеством, а линейной функцией от исходных данных. Правило данного метода имеет вид:

$$if\,(X_1\ is\ a_1) \wedge (X_2\ is\ a_2) \wedge .... \wedge (X_n\ is\ a_n)\ then\ (Y = b_0 + X_1 b_1 + X_2 b_2 + ... + X_n b_n)(W),$$

где ***b*** — вектор весовых коэффициентов при свободных членах функции выходного значения.

2.3 OpenCL как средство параллельных вычислений в MQL5

В процессе обучения и эксплуатации нейронных сетей осуществляется большое количество вычислений. Это довольно ресурсоемкий процесс. Для решения более сложных задач требуются и более сложные нейронные сети с большим количеством нейронов. С ростом количества нейронов в сети растет количество осуществляемых вычислений, следом растет количество затрачиваемых ресурсов и времени. И если человечество научилось создавать новые и более совершенные вычислительные машины, то управлять временем нам пока не под силу.

Вполне естественно, что очередной подъем развития нейронных сетей пришел с развитием вычислительных мощностей. Как правило, в нейронах осуществляются довольно простые операции, но в большом количестве. При этом в нейронной сети насчитывается большое количество однотипных нейронов. Это позволяет распараллеливать отдельные блоки вычислений на разных вычислительных ресурсах, а затем консолидировать полученные данные. В итоге время на выполнение операций значительно сокращается.

Развитие вычислительных технологий привело к появлению видеокарт (*GPU*) с большим количеством вычислительных ядер, способных осуществлять несложные математические операции. Следом появилась возможность перенести часть вычислений из *CPU* на *GPU*, что позволило распараллелить процесс вычислений как между микропроцессором и видеокартой, так и на уровне видеокарты между различными вычислительными единицами.

OpenCL (*Open Computing Language*) — это открытый бесплатный стандарт кроссплатформенного параллельного программирования различных ускорителей, используемых в суперкомпьютерах, облачных серверах, персональных компьютерах, мобильных устройствах и встроенных платформах.

OpenCL — это *C*-подобный язык программирования, позволяющий организовать вычисления на *GPU*. Поддержка этого языка в *MQL5* дает нам возможность организовать многопоточные вычисления наших нейронных сетей на *GPU* прямо из *MQL5*-программы.

Для понимания процесса организации вычислений на *GPU* необходимо сделать небольшой экскурс в архитектуру видеокарт и *OpenCL API*.

В терминологии *OpenCL* микропроцессор компьютера (*CPU*) является *Host*. Он управляет всеми процессами выполняемой программы. Все микропроцессоры с поддержкой технологии *OpenCL* в составе *CPU* и *GPU* являются устройствами *Device*. Каждое устройство имеет свой уникальный номер внутри платформы.

Один Device может иметь несколько вычислительных единиц *Computer Units*. Их число определяется количеством физических и виртуальных ядер микропроцессора. Для видеокарт это будут *SIMD*-ядра. Каждое *SIMD*-ядро содержит несколько потоковых процессоров *Stream Cores*. Каждый потоковый процессор имеет несколько вычислительных элементов *Processing Elements* (или *ALU*).

Конкретное количество *Computer Units*, *SIMD*-ядер, *Stream Cores* и *Processing Elements* зависит от архитектуры конкретного устройства.

Важная черта *GPU* — векторные вычисления. Каждый микропроцессор состоит из нескольких вычислительных модулей. Все они могут выполнять одну и ту же инструкцию. При этом у разных выполняемых потоков исходные данные могут быть разные. Это позволяет всем нитям *GPU-программы* осуществлять параллельную обработку данных. Таким образом, все вычислительные модули загружаются равномерно. Большим плюсом является то, что векторизация вычислений

осуществляется полностью автоматически на аппаратном уровне, без необходимости дополнительной обработки в коде программы.

OpenCL был разработан как кроссплатформенная среда для создания программ с применением технологии массового параллелизма вычислений. Создаваемые в ней приложения имеют свою иерархию и структуру. Организация работы программы, подготовка и консолидация данных осуществляются в *Host*-программе.

Host, как и классическое приложение, запускается и работает на *CPU*. Для организации многопоточности вычислений выделяется контекст — среда для выполнения специальных объектов программы *OpenCL*. Контекст объединяет набор *OpenCL*-устройств для запуска программы, сами программные объекты с их исходными кодами, а также набор объектов памяти, видимых хосту и *OpenCL*-устройствам. За создание контекста в *MQL5* отвечает функция *CLContextCreate*, в параметрах которой указывается устройство для выполнения программы. Функция возвращает хендл контекста.

```
int CLContextCreate(
  int          device       // порядковый номер OpenCL устройства или макрос
  );
```

Внутри контекста создается *OpenCL*-программа с помощью функции *CLProgramCreate*. В параметрах этой функции указывается хендл контекста и исходный код самой программы. В результате работы функции получаем хендл программы.

```
int CLProgramCreate(
  int          context,     // хендл на контекст OpenCL
  const string source       // исходный код
  );
```

OpenCL-программа делится на отдельные кернелы — исполняемые функции (ядра). Для объявления кернела предусмотрена функция *CLKernelCreate*. В параметрах функции указывается хендл ранее созданной программы и наименование кернела в ней. На выходе получаем хендл кернела.

```
int CLKernelCreate(
  int          program,     // хендл на объект OpenCL
  const string kernel_name  // имя кернела
  );
```

Обратите внимание, что в последующем при вызове кернела из основной программы на *GPU* осуществляется запуск нескольких его экземпляров в различных параллельных потоках. При этом определяется пространство индексов *NDRange*, которое может быть одномерным, двумерным и трехмерным. *NDRange* представляет собой массив целых чисел. Размер массива указывает на размерность пространства, а его элементы означают размерность в каждом из направлений.

Каждая копия кернела выполняется для каждого индекса из этого пространства и называется «*Work-Item*» (рабочей единицей). Каждой рабочей единице предоставляется глобальный индекс *ID*. Кроме того, каждая такая единица выполняет один и тот же код, но данные для выполнения могут быть различными.

Рабочие единицы организовываются в рабочие группы *Work-Groups*. Группы предоставляют собой более крупное разбиение в пространстве индексов. Каждой группе присваивается групповой индекс *ID*. Размерность групп соответствует размерности для адресации отдельных элементов. Каждому элементу сопоставляется уникальный в рамках группы локальный индекс *ID*.

Таким образом, рабочие единицы могут быть адресованы как по глобальному индексу *ID*, так и по комбинации группового и локального.

Такой подход позволяет снизить время на вычисления, но при этом затрудняет процесс обмена данными между различными экземплярами кернела. Это надо учитывать при создании программ.

Как уже сказано выше, программа OpenCL работает в своем контексте изолированно от вызывающей программы. Как следствие, она не имеет доступ к переменным и массивам основной программы. Поэтому перед запуском программы нужно все данные, необходимые для выполнения программы, из *ОЗУ* скопировать в память *GPU*. После завершения работы кернела обратно из памяти *GPU* нужно загрузить полученные результаты. В этом месте надо понимать, что затраты времени на копирование данных из *ОЗУ* в память *GPU* и обратно являются накладными затратами времени при выполнении расчетов на видеокартах. Поэтому для сокращения общего времени на выполнение всей программы перенос расчетов на GPU целесообразен только тогда, кода экономия времени от расчетов на *GPU* значительно выше расходов на перенос информации.

Внутри *GPU* также существует ранжирование памяти на глобальную, локальную и частную. Наиболее быстрый доступ получается из кернела к частной памяти, но и доступ к ней возможен только из текущего экземпляра кернела. Наибольшее время требуется для доступа к глобальной памяти, но ее объем — наибольший из трех упомянутых. К ней имеют доступ все запущенные экземпляры кернела. Глобальная память используется для обмена информацией с основной программой.

Глобальная память предоставляет доступ на чтение и запись элементам всех групп. Каждая рабочая единица *Work-Item* может писать и читать из любой части глобальной памяти.

Локальная память — локальная для группы область памяти, в ней можно создавать переменные, разделяемые всей группой. Она может быть реализована как отдельная память на *OpenCL*-устройстве или размечена как область в глобальной памяти.

Частная (private) память — область, видимая только рабочей единице *Work-Item*. Переменные, определенные в частной памяти одной рабочей единицы, не видны другим.

Иногда еще выделяют константную память. Это область глобальной памяти, которая остается постоянной во время исполнения кернела. Хост выделяет и инициализирует объекты памяти, расположенные в константной памяти.

Рассмотрим две реализации одной задачи, с использованием технологии *OpenCL* и без. Как вы увидите позже, одной из основных операций, которой мы будем пользоваться, является матричное умножение. Мы будем умножать матрицу на матрицу и матрицу на вектор. Для эксперимента я предлагаю сравнить умножение матрицы на вектор.

Для функции умножения матрицы на вектор в классической реализации воспользуемся системой из двух вложенных циклов. Ниже приведен пример такой реализации. В параметрах функции передается матрица и два вектора: матрица и один вектор исходных данных, а также один вектор для записи результатов. По правилам векторной математики умножение возможно, только когда количество столбцов матрицы равно размеру вектора. Результатом такой операции будет вектор размером, равным количеству строк матрицы.

```
//+------------------------------------------------------------------+
//|  Функция умножения векторов на CPU                               |
//+------------------------------------------------------------------+
bool MultCPU(matrix<TYPE> &source1, vector<TYPE> &source2, vector<TYPE> &result)
  {
//---
   ulong rows = source1.Rows();
   ulong cols = source1.Cols();
   if(cols != source2.Size())
     {
      PrintFormat("Size of vectors not equal: %d != %d", cols, source2.Size());
      return false;
     }
//---
   result = vector<TYPE>::Zeros(rows);
   for(ulong r = 0; r < rows; r++)
     {
      result[r] = 0;
      for(ulong c = 0; c < cols; c++)
         result[r] += source1[r, c] * source2[c];
     }
//---
   return true;
  }
//+------------------------------------------------------------------+
```

В теле функции мы вначале определим размеры матрицы, проверим на соответствие размеру вектора и зададим размер исходящего вектора, равный размеру строк матрицы исходных данных. После этого построим систему циклов. Внешний цикл будет отсчитывать строки матрицы и, соответственно, элементы вектора результатов. В теле указанного цикла мы сначала обнулим соответствующий элемент вектора результатов, затем создадим вложенный цикл и будем суммировать произведения соответствующих элементов текущей строки матрицы и вектора.

Функция не сложная. Однако слабое место такой реализации — это рост времени выполнения пропорционально росту количества элементов в матрице и векторе.

Решить это поможет использование *OpenCL*. Конечно, такая реализация будет немного сложнее. Вначале, напишем *OpenCL*-программу и сохраним в файле *mult_vect_ocl.cl*. Расширение **.cl* общепринято для *OpenCL*-программ, но необязательно для реализации в среде *MQL5* — в данном случае мы будем использовать файл только для хранения текста программы, а загружать программу будем в виде текста.

В коде программы включим поддержку типа *double*. Здесь стоит обратить внимание, что не все *GPU* поддерживают тип *double*. А даже если и поддерживают, то в большинстве случаев этот функционал отключен по умолчанию.

```
//--- by default some GPUs don't support doubles
//--- cl_khr_fp64 directive is used to enable work with doubles
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
```

Еще один момент. MetaTrader 5 позволяет использовать для вычислений OpenCL-устройства как с поддержкой типа double, так и без. Поэтому при использовании типа double в своей OpenCL-программе необходимо проверить совместимость используемого устройства. В противном случае

мы рискуем получить ошибку в процессе выполнения OpenCL-программы и прекращение ее работы.

В то же время MetaTrader 5 не ограничивает возможность использования всех доступных типов данных. Тут надо сказать, что язык OpenCL позволяет использовать различные скалярные типы данных:

- логический: *bool*,
- целочисленные: *char, uchar, short, ushort, int, uint, long, ulong*,
- с плавающей запятой: *float, double*.

Поддержка аналогичных типов данных есть и в MQL5. Здесь надо помнить, что каждый тип данных имеет свои ограничения возможного диапазона значений, а вместе с тем и используемый объем памяти для хранения данных. Поэтому если в вашей программе не требуется особая точность или диапазон возможных значений не слишком велик, то рекомендуется использовать менее ресурсоемкие типы данных. Это позволит более эффективно использовать память девайса и снизить затраты на копирование данных между основной памятью и памятью контекста *OpenCL*. В частности, тип *double* можно заменить на *float*. Он, конечно, дает меньшую точность, но при этом занимает в 2 раза меньше памяти и поддерживается всеми современными *OpenCL*-устройствами. Это позволяет сократить затраты на передачу данных между устройствами и расширить область использования программы.

Также OpenCL позволяет использовать векторные типы данных. Векторизация позволяет распараллелить вычисления не на программном уровне, а на уровне микропроцессора. Использование вектора из четырех элементов типа *double* позволяет полностью заполнить 256-битный вектор *SIMD*-команд и за один такт произвести вычисления над всем вектором. Таким образом, за один такт микропроцессора мы осуществляем операции над четырьмя элементами нашего массива данных.

OpenCL поддерживает векторные переменные всех целочисленных типов и с плавающей запятой из 2, 3, 4, 8 и 16 элементов. Но возможность их использования зависит от конкретного устройства. Поэтому перед выбором размерности вектора ознакомьтесь с техническими характеристиками вашего оборудования.

Но вернемся к нашей программе. Ниже приведен код кернела для вычисления векторного произведения строки матрицы на вектор. В параметрах кернела укажем указатели на массивы исходных данных и результатов. Также в параметрах передадим количество столбцов матрицы. Количество строк в матрице для операций в кернеле не имеет значение, так как в нем осуществляются операции только умножения одной строки на вектор. По существу, это умножение двух векторов.

Здесь надо обратить внимание, что вместо типа буферов данных мы указали абстрактный тип *TYPE*. Вы не сможете найти такой тип данных ни в одной документации. На самом деле, как было сказано выше, не все *OpenCL*-устройства поддерживают тип *double*. Чтобы сделать нашу программу более универсальной, было решено заменить тип данных макроподстановкой. Непосредственно тип данных мы будем указывать в основной программе. Такой подход позволяет буквально поменять тип данных в одном месте основной программы, после чего вся программа переключится на работу с указанным типом данных без риска потери информации из-за несоответствия типов.

В теле кернела функция *get_global_id* укажет глобальный индекс *ID* запущенной единицы *Work-Item*. Индекс в данном случае является аналогом счетчика итераций внешнего цикла в классической реализации. Он указывает порядковый номер строи матрицы и элемент вектора

результатов. Далее посчитаем сумму значений соответствующего потока аналогично расчету внутри вложенного цикла классической реализации. Но здесь есть нюанс. Для вычислений мы воспользуемся векторными операциями из четырех элементов. В свою очередь, для использования векторных операций нам необходимо подготовить данные. Из *Host* программы мы получаем массив из скалярных элементов, поэтому перенесем необходимые элементы в наши приватные векторные переменные с помощью функции *ToVect* (ее код рассмотрим ниже). Затем с помощью векторной операции *dot* получим значение умножения двух векторов из четырех элементов. То есть одной операцией мы получаем сумму произведений четырех пар значений. Полученное значение складываем в локальную переменную, в которой накапливается значение произведения строки матрицы на вектор.

После выхода из цикла сохраним накопленную сумму в соответствующий элемент вектора результатов.

```
//+------------------------------------------------------------------+
//| Mult of vectors                                                  |
//+------------------------------------------------------------------+
__kernel void MultVectors(__global TYPE *source1,
                          __global TYPE *source2,
                          __global TYPE *result,
                          int cols)
  {
   int shift = get_global_id(0) * cols;
   TYPE z = 0;
   for(int i = 0; i < cols; i+=4)
     {
      TYPE4 x = ToVect(source1, i, cols, shift);
      TYPE4 y = ToVect(source2, i, cols, 0);
      z += dot(x,y);
     }
   result[get_global_id(0)] = z;
  }
```

Как уже было сказано выше, для переноса данных из буфера скалярных значений в векторную переменную мы создали функцию *ToVect*. В параметрах функции передаем указатель на буфер данных, начальный элемент, общее количество элементов в векторе (строке матрицы) и смещение в буфере до начала вектора. Последний параметр, смещение, необходим для точного определения начала строки в буфере матрицы, так как *OpenCL* использует одномерные буфера данных.

Далее проверяем количество элементов до конца вектора, чтобы не выйти за его пределы, и переносим данные из буфера в частную векторную переменную. Недостающие элементы заполняем нулевыми значениями.

```
TYPE4 ToVect(__global TYPE *array, int start, int size, int shift)
  {
   TYPE4 result = (TYPE4)0;
   if(start < size)
     {
      switch(size - start)
        {
         case  1:
            result = (TYPE4)(array[shift+start], 0, 0, 0);
            break;
         case  2:
            result = (TYPE4)(array[shift+start], array[shift+start + 1], 0, 0);
            break;
         case  3:
            result = (TYPE4)(array[shift+start], array[shift+start + 1],
                             array[shift+start + 2], 0);
            break;
         default:
            result = (TYPE4)(array[shift+start], array[shift+start + 1],
                             array[shift+start + 2], array[shift+start + 3]);
            break;
        }
     }
   return result;
  }
```

В результате функция возвращает созданную векторную переменную с соответствующими значениями.

На этом завершается работа с *OpenCL*-программой. Далее мы продолжим работу на стороне основной программы (*Host*). Для работы с *OpenCL* в *MQL5* предлагается класс *COpenCL* в стандартной библиотеке *OpenCL.mqh*.

Вначале проведем подготовительную работу: подключим стандартную библиотеку, подгрузим ресурсом созданную ранее *OpenCL*-программу и объявим константы для индексов кернела, буферов и параметров программы. Также укажем используемый в программе тип данных. Я указал тип *float*, так как интегрированная *GPU* моего ноутбука не поддерживает тип *double*.

```
#include <OpenCL/OpenCL.mqh>
#resource "mult_vect_ocl.cl" as string OCLprogram
#define TYPE                          float
const string ExtType = StringFormat("#define TYPE %s\r\n"
                                    "#define TYPE4 %s4\r\n",
                                    typename(TYPE), typename(TYPE));
//+------------------------------------------------------------------+
//|  Defines                                                         |
//+------------------------------------------------------------------+
#define cl_program                    ExtType+OCLprogram
//---
#define k_kernel                      0
#define k_source1                     0
#define k_source2                     1
#define k_result                      2
#define k_cols                        3
```

Объявим экземпляр класса для работы с *OpenCL* и переменные для хранения хендлов буферов данных.

```
COpenCL*                  cOpenCL;
int                       buffer_Source1;
int                       buffer_Source2;
int                       buffer_Result;
```

На следующем этапе нам предстоит инициализировать экземпляр класса. Для этого создадим функцию *OpenCL_Init*. В параметрах функции передадим матрицу и вектор исходных данных.

В теле функции создадим экземпляр класса для работы с *OpenCL*, инициализируем программу, укажем количество кернелов и создадим указатели на кернел и буферы данных. Также скопируем исходные данные в память контекста. На каждом шаге проверяем результат выполнения операций, и в случае ошибки выходим из метода с результатом *false*. Код функции приведен ниже.

```
bool OpenCL_Init(matrix<TYPE> &source1, vector<TYPE> &source2)
  {
//--- создание OpenCL программы, кернела и буферов
   cOpenCL = new COpenCL();
   if(!cOpenCL.Initialize(cl_program, true))
      return false;
   if(!cOpenCL.SetKernelsCount(1))
      return false;
   if(!cOpenCL.KernelCreate(k_kernel, "MultVectors"))
      return false;
   buffer_Source1 = CLBufferCreate(cOpenCL.GetContext(),
                                   (uint)(sizeof(TYPE) * source1.Rows() *
                                          source1.Cols()), CL_MEM_READ_ONLY);
   buffer_Source2 = CLBufferCreate(cOpenCL.GetContext(),
                                  (uint)(sizeof(TYPE) * source2.Size()),
                                         CL_MEM_READ_ONLY);

   buffer_Result = CLBufferCreate(cOpenCL.GetContext(),
                                  (uint)(sizeof(TYPE) * source1.Rows()),
                                         CL_MEM_WRITE_ONLY);
   if(buffer_Result <= 0 || buffer_Source1 <= 0 || buffer_Source2 <= 0)
      return false;
   if(!CLBufferWrite(buffer_Source1,0,source1) ||
      !CLBufferWrite(buffer_Source2,0,source2))
     return false;
//---
   return true;
  }
```

Непосредственно вычисления будут осуществляться в кернеле. Для его запуска напишем функцию *MultOCL*. В параметрах функции передадим указатель на вектор результатов и размеры матрицы исходных данных.

Вначале передадим в кернел указатели на буферы данных и параметры размеров буферов. Данные операции выполняются методами *CLSetKernelArgMem* и *SetArgument*. Пространство индексов зададим в массиве *NDRange* по числу строк в матрице исходных данных. Запустим кернел на выполнение методом *Execute*. После выполнения всего массива экземпляров кернела считываем результаты вычислений из памяти устройства с помощью метода *CLBufferRead*.

```
bool MultOCL(int rows, int cols, vector<TYPE> &result)
  {
   result=vector<TYPE>::Zeros(rows);
//--- Set parameters
   if(!CLSetKernelArgMem(cOpenCL.GetKernel(k_kernel), k_source1, buffer_Source1))
      return false;
   if(!CLSetKernelArgMem(cOpenCL.GetKernel(k_kernel), k_source2, buffer_Source2))
      return false;
   if(!CLSetKernelArgMem(cOpenCL.GetKernel(k_kernel), k_result, buffer_Result))
      return false;
   if(!cOpenCL.SetArgument(k_kernel, k_cols, cols))
      return false;
//--- Run kernel
   int off_set[] = {0};
   int NDRange[] = {rows};
   if(!cOpenCL.Execute(k_kernel, 1, off_set, NDRange))
      return false;
//--- Get result
   uint data_read = CLBufferRead(buffer_Result, 0, result);
   if(data_read <= 0)
      return false;
//---
   return true;
  }
```

После отработки программы нужно освободить ресурсы и удалить экземпляр класса для работы с *OpenCL*. Этот функционал выполняется в функции *OpenCL_Deinit*. В ней сначала проверим действительность указателя на объект, вызовем метод *Shutdown* для высвобождения ресурсов и удалим объект.

```
void OpenCL_Deinit()
  {
   if(!cOpenCL)
      return;
//---
   cOpenCL.Shutdown();
   delete cOpenCL;
  }
```

Очевидно, что при использовании *OpenCL* объем работы программиста увеличивается. Что же мы получаем взамен?

Для оценки результативности создадим небольшой скрипт *opencl_test.mq5*. Во внешних параметрах скрипта укажем размер матрицы исходных данных.

```
//+------------------------------------------------------------------+
//| Внешние параметры                                                |
//+------------------------------------------------------------------+
sinput int Rows = 100000;    // Строк в матрице
sinput int Colms = 100;      // Столбцов в матрице
```

В теле скрипта объявим матрицу и векторы данных. Заполним исходные данные случайными значения.

```
//+------------------------------------------------------------------+
//| Программа скрипта                                                |
//+------------------------------------------------------------------+
void OnStart()
  {
   matrix<TYPE> X = matrix<TYPE>::Zeros(Rows, Colms);
   vector<TYPE> Y = vector<TYPE>::Zeros(Colms);
   vector<TYPE> Z;
   for(int i = 0; i < Colms; i++)
     {
      for(int r = 0; r < Rows; r++)
         X[r, i] = MathRand() / (TYPE)32767;
      Y[i] = MathRand() / (TYPE)32767;
     }
```

На следующем этапе инициализируем контекст *OpenCL* через вызов ранее рассмотренной функции *OpenCL_Init*. При этом не забываем проверить результат выполнения операций.

```
   if(!OpenCL_Init(X, Y))
      return;
```

Теперь мы можем замерить скорость выполнения операций в *OpenCL*-контексте. С помощью функции *GetTickCount* получим количество миллисекунд от старта системы до и после вычислений. Вычисления осуществим в ранее рассмотренной функции *MultOCL*.

```
   uint start = GetTickCount();
   if(!MultOCL(Rows, Colms, Z))
      Print("Error OCL function");
   uint end = GetTickCount();
   PrintFormat("%.1e OCL duration %0 000d msec, result %.5e",
                           Rows * Colms, end - start, Z.Sum());
   OpenCL_Deinit();
```

После выполнения операций очищаем контекст *OpenCL*.

Аналогичным способом замерим время выполнения операций классическим способом на CPU.

```
   start = GetTickCount();
   if(!MultCPU(X, Y, Z))
      Print("Error CPU function");
   end = GetTickCount();
   PrintFormat("%.1e CPU duration %0 000d msec, result %.5e",
                            Rows * Colms, end - start, Z.Sum());
```

В заключение скрипта еще раз добавим замер времени выполнения умножения матрицы на вектор с использованием матричных операций *MQL5*.

```
   start = GetTickCount();
   Z = X.MatMul(Y);
   end = GetTickCount();
   PrintFormat("%.1e matrix operation duration %0 000d msec, result %.5e",
                                        Rows * Colms, end - start, Z.Sum());
  }
```

Тестирование описанного скрипта проводилось на ноутбуке с CPU Intel Core i7-1165G7 и интегрированном графическом процессоре GPU Intel(R) Iris(R) Xe. По результатам замера времени выполнения вычислений победила технология OpenCL. Наиболее медленной оказалась классическая реализация с использованием системы вложенных циклов. При этом результат вычислений во всех трех вариантах был идентичен.

```
OpenCL: GPU device 'Intel(R) Iris(R) Xe Graphics' selected
1.0e+07 OCL duration  0 msec, result 2.60176e+06
1.0e+07 CPU duration  125 msec, result 2.60176e+06
1.0e+07 matrix operation duration  16 msec, result 2.60176e+06
```

Результаты сравнительного тестирования выполнения вычислений с использованием OpenCL и без

Здесь следует отметить, что при замере скорости вычислений с использованием технологии *OpenCL* мы исключили накладные расходы в виде инициализации и деинициализации *OpenCL*-контекста, программы, буферов и передачи данных. Поэтому при выполнении единичных операций ее использование будет не столь эффективным. Но как будет показано дальше, при обучении и эксплуатации нейронных сетей таких операций будет много, а процесс инициализации контекста и программы *OpenCL* будет осуществляться только один раз, при запуске программы. При этом мы будем стараться минимизировать процесс обмена данным между устройствами. Поэтому использование подобной технологии будет весьма полезно.

2.4 Интеграция с Python

Python — высокоуровневый язык программирования с динамической типизацией и автоматическим управлением памятью. Он ориентирован на повышение производительности разработчика и читаемости кода и относится к полностью объектно-ориентированным языкам программирования.

Python относится к интерпретируемым языкам программирования. Он часто используется для написания скриптов.

Синтаксис языка минималистичен, что повышает производительность программиста. В совокупности с интерпретируемостью языка это позволяет быстро кодировать и сразу тестировать отдельные блоки программы. Это позволяет сократить время на поиск и устранение ошибок при отладке программных продуктов, а в некоторых случаях дает возможность оценить эффективность решения на стадии проектирования без необходимости создания полного продукта.

В то же время интерпретируемые языки программирования заметно уступают компилируемым в скорости выполнения программ. Решение этой проблемы заложено в самой архитектуре *Python*. Он спроектирован так, что его маленькое ядро легко расширяется набором библиотек, в том числе написанных на компилируемых языках программирования.

Таким образом, *Python* можно сравнить с конструктором, в котором программы собираются из готовых блоков, которые уже прописаны и определены в библиотеках. Это объясняет большое количество стандартных библиотек. При этом в своей программе вы используете только тот функционал, который необходим для решения конкретной задачи.

Необычной особенностью языка является выделение блоков кода пробельными отступами. Если вы привыкли к четкому выделению блоков фигурными скобками в *C*-подобных языках, это может показаться неудобным. С другой стороны, структурирование программного кода делает его визуально понятным. Достаточно одного взгляда на код, чтобы определить наличие вложенных блоков и их границ.

В то же время это накладывает определенную ответственность на программиста. Если наличие указателей открытия и закрытия блоков проверяется компилятором и при их несоответствии выдается сообщение об ошибке, то при структурировании кода вся ответственность ложится на программиста. При этом неправильная структура может изменить ход выполнения программы.

Динамическая типизация позволяет программисту отвлечься от совместимости данных при их сохранении в переменные, потому что она автоматически получит тип принимаемых данных.

В стандартной библиотеке содержится большой набор полезных функций. Здесь есть средства и для работы с текстом, и для написания сетевых приложений.

Дополнительный функционал может быть реализован средствами обширного количества сторонних библиотек. Среди них можно найти инструменты и для математического моделирования, и для написание веб-приложений, и для разработки игр. Кроме того, существует возможность интеграции библиотек, написанных на *C* или *C++* и других языках.

Создан специализированный репозиторий программного обеспечения, написанного для *Python*, который предоставляет средства для простой установки пакетов в операционную систему. Среди библиотек репозитория можно найти функции на любой вкус, в том числе для валютных рынков и машинного обучения.

Благодаря всему выше перечисленному *Python* стал одним из самых популярных языков программирования. Он используется в анализе данных и машинном обучении. По состоянию на июль 2021 года *Python* занимает третье место в рейтинге популярности языков программирования TIOBE с показателем 10,95%.

Начиная с версии Build 2085, вышедшей в июне 2019 года, в *MetaTrader 5* добавлены *API* для запроса данных из терминала в *Python*-приложения. С тех пор этот функционал постоянно развивается. В настоящее время можно запускать скрипты *Python* прямо на графике терминала наравне с *MQL5*-приложениями.

Вместе с тем расширяется и функционал *Python*-приложений. Можно получать котировки из терминала для их анализа, а по результатам анализа открывать и закрывать позиции, выставлять отложенные ордера. Также есть возможность получить информацию о текущем состоянии счета, открытых позициях и ордерах. С полным списком функций можно ознакомиться на странице документации по интеграции с Python.

Для настройки подключения *Python* к *MetaTrader 5* вначале нужно скачать и установить последнюю версию интерпретатора со страницы https://www.python.org/downloads/windows/.

При установке Python обязательно отметьте чекбокс "Add Python 3.9 to PATH%" (версия может отличаться) для возможности запуска скриптов Python из командной строки.

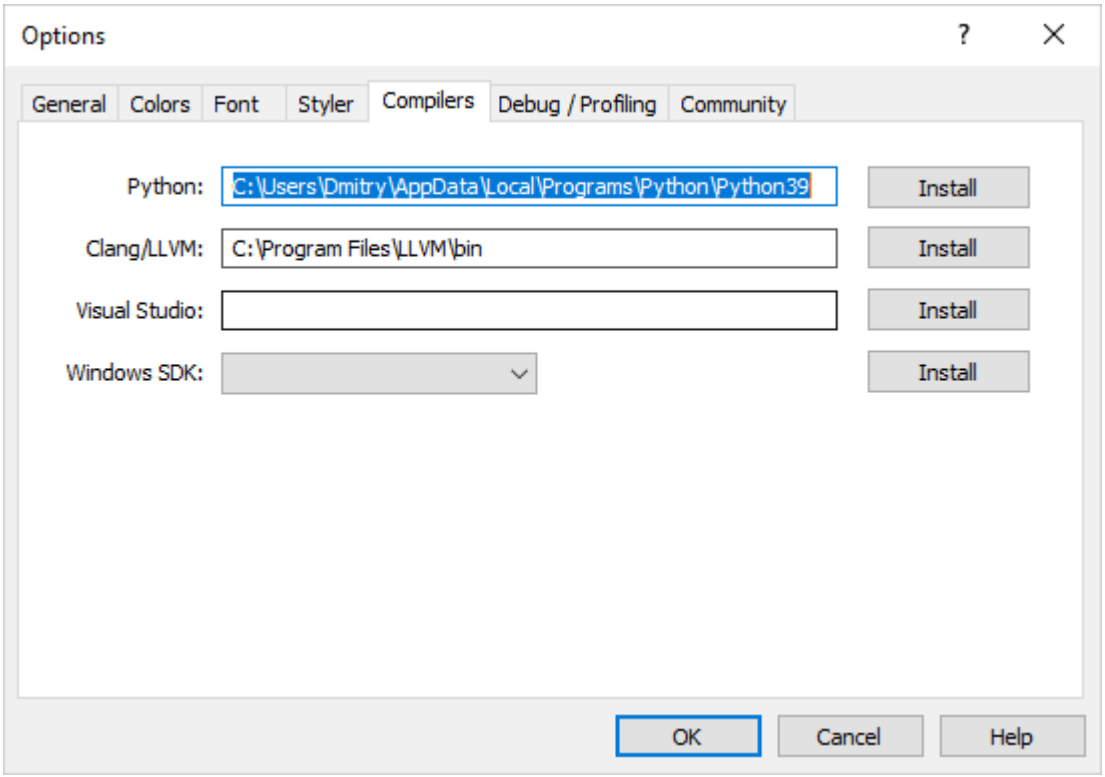
После этого запустите и обновите модуль *MetaTrader5*. В данном случае речь идет о *Python*-библиотеке, а не терминале. Для этого в командной строке введите нижеследующие команды.

```
pip install MetaTrader5
pip install --upgrade MetaTrader5
```

После этих итераций скрипты *Python* смогут получить доступ к операциям с терминалом *MetaTrader 5*.

В *MetaEditor* также реализована поддержка *Python*. В настройках редактора на вкладке «Компиляторы» достаточно указать расположение интерпретатора.

После этого можно создавать мультиязычные проекты в интегрированной среде *MetaEditor*. Такие проекты будут включать в себя программы, написанные на *mql* и *Python*. Аналогичным образом можно добавить поддержку языка *C/C++*.



Интеграция Python в MetaEditor

3. Построение первой модели нейронной сети средствами MQL5

Данную книгу нельзя рассматривать как учебник для изучения искусственного интеллекта и нейронных сетей. Мы не ставили перед собой задачу создания монументального труда, объединяющего все стороны данного учения. Напротив, в книге даны лишь базовые понятия, без погружения в математическое объяснения тех или иных моментов.

Это практический труд. Мы предлагаем вам посмотреть на возможные варианты решения практического кейса и сравнить эффективность различных алгоритмов в решении конкретной задачи. Думаю, эта книга будет полезна в изучении моментов практической реализации нейронных сетей различных алгоритмов, их обучения и практического использования.

И конечно, наш практический кейс будет напрямую связан с финансовыми рынками. Технологии искусственного интеллекта уже давно используются в финансовой сфере, но нельзя сказать, что

данная тема широко освещена. Во многом это связано с коммерческим использованием подобных продуктов.

В данной главе освещается тема алгоритмической торговли, где акцент делается на демонстрацию разнообразных методик решения задач, связанных с алготрейдингом, а также на анализе и сравнении производительности различных алгоритмических подходов. Основой для обсуждения является четкая постановка задачи, которая включает в себя определение ключевых целей и ограничений, характерных для рассматриваемого случая в контексте финансовых рынков.

Отдельный раздел посвящен выбору и анализу исходных данных, где основное внимание уделяется отбору подходящих финансовых показателей, анализу их корреляции и особенностям работы с временными рядами, что критически важно для успешного прогнозирования рыночных движений. Далее мы перейдем к обсуждению создания скелета будущей программы на языке MQL5, включая определение констант для гарантии стабильности и переносимости кода, а также механизм описания структуры создаваемой нейронной сети, что позволяет понять, как эффективно организовать работу с сложными сетевыми архитектурами.

В разделе создания базового класса нейронной сети читатели ознакомятся с концепциями прямого и обратного распространения ошибки в контексте программирования нейросетей. Особое внимание уделяется динамическому массиву для хранения нейронных слоев, что значительно упрощает управление сложными структурами данных в процессе разработки.

Описание интеграции нейронной сети в программу на Python дает понимание того, как можно совместить различные компоненты нейросети в единую систему, используя возможности этого популярного языка программирования. Разбор полносвязного нейронного слоя предоставляет информацию о его архитектуре и принципах создания, что помогает лучше понять структуру и функционирование таких слоев в нейронных сетях. Кроме того, обсуждается процесс создания класса функции активации и выбор подходящих функций активации для нейронов.

В разделе, посвященном параллельным вычислениям с использованием OpenCL, демонстрируется, как данная технология может быть применена для ускорения вычислительных процессов в нейросетях. Такой подход позволяет значительно повысить эффективность обработки данных за счет распределения задач между несколькими вычислительными устройствами.

3.1 Постановка задачи

Прежде чем приступить к практической реализации нашей первой нейронной сети, необходимо определиться с целью и средствами ее достижения. Разрабатывая архитектуру нейронной сети, мы должны четко понимать, какие данные должны быть на входе и выходе нейронной сети. Количество нейронов во входном слое исходных данных и их тип полностью зависят от набора исходных данных. Архитектура выходного слоя зависит от ожидаемого результата, от того, в каком виде будут представлены результаты работы создаваемой нейронной сети.

Давайте сформулируем задачу, которую мы бы хотели решить с помощью искусственного интеллекта. Мы работаем на финансовых рынках, и нам необходим инструмент для прогнозирования будущего движения анализируемого инструмента. Задача как бы не нова и у всех трейдеров на устах. Каждый пытается решить ее для себя по-своему.

Но в этой задаче нет конкретики. Что мы подразумевает под «будущим движением»? Будущее — это 5 минут, 1 час, 1 день, 1 месяц? А может что-то среднее? На какое минимальное движение цены мы будем реагировать. Какими метриками мы можем оценить точность работы нашей модели? Наша цель должна быть конкретной и измеримой.

Мы понимаем, что цена не движется по прямой. Всегда есть большие и малые ценовые колебания. Малые колебания, по существу, являются шумом. И чтобы выделить тренды и тенденции, способные дать нам потенциальную прибыль, мы должны отсеять этот шум. В поставке *MetaTrader 5* есть индикатор *ZigZag*. Это один из старейших индикаторов, используемых на финансовых рынках. Единственная задача данного индикатора — выделить наиболее значимые экстремумы на графике инструмента и тем самым показать тренды и тенденции, исключив незначительные шумовые колебания.

ZigZag на ценовом графике инструмента

Для настройки индикатора используются три параметра:

- ***Depth*** — задает количество свечей для поиска экстремумов. С ростом параметра индикатор выделяет наиболее значимые экстремумы.
- ***Deviation*** — определяет количество пунктов между двумя соседними экстремумами для отображения на графике.
- ***Backstep*** — указывает минимальном расстояние между соседними экстремумами в свечах.

В нашем случае мы можем использовать *ZigZag* для поиска экстремумов и указания целей для обучения нашей нейронной сети. Наложив индикатор на исторические данные обучающей выборки, мы сможем для каждой свечи и предшествующей ей свечной комбинации указать направление и расстояние до ближайшего экстремума. Тем самым мы будем учить модель определять потенциальные направление и силу будущего ценового движение.

Метрикой для оценки работы модели может быть как доля правильно указанных направлений прогнозных движений, так и точность определения силы такого движения.

Задача прогнозирования предстоящего направления движения представляется задачей бинарной классификации. Основываясь на данных индикатора *ZigZag*, в каждой конкретной точке у нас может быть либо движение вверх ценового графика (*Buy*), либо движение вниз ценового графика (*Sell*). Это не противоречит общепринятому делению трендовых движений на *BUY*, *SELL* и *FLAT*, так как флетовые движения представляют собой чередующиеся колебания *Buy* и *Sell* небольшой амплитуды.

В то же время средствами математической статистики мы не можем дать однозначный ответ о направлении предстоящего движения. Мы можем дать лишь вероятностный ответ, основанный на нашем прошлом опыте. Этот опыт мы будем «черпать» из обучающей выборки.

Что же касается прогнозирования силы движения, то здесь нам бы хотелось получить количественную оценку. Это поможет правильно оценить риск на сделку и определить точку для выставления тейк-профита с наибольшей вероятностью достижения.

Таким образом, задача прогнозирования будущего движения становится конкретной и измеримой. Сформулируем ее как прогнозирование наиболее вероятного направления предстоящего ценового движения и его ожидаемой силы.

3.2 Структура расположения файлов

В этой книге мы создадим много фалов для различных целей. Прежде чем продолжить работу, предлагаю определиться со структурой расположения файлов. Надо сказать, что работа в среде разработки *MQL5* накладывает некоторые ограничения в структуре файлов: для каждого типа программ предусмотрен свой каталог.

- *terminal_dir**MQL5**Experts* — каталог для хранения экспертов;
- *terminal_dir**MQL5**Indicators* — каталог индикаторов;
- *terminal_dir**MQL5**Scripts* — каталог скриптов;
- *terminal_dir**MQL5**Include* — каталог для хранения различных библиотек включаемых файлов;
- *terminal_dir**MQL5**Libraries* — каталог для хранения скомпилированных динамических библиотек.

В то же время среда разработки не ограничивает создание подкаталогов для структурирования файлов. В рамках данной книги мы будем создавать три типа файлов. Прежде всего, это наша библиотека включаемых файлов, в которой мы и будем проводить основную работу по организации работы моделей нейронных сетей. В рамках тестирования создаваемых моделей мы будем создавать и использовать различные скрипты. В конце книги мы создадим шаблон эксперта для демонстрации подходов использования моделей в практическом трейдинге.

Таким образом, мы будем создавать наши файлы в трех подкаталогах:

- *terminal_dir**MQL5**Experts* — каталог для хранения экспертов;
- *terminal_dir**MQL5**Scripts* — каталог скриптов;
- *terminal_dir**MQL5**Include* — каталог для хранения различных библиотек включаемых файлов.

Для того чтобы отделить наши файлы от всех остальных, в каждом из указанных каталогов мы создадим подкаталог *NeuroNetworksBook*. Более глубокое структурирование будем указывать для каждого создаваемого файла.

3.3 Выбор исходных данных

Мы определились с постановкой задачи, теперь перейдем к выбору исходных данных. Здесь есть свои особенности и подходы к решению данной задачи. На первый взгляд может показаться, что можно сгрузить всю доступную информацию в нейронную сеть и дать ей возможность самой выстроить правильные зависимости в процессе обучения. Такой подход непредсказуемо затянет процесс обучения без гарантии получения желаемого результата.

Первая проблема на нашем пути — это объем информации. Чтобы передать большой объем информации в нейронную сеть нам потребуется довольно большой входной слой нейронов с большим количеством связей. Следовательно, потребуется больше времени на обучение.

Далее мы столкнемся с проблемой несопоставимости данных. Выборки различных показателей будут иметь сильно отличающиеся статистические характеристики. К примеру, цена инструмента будет всегда положительная, а ее изменение может быть как положительное, так и отрицательное.

Некоторые индикаторы имеют нормализованные значения, а другие нет. Величины значений различных показателей и амплитуда их изменений могут отличаться в сотни раз. При этом их влияние на конечный результат может быть сопоставимым, или даже показатель с меньшими значениями может оказывать большее влияние.

Такая ситуация сильно усложнит процесс обучения, так как в массиве больших значений сложно будет выявить влияние малых величин.

Еще одна проблема заключается в использовании сильно коррелируемых показателей. Наличие корреляции между показателями может свидетельствовать либо о причинно-следственной связи между ними, либо о том, что оба показателя зависимы от одного общего фактора. Следовательно, использование коррелируемых показателей объединяет две вышеуказанные проблемы и их следствия. Использование нескольких показателей, зависящих от одного фактора, завышает его влияние на общий результат. При этом излишние нейронные связи усложняют модель и затягивают процесс обучения.

Отбор показателей

Проведем подготовительную работу с учетом вышеизложенных соображений. Конечно, мы не будем вручную выбирать и сопоставлять данные, ведь мы живем во времена компьютерных технологий. Для расчета коэффициента корреляции создадим небольшой скрипт в файле *initial_data.mq5*.

Напомню, что согласно описанной ранее структуре каталогов, все скрипты сохраняются в папке *terminal_dir\MQL5\Scripts*. Для скриптов нашей книги мы создадим подкаталог *NeuroNetworksBook*, а для всех скриптов из данной главы — подкаталог *initial_data*. Таким образом, полный путь создаваемого файла примет вид:

- *terminal_dir\MQL5\Scripts\NeuroNetworksBook\initial_data\initial_data.mq5*

Непосредственно расчет коэффициента корреляции будем осуществлять с использованием библиотеки математической статистики из поставки *MetaTrader 5*. В заголовке скрипта подключим необходимую библиотеку, во внешних параметрах укажем период для анализа.

```
#include <Math\Stat\Math.mqh>
//+------------------------------------------------------------------+
//| Параметры скрипта                                                |
//+------------------------------------------------------------------+
input datetime Start = D'2015.01.01 00:00:00';   // Period Start
input datetime End   = D'2020.12.31 23:59:00';   // Period End
```

При запуске скрипта *MetaTrader 5* генерирует событие ***Start***, которое обрабатывает функция ***OnStart*** в теле скрипта. В начале этой функции получим хендлы нескольких индикаторов для проведения дальнейшего анализа.

Следует обратить внимание, что в моем списке индикаторов первым идет *ZigZag*, который мы будем использовать для получения эталонных значений при обучении нейронной сети. Здесь же мы будем использовать его для проверки корреляции показаний индикаторов с эталонными значениями.

Параметры индикатора определяются пользователем на стадии постановки задачи. Я планирую производить обучение нейронной сети на данных таймфрейма M5, поэтому указал параметр *Depth* равным 48, что соответствует четырем часам. Таким образом, я ожидаю, что индикатор отобразит 4-часовые экстремумы.

Список и параметры индикаторов для анализа определяются архитектором нейронной сети исходя из своих соображений. Также возможен подбор параметров при оценке корреляции, что мы посмотрим немного позже. На данном этапе укажем индикаторы и их параметры из наших субъективных соображений.

```
void OnStart(void)
  {
   int h_ZZ=iCustom(_Symbol,PERIOD_M5,"Examples\\ZigZag.ex5",48,1,47);
   int h_CCI=iCCI(_Symbol,PERIOD_M5,12,PRICE_TYPICAL);
   int h_RSI=iRSI(_Symbol,PERIOD_M5,12,PRICE_TYPICAL);
   int h_Stoh=iStochastic(_Symbol,PERIOD_M5,12,8,3,MODE_LWMA,STO_LOWHIGH);
   int h_MACD=iMACD(_Symbol,PERIOD_M5,12,48,12,PRICE_TYPICAL);
   int h_ATR=iATR(_Symbol,PERIOD_M5,12);
   int h_BB=iBands(_Symbol,PERIOD_M5,48,0,3,PRICE_TYPICAL);
   int h_SAR=iSAR(_Symbol,PERIOD_M5,0.02,0.2);
   int h_MFI=iMFI(_Symbol,PERIOD_M5,12,VOLUME_TICK);
```

Следующим этапом загрузим исторические данные котировок и индикаторов. Для получения исторических данных создадим ряд массивов, названия которых будут созвучны с названиями индикаторов и котировок. Это позволит нам не путаться в работе с ними.

Получение котировок в *MQL5* осуществляется функциями ***CopyOpen, CopyHigh, CopyLow*** и ***CopyClose***. Функции созданы по одному шаблону, а из названия функции понятно, какие котировки она возвращает. За получение данных с индикаторных буферов отвечает функция ***CopyBuffer***. Вызов функции похож на функции получения котировок, только название инструмента и таймфрейм заменяются хендлом индикатора и номером буфера. Напомню, что хендлы индикаторов мы получили немного выше.

```
double close[], open[],high[],low[];
if(CopyClose(_Symbol,PERIOD_M5,Start,End,close)<=0 ||
   CopyOpen(_Symbol,PERIOD_M5,Start,End,open)<=0   ||
   CopyHigh(_Symbol,PERIOD_M5,Start,End,high)<=0   ||
   CopyLow(_Symbol,PERIOD_M5,Start,End,low)<=0)
   return;
```

Все функции записывают данные в указанный массив и возвращают количество скопированных значений. Поэтому при вызове проверяем наличие загруженных данных, а при их отсутствии выходим из скрипта. В таком случае дадим терминалу немного времени на загрузку котировок с сервера и пересчет значений индикаторов. После этого повторно запустим скрипт.

```
double zz[], cci[], macd_main[], macd_signal[],rsi[],atr[], bands_medium[];
double bands_up[], bands_low[], sar[],stoch[],ssig[],mfi[];
datetime end_zz=End+PeriodSeconds(PERIOD_M5)*(12*24*5);
if(CopyBuffer(h_ZZ,0,Start,end_zz,zz)<=0 ||
   CopyBuffer(h_CCI,0,Start,End,cci)<=0  ||
   CopyBuffer(h_RSI,0,Start,End,rsi)<=0  ||
   CopyBuffer(h_MACD,MAIN_LINE,Start,End,macd_main)<=0     ||
   CopyBuffer(h_MACD,SIGNAL_LINE,Start,End,macd_signal)<=0 ||
   CopyBuffer(h_ATR,0,Start,End,atr)<=0  ||
   CopyBuffer(h_BB,BASE_LINE,Start,End,bands_medium)<=0 ||
   CopyBuffer(h_BB,UPPER_BAND,Start,End,bands_up)<=0    ||
   CopyBuffer(h_BB,LOWER_BAND,Start,End,bands_low)<=0   ||
   CopyBuffer(h_SAR,0,Start,End,sar)<=0  ||
   CopyBuffer(h_Stoh,MAIN_LINE,Start,End,stoch)<=0  ||
   CopyBuffer(h_Stoh,SIGNAL_LINE,Start,End,ssig)<=0 ||
   CopyBuffer(h_MFI,0,Start,End,mfi)<=0)
  {
   return;
  }
```

Как уже было сказано выше, не все показатели имеют сопоставимый вид. И хотя линейные преобразования несильно повлияют на коэффициент корреляции, нам все же нужно провести предварительную обработку некоторых величин.

В первую очередь это касается параметров, прямо указывающих на цену инструмента. Ведь мы же хотим воссоздать инструмент, способный проецировать накопленные знания на будущие рыночные ситуации, когда будет схожее ценовое движение, но уже на новом ценовом уровне. Поэтому мы должны уйти от абсолютных ценовых значений в некоторую относительную область.

Так вместо цен открытия и закрытия свечи мы можем взять их разность (размер тела свечи) в качестве показателя интенсивности ценового движения. Свечные экстремумы *High* и *Low* мы также заменим на их отклонение от цены открытия или закрытия свечи. Аналогично поступим с индикаторами *SAR* и *Bollinger Bands*.

Вспомним о классических правилах торговли по индикатору *MACD*. В них помимо самих значений индикатора важно и положение сигнальной линии относительно гистограммы. Для проверки такой зависимости добавим в качестве еще одного показателя разность значений между линиями индикатора.

Отдельно стоит сказать о нашем ориентире ценового движения. Индикатор *ZigZag* дает абсолютные ценовые величины экстремумов на конкретной свече. Нам же желательно знать

ценовой ориентир для каждой рыночной ситуации. Иными словами, нам нужен ценовой ориентир предстоящего движения на каждой отдельно взятой свече. При этом мы будем рассматривать два варианта такого ориентира:

- направление движения (вектор *target1*),
- величина движения (вектор *target2*).

Решить эту задачу мы можем с помощью цикла. Запустим перебор значений индикатора *ZigZag* в обратном порядке (от последних значений к самым старым). Если индикатор нашел экстремум, сохраним его значение в локальную переменную ***extremum***. При отсутствии экстремума используем последнее сохраненное значение.

Одновременно, в этом же цикле, будем рассчитывать и сохранять целевые значения для нашей выборки. Для этого будем от ценового значения последней вершины отнимать цену закрытия анализируемого бара. Таким образом мы получим величину движения до ближайшего будущего экстремума (вектор target2). Знак значения покажет направление движения (вектор target1).

```
int total = ArraySize(close);
double target1[], target2[], oc[], bmc[], buc[], blc[], macd_delta[];
if(ArrayResize(target1, total) <= 0 || ArrayResize(target2, total) <= 0 ||
   ArrayResize(oc, total) <= 0 || ArrayResize(bmc, total) <= 0   ||
   ArrayResize(buc, total) <= 0 || ArrayResize(blc, total)  <= 0 ||
   ArrayResize(macd_delta, total) <= 0)
   return;

double extremum = -1;
for(int i = ArraySize(zz) - 2; i >= 0; i--)
  {
   if(zz[i + 1] > 0 && zz[i + 1] != EMPTY_VALUE)
      extremum = zz[i + 1];
   if(i >= total)
      continue;
   target2[i] = extremum - close[i];
   target1[i] = (target2[i] >= 0);
   oc[i] = close[i] - open[i];
   sar[i] -= close[i];
   bands_low[i] = close[i] - bands_low[i];
   bands_up[i] -= close[i];
   bands_medium[i] -= close[i];
   macd_delta[i] = macd_main[i] - macd_signal[i];
  }
```

После проведенной подготовительной работы проведем непосредственно проверку корреляции данных. Так как мы будем осуществлять одну и ту же операцию для данных различных индикаторов, то есть смысл перенести выполнение данной итерации в отдельную функцию. Из тела функции *Start* будем осуществлять только ее вызов, передавая при этом различные исходные данные. Результаты корреляционного анализа сохраним в *CSV*-файл для дальнейшей обработки.

```
   int handle = FileOpen("correlation.csv", FILE_WRITE | FILE_CSV | FILE_ANSI,
                                                                  "\t", CP_UTF8);
   string message = "Indicator\tTarget 1\tTarget 2";
   if(handle != INVALID_HANDLE)
      return;
   FileWrite(handle, message);
//---
   Correlation(target1, target2, oc, "Close - Open", handle);
   Correlation(target1, target2, hc, "High - Close %.5f", handle);
   Correlation(target1, target2, lc, "Close - Low", handle);
   Correlation(target1, target2, cci, "CCI %.5f", handle);
   Correlation(target1, target2, rsi, "RSI", handle);
   Correlation(target1, target2, atr, "ATR", handle);
   Correlation(target1, target2, sar, "SAR", handle);
   Correlation(target1, target2, macd_main, "MACD Main", handle);
   Correlation(target1, target2, macd_signal, "MACD Signal", handle);
   Correlation(target1, target2, macd_delta, "MACD Main-Signal", handle);

   Correlation(target1, target2, bands_medium, "BB Main", handle);
   Correlation(target1, target2, bands_low, "BB Low", handle);
   Correlation(target1, target2, bands_up, "BB Up", handle);
   Correlation(target1, target2, stoch, "Stochastic Main", handle);
   Correlation(target1, target2, ssig, "Stochastic Signal", handle);
   Correlation(target1, target2, mfi, "MFI", handle);
//---
   FileFlush(handle);
   FileClose(handle);
  }
```

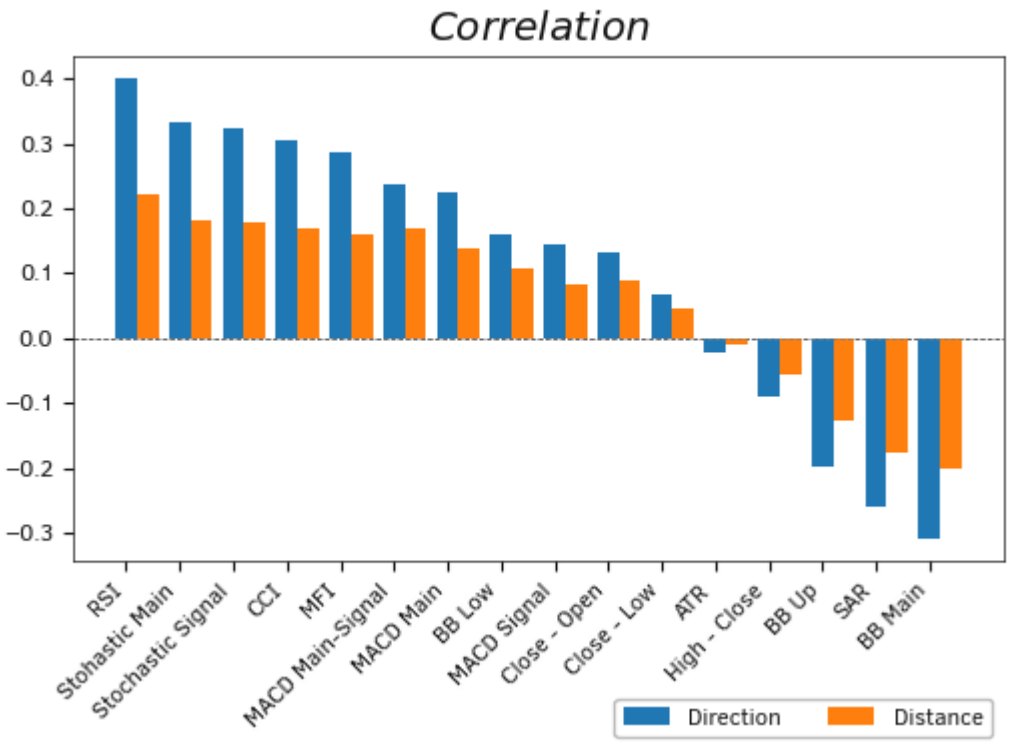
Алгоритм нашего метода проверки корреляции довольно прост. Расчет коэффициента корреляции осуществляет функция из стандартной библиотеки статистического анализа *MQL5* ***MathCorrelationPearson***, которую мы поочередно вызываем для двух наборов данных:

- индикатор и направление предстоящего движения,
- индикатор и сила предстоящего движения.

Из результатов анализа формируется текстовое сообщение и записывается в локальный файл.

```
void Correlation(double &target1[], double &target2[],
                         double &indicator[], string name,
                         int handle)
  {
//---
   double correlation=0;
   string message="";
   if(MathCorrelationPearson(target1,indicator,correlation))
      message=StringFormat("%s\t%.5f",name,correlation);
   if(MathCorrelationPearson(target2,indicator,correlation))
      message=StringFormat("%s\t%.5f",message,correlation);
   if(handle!=INVALID_HANDLE)
      FileWrite(handle,message);
  }
```

Результаты проведенного мной анализа представлены на графике ниже. Полученные данные свидетельствуют об отсутствии корреляции между нашими целевыми данными и значениями индикатора *ATR*. Отклонение от экстремумов свечи до цены ее закрытия (***High — Close***, ***Close — Low***) также демонстрируют низкую корреляцию с ожидаемым ценовым движением. Следовательно, мы можем смело исключить данные показатели из нашего дальнейшего анализа.



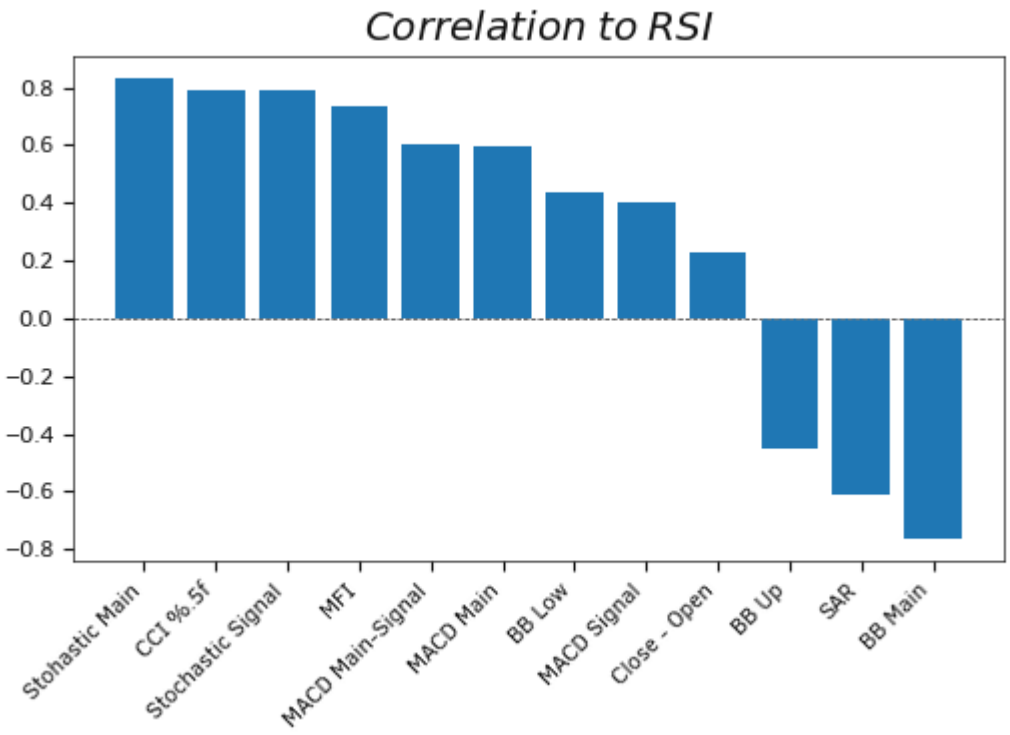
Корреляция показателей индикаторов к ожидаемому ценовому движению

В целом же, проведенный анализ показывает, что определить направление предстоящего движения гораздо проще, чем предсказать его силу. Все индикаторы показали большую корреляцию с направлением, нежели с величиной предстоящего движения. В то же время корреляция со всеми показателями остается довольно низкая. Наибольшую корреляцию показал индикатор *RSI* с показателями 0,40 к направлению и 0,22 к величине движения.

Напомню, что коэффициент корреляции принимает значения от −1 (обратная зависимость) до 1 (прямая зависимость). При 0 полностью отсутствует зависимость между случайными величинами.

Наверное, следует обратить внимание, что из трех массивов данных, полученных от индикатора *MACD* (гистограмма, сигнальная линия и разница между ними), именно расстояние между линиями *MACD* показало наибольшую корреляцию с целевыми данными. Это только подтверждает правильность классического подхода к использованию сигналов индикатора.

Следующим этапом проверим корреляцию между данными различных индикаторов. Чтобы не сравнивать каждый индикатор со всеми остальными, проведем анализ корреляции индикаторов с *RSI* (победителем предыдущего этапа). Работу выполним с помощью ранее созданного скрипта с небольшими изменениями. Новый скрипт сохраним в файле *initial_data_rsi.mq5* нашего подкаталога.



Корреляция показателей индикаторов к RSI

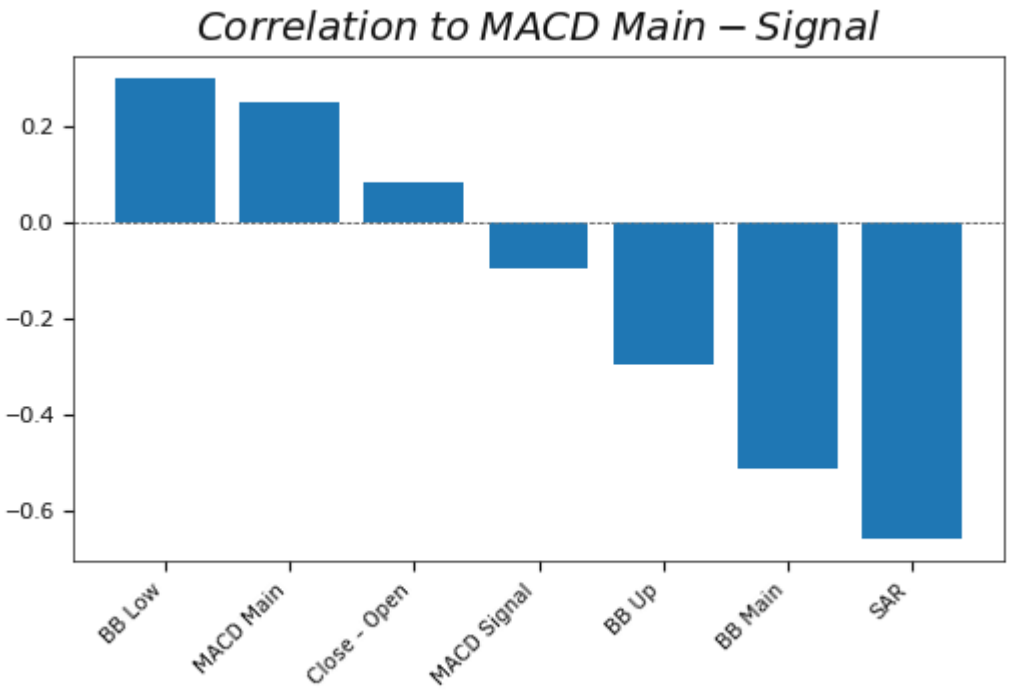
Проведенный анализ продемонстрировал сильную корреляцию *RSI* с целым рядом индикаторов. *Stochastic*, *CCI* и *MFI* имеют коэффициент корреляции с *RSI* более 0,70, а основная линия *Bollinger Bands* показала обратную корреляцию с *RSI* с показателем −0,76. Это свидетельствует о том, что показатели вышеуказанных индикаторов лишь будут дублировать сигналы. Включение их для анализа в нашу нейронную сеть лишь усложнит ее архитектуру и обслуживание. При этом ожидаемый эффект от их использования будет минимален. Поэтому мы исключаем вышеуказанные индикаторы из дальнейшей проработки.

Минимальную корреляцию с *RSI* демонстрируют 2 показателя отклонений:

- сигнальная линия *MACD* (0,40);
- между ценами открытия и закрытия (0,23).

Отклонение сигнальной линии *MACD* от гистограммы на первом этапе показало большую корреляцию с целевыми данными предстоящего ценового движения. Опираясь на эти данные, именно *MACD* возьмем в нашу корзину показателей. Далее проверим корреляцию с ним оставшихся индикаторов.

Обновленный скрипт сохраним в файле *initial_data_macd.mq5* подкаталога.



Корреляция показателей индикаторов к MACD Main-Signal

Здесь довольно интересные данные показывает индикатор *SAR*. При средних показателях обратной корреляции с целевыми данными он демонстрирует довольно высокую обратную корреляцию с обоими отобранными индикаторами. Коэффициент корреляции с *MACD* составил −0,66, а для *RSI* — −0,62. Это дает нам основание для исключения индикатора *SAR* из корзины анализируемых показателей.

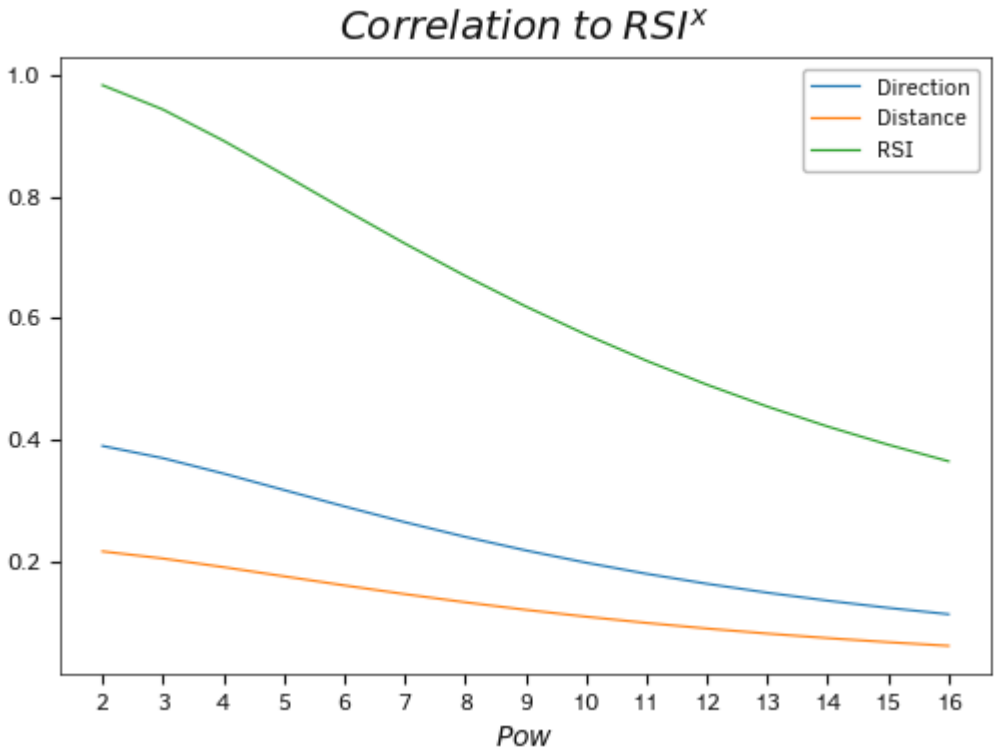
Аналогичная ситуация по всем трем линиям индикатора *Bollinger Bands*.

Таким образом, на текущий момент мы отобрали два индикатора в нашу корзину показателей для последующего обучения нейронной сети.

Но это еще не конец на пути по отбору исходных данных. Следует сказать, что нейронная сеть анализирует линейные зависимости между целевыми значениями и исходными данными в чистом виде. То есть анализируются те данные, которые ей поданы на вход. При этом каждый показатель анализируется в отрыве от других данных, имеющихся на входном слое нейронов.

Поэтому нейронная сеть не построит зависимость между исходными данными и целевыми показателями, если она не линейная, а, к примеру, степенная или логарифмическая. Для поиска таких зависимостей нам необходимо предварительно подготовить данные. А для проверки целесообразности такой работы мы должны проверить наличие корреляции между такими значениями.

В скрипте *initial_data_rsi_pow.mq5* проведем анализ изменения корреляции с ожидаемым ценовым движением при возведении в различную степень значений индикатора RSI. Новый скрипт разместим в соответствующем подкаталоге.



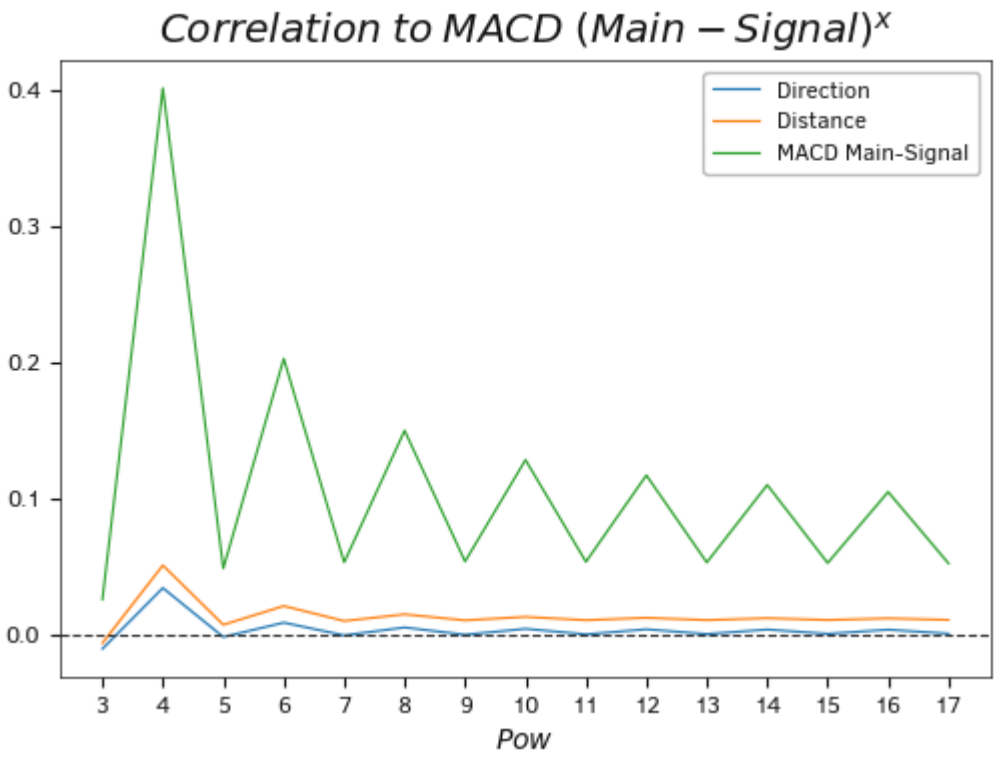
Динамика изменения корреляции значений RSI к ожидаемому движению при возведении показателя в степень

На представленном графике четко прослеживается, что с ростом степени возведения значений индикатора корреляция с исходными значениями снижается значительно быстрее чем с ожидаемым ценовым движением. Это наблюдение дает нам потенциальную возможность расширить корзину показателей исходных данных степенными значениями отобранных показателей. Чуть позже мы сможем посмотреть, как это работает на практике.

Следует обратить внимание, что при использовании степенных операций над показателями необходимо помнить о свойствах степеней и природы значений показателя. Так при возведении любого числа в четную степень результат всегда будет положительным. Иными словами, мы теряем знак числа.

К примеру, у нас есть два равных по модулю значения одного показателя с разными знаками, и они коррелируют с целевыми данными. При положительном значении показателя целевая функция растет, а при отрицательных падает. Возведение в квадрат значений такого показателя даст нам одинаковое значение. В таком случае мы увидим снижение корреляции или полное ее отсутствие, так как наша целевая функция осталась неизменной, в то время как показатель лишился знака.

Этот эффект хорошо заметен при анализе изменения корреляции при возведении в степень разницы между линиями индикатора MACD, который мы провели в скрипте *initial_data_macd_pow.mq5* нашего подкаталога.



Динамика изменения корреляции значений MACD к ожидаемому движению при возведении показателя в степень

Аналогичным образом можно проверить корреляцию различных показателей с целевыми значениями. Ограничением являются только ваши возможности и здравый смысл. Помимо стандартных индикаторов и ценовых котировок это могут быть и пользовательские индикаторы, котировки других инструментов, в том числе и синтетических. Не бойтесь экспериментировать — иногда можно найти хорошие показатели в самых неожиданных местах.

Влияние временного сдвига на коэффициент корреляции

После выбора показателей для анализа вспомним, что мы имеем дело с временными рядами. Одной из особенностей временных рядов является их «историческая память». Каждое последующее значение является зависимым не только от одного предыдущего значения, но и от некой глубины исторических значений.

Можно, конечно, пойти экспериментальным путем — построить нейронную сеть и после ряда экспериментов выбрать оптимальный вариант. Такой подход потребует времени на создание нескольких экспериментальных моделей и их обучение. А можно воспользоваться нашей выборкой и проверить корреляцию данных со сдвигом в историю.

Для решения данной задачи немного изменим скрипт и заменим в нем функцию *Correlation* на ***ShiftCorrelation***. Новая функция является полным наследником функции *Correlation* и построена по тому же алгоритму.

В параметрах функции добавим новую переменную *max_shift*, в которую передадим максимальный сдвиг для анализа.

Смещение по времени организуем путем копирования данных в новые массивы со сдвигом. Исходные данные будут копироваться без смещения, но в меньшем объеме. Сокращение

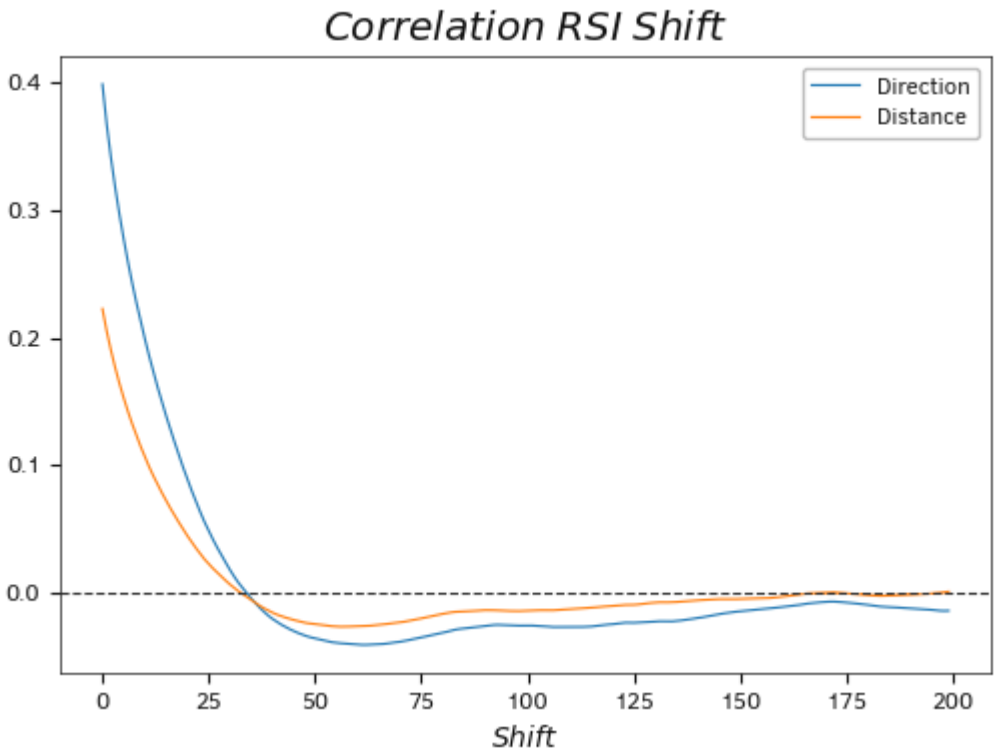
количества данных соответствует смещению во времени. В то же время данные о целевых значениях будем переносить в новые массивы со смещением. Но так как размер целевых данных в нашей выборке постоянный, то при смещении количество элементов для анализа корреляции сокращается. Таким образом, после копирования данных мы получаем массивы данных, сопоставимые по размеру, и с заданным сдвигом во времени.

Остается только вызвать функцию расчета коэффициента корреляции и записать полученные данные в файл.

Для анализа изменения корреляции с ростом смещения во времени обернем все операции в цикл. Количество циклических итераций соответствует параметру *max_shift*.

```
void ShiftCorrelation(double &targ1[], double &targ2[],
                      double &signal[], string name,
                      int max_shift, int handle)
  {
   int total = ArraySize(targ1);
   if(max_shift > total)
      max_shift = total - 10;
   if(max_shift < 10)
      return;
   double correlation = 0;
   for(int i = 0; i < max_shift; i++)
     {
      double t1[], t2[], s[];
      if(ArrayCopy(t1, targ1, 0, i, total - i) <= 0 ||
         ArrayCopy(t2, targ2, 0, i, total - i) <= 0 ||
         ArrayCopy(s, signal, 0, 0, total - i) <= 0)
        {
         continue;
        }
      //---

      string message;
      if(MathCorrelationPearson(s, t1, correlation))
         message = StringFormat("%d\t%.5f", i, correlation);
      if(MathCorrelationPearson(s, t2, correlation))
         message = StringFormat("%s\t%.5f", message, correlation);
      if(handle != INVALID_HANDLE)
         FileWrite(handle, message);
     }
//---
   return;
  }
```



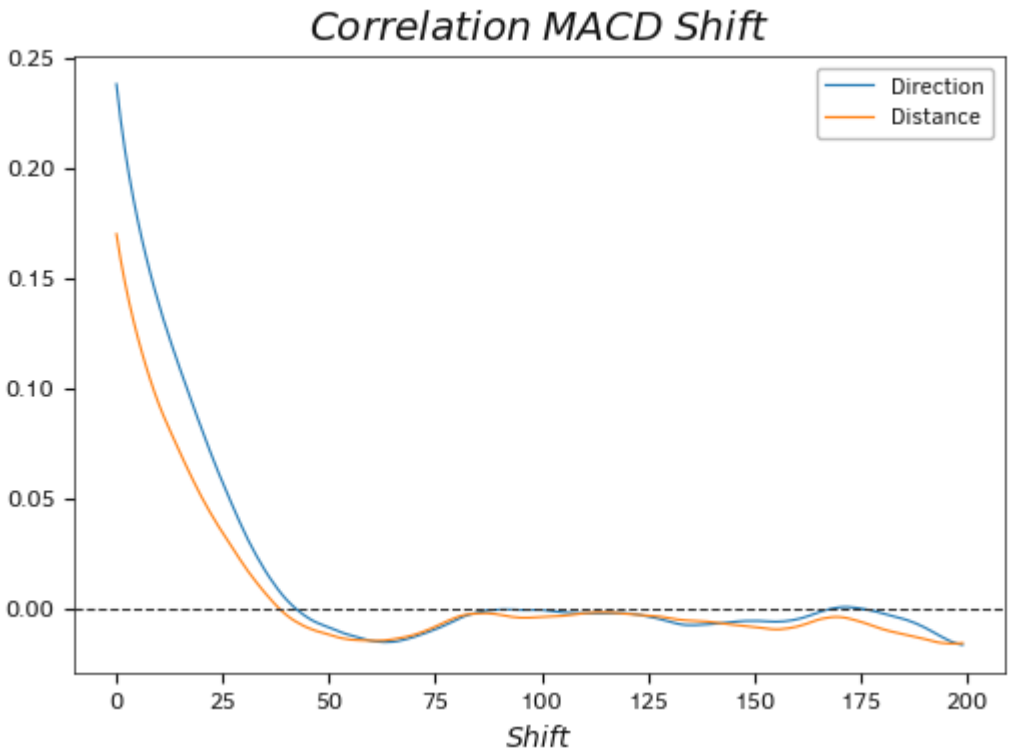
Динамика изменения корреляции значений RSI к ожидаемому движению при сдвиге во времени

Чтобы проанализировать влияние смещения значений индикатора *RSI* во времени на корреляцию с целевыми данными, создадим новый скрипт в файле *initial_data_rsi_shift.mq5* подкаталога.

Результаты проведенного анализа показывают стремительное снижение корреляции до 30-го бара. Затем наблюдается небольшая обратная корреляция с пиковым коэффициентом −0,042 в районе 60-го бара и последующим плавным приближением к 0. В такой ситуации наибольшую эффективность даст нам использование первых 30 баров. Дальнейшее расширение глубины анализа может привести к снижению эффективности использования вычислительных ресурсов. Ценность такого решения можно проверить на практике.

Аналогичный анализ данных индикатора MACD в скрипте *initial_data_macd_shift.mq5* показал схожую динамику с небольшим смещение зоны перехода из прямой в обратную корреляцию в район 40-го бара.

Таким образом, проведение анализа корреляции доступных исходных данных и целевых значений позволяет нам на этапе подготовительной работы выбрать оптимальную корзину показателей и глубину истории, необходимой для анализа данных и прогнозирования целевых показателей. Это позволяет нам при относительно небольших трудозатратах на подготовительном этапе значительно сократить расходы на этапах проектирования и обучения нейронной сети.



Динамика изменения корреляции значений MACD к ожидаемому движению при сдвиге во времени

3.4 Закладываем скелет будущей программы MQL5

В предыдущем разделе мы поговорили о подготовительной работе и методах отбора показателей для анализа нейронной сетью. После проведения анализа мы определили корзину показателей для изучения нейронной сетью и глубину загружаемых исторических данных.

Теперь перейдем к практической части нашей книги. В ней мы рассмотрим различные алгоритмы и архитектуры нейронных сетей. Вы узнаете особенности построения и реализации полносвязного перцептрона, сверточных и рекуррентных нейронных сетей. Затем мы поговорим об особенностях и преимуществах механизмов внимания. И в заключение разберем архитектуру *GPT*, которая на момент написания книги демонстрирует наилучшие результаты в задачах обработки естественного языка.

По мере рассмотрения алгоритмов мы шаг за шагом создадим инструмент для проектирования и организации нейронных сетей средствами *MQL5*. Каждый рассматриваемый алгоритм будет реализован в трех вариантах:

- *MQL5*
- *OpenCL*
- *Python*

Мы построим и обучим нейронные сети с использованием всех изученных алгоритмов и на практике оценим достоинства и недостатки их использования для прогнозирования временных рядов. Обучать и тестировать построенные модели будем на реальных данных. И конечно, в процессе обучения мы обсудим нюансы этого процесса.

В книге будут продемонстрированы практические результаты использования нейронных сетей для решения задачи, определенной в предыдущих разделах, на реальных данных. В процессе тестирования мы сделаем сравнительный анализ различных реализаций и оценим практическую стоимость каждой реализации в решении поставленной задачи.

Начнем работу с проработки архитектуры нашего будущего инструмента. Вполне логично объединить всю нашу разработку в некий объект (класс), который будет легко подключаться к любой программе. Таким образом, мы сможем настроить всю работу нашей модели внутри этого класса.

При этом нужно предусмотреть возможность создавать модели различной архитектуры внутри нашей модели. Сама же архитектура модели будет задаваться в основной программе и передаваться в класс посредством созданных интерфейсов. Чтобы сделать этот процесс удобным и понятным в использовании, необходимо стандартизировать его. В вопросах стандартизации хорошо помогаю константы и именованные перечисления.

3.4.1 Определяем константы и перечисления

Процесс определения констант — один из тех базовых процессов, которому часто не уделяют должного внимания. При этом он позволяет организовать и систематизировать будущую работу по созданию программного продукта. Особое внимание ему следует уделить при создании сложных структурированных продуктов, имеющих многоблочную разветвленную архитектуру.

Здесь мы не будем говорить о каких-то локальных переменных и константах, так как их зона действия часто будет определена отдельным блоком или функцией. Мы поговорим о создании констант, которые красной нитью пройдут через всю нашу программу и часто будут использоваться элементами организации взаимодействия как между блоками нашего продукта, так и для обмена данными с внешней программой.

Начинать работу над крупным проектом с создания констант и перечислений — очень полезная практика. Сюда мы можем добавить и создание глобальных переменных.

Прежде всего, это одна из составных частей разработки архитектуры проекта. Задумываясь над списком глобальных констант и перечислений, мы еще раз оцениваем наш проект в целом, переосмысливаем его задачи и средства их решения. Пусть пока и крупными блоками, но мы продумываем структуру проекта, определяем задачи каждого блока и поток информации между ними. Также понимаем, какую информацию необходимо получить от внешней программы, какую вернуть и на каком этапе.

Работа, проделанная на данном этапе, будет нашей «дорожной картой» в создании проекта. Детальное рассмотрение организации интерфейсов обмена данными позволяет оценить необходимость наличия той или иной информации на каждом этапе. Также это дает возможность определить источники информации и выявить возможный дефицит данных. А ликвидировать дефицит данных на стадии проектирования будет гораздо легче, чем уже в процессе реализации, ведь тогда нам придется возвращаться на стадию проектирования для поиска источников необходимых данных. Затем необходимо будет продумывать возможные варианты передачи информации от источника к месту ее обработки и попытаться «впихнуть» в уже выстроенную архитектуру с минимальными доработками. Это повлечет за собой непредсказуемое количество правок в уже выстроенных процессах, при этом необходимо будет оценить влияние этих правок на смежные процессы.

Все файлы выстраиваемой библиотеки мы соберем в подкаталоге *NeuroNetworksBook\realization* в соответствие со структурой файлов.

Все глобальные константы нашего проекта соберем в один файл *defines.mqh*.

Какие же константы мы будем определять?

Давайте посмотрим на архитектуру проекта. Мы уже проговорили, что результатом нашей работы будет некий класс, охватывающий полную организацию работы нейронной сети. В архитектуре *MQL5* все объекты наследуются от базового класса *CObject*. В нем определен виртуальный метод для идентификации класса *Type*, который возвращает целочисленное значение. Следовательно, для однозначной идентификации нашего класса мы должны определить некую константу, желательно отличную от констант уже имеющихся классов. Это будет прототипом визитной карточки нашего класса внутри программы. Для создания именованных констант воспользуемся механизмом макроподстановки.

```
#define defNeuronNet             0x8000
```

Далее наша нейронная сеть будет состоять из нейронов. Нейроны объединяются в слои, а нейронная сеть может состоять из нескольких слоев. Так как мы строим некий универсальный конструктор, то на данном этапе мы не знаем ни количество слоев в нейронной сети, ни количество нейронов в каждом слое. Следовательно, мы предполагаем, что будет некий динамический массив для хранения указателей на слои нейронов. Вероятнее всего, помимо простого хранения указателей на объекты нейронных слоев нам потребуется создание дополнительных методов для работы с ними. Из этих соображений мы будем создавать отдельный класс для такого хранилища. Следовательно, создадим и для него визитную карточку.

```
#define defArrayLayers           0x8001
```

Дальше по структуре будет создаваться отдельный класс для нейронного слоя. Позже, когда мы подойдем к вопросам реализации алгоритмов вычислений с использованием технологии *OpenCL*, мы поговорим об организации векторных вычислений и средствах передачи данных в память *GPU*. В данном ключе будет не очень удобно создавать классы для каждого нейрона, но потребуется класс для хранения информации и буферной организации обмена данными. Таким образом, мы должны создать «визитные карточки» и для этих объектов.

Надо сказать, что в книге будут рассмотрены несколько архитектурных решений организации нейронов. При этом каждая архитектура имеет свои особенности алгоритмов прямого и обратного прохода. Но мы уже определились, что не будем создавать отдельные объекты для нейронов. Значит, нужно ввести идентификацию на уровне нейронных слоев. Поэтому создадим отдельные идентификаторы для каждой архитектуры нейронного слоя.

```
#define defBuffer                0x8002
#define defActivation            0x8003
#define defLayerDescription      0x8004
#define defNeuronBase            0x8010
#define defNeuronConv            0x8011
#define defNeuronProof           0x8012
#define defNeuronLSTM            0x8013
#define defNeuronAttention       0x8014
#define defNeuronMHAttention     0x8015
#define defNeuronGPT             0x8016
#define defNeuronDropout         0x8017
#define defNeuronBatchNorm       0x8018
```

С константами идентификаторов объектов определились, двигаемся дальше. Давайте вспомним, с чего начинается данная книга. В самом начале книги мы рассмотрели математическую модель

нейрона. Каждый нейрон имеет функцию активации. Мы рассмотрели несколько вариантов функций активации, и все они имеют право на жизнь. Ввиду отсутствия производной мы исключим из списка пороговую функцию, но остальные рассмотренные функции реализуем с использование технологии *OpenCL*. В случае работы на *CPU* воспользуемся векторными операциями, в которых уже реализованы функции активации. Чтобы сохранить единство подходов, для указания используемой функции активации воспользуемся стандартным перечислением указания функции *ENUM_ACTIVATION_FUNCTION*.

Однако стоит отметить, что позже, при рассмотрении алгоритмов сверточных сетей, мы познакомимся с организацией подвыборочного слоя. В нем используются другие функции.

```
//--- функции активации подвыборочного слоя
enum ENUM_PROOF
  {
   AF_MAX_POOLING,
   AF_AVERAGE_POOLING
  };
```

Посмотрим на главу «Обучение нейронной сети». В ней мы говорили о различных вариантах функции потерь и методов оптимизации нейронных сетей. В моем понимании мы должны дать возможность пользователю выбрать, что он хочет использовать. Однако нужно ограничить выбор возможностями нашей библиотеки. Если для функции потерь мы можем использовать стандартное перечислением *ENUM_LOSS_FUNCTION* по аналогии с функцией активации, то для методов оптимизации моделей создадим новое перечисления.

Как можно заметить, в перечисление методов оптимизации я добавил элемент *None* для возможности отключения обучения отдельно взятого слоя. Такой подход часто применяется при использовании предварительно обученной сети на новых данных. К примеру, у нас есть уже обученная и рабочая нейронная сеть, которая отлично работает на одном из инструментов, и мы бы хотели тиражировать ее на другие инструменты или таймфреймы. С большой долей вероятности без переобучения ее результативность сильно упадет.

```
enum ENUM_OPTIMIZATION
  {
   None=-1,
   SGD,
   MOMENTUM,
   AdaGrad,
   RMSProp,
   AdaDelta,
   Adam
  };
```

В таком случае у нас есть выбор: обучать нейронную сеть с чистого листа или переобучить существующую сеть. Второй вариант обычно требует меньших затрат времени и ресурсов. Но чтобы не разбалансировать всю сеть, переобучение начинается с малым коэффициентов обучения (learning rate) и на последних слоях (нейроны принятия решения), оставляя без обучения первые аналитические слои.

После методов обучения мы с вами говорили о приемах повышения сходимости нейронных сетей. Здесь все довольно просто. Нормализация и dropout будут организованы отдельными слоями — для них мы уже задали константы при рассмотрении нейронных слоев. Регуляризацию предлагаю реализовать одну — *Elastic Net.* Управлять процессом будем через переменные λ1 и

λ2. Если обе переменные равны нулю, регуляризация отключена. В случае, когда один из параметров равен нулю, получим L1 или L2-регуляризацию, в зависимости от ненулевого параметра.

Вы заметили, как в этой главе мы освежили в памяти основные вехи изученного материала? При этом за каждой константой или элементом перечисления стоит определенный функционал, который нам предстоит еще реализовать.

Но я бы хотел добавить еще один момент. При знакомстве с технологией *OpenCL* мы обсуждали, что не все устройства с поддержкой *OpenCL* работают с типом *double*. Наверное, было бы глупо создавать копии библиотеки для различных типов данных.

Тут надо понимать, что различные типы данных обеспечивают различную точность вычислений. Поэтому, создавая модель, нужно обеспечить аналогичные условия для всех вариантов работы модели, как с использованием технологии *OpenCL*, так и без. Для решения этого вопроса мы добавим макроподстановку типа данных, а вместе с этим и соответствующих типов для векторов и матриц.

```
#define TYPE                    double
#define MATRIX                  matrix<TYPE>
#define VECTOR                  vector<TYPE>
```

Аналогичную макроподстановку организуем и для *OpenCL*-программы.

```
#resource "opencl_program.cl" as string OCLprogram
//---
#define LOCAL_SIZE              256
const string ExtType=StringFormat("#define TYPE %s\r\n"
                                  "#define TYPE4 %s4\r\n"
                                  "#define LOCAL_SIZE %d\r\n",
                                   typename(TYPE),typename(TYPE),LOCAL_SIZE);
#define cl_program              ExtType+OCLprogram
```

Здесь же мы можем добавить различные гиперпараметры моделей. К примеру, это может быть коэффициент обучения. Также можно добавить параметры методов оптимизации и регуляризации.

```
#define defLossSmoothFactor     1000
#define defLearningRate         (TYPE)3.0e-4
#define defBeta1                (TYPE)0.9
#define defBeta2                (TYPE)0.999
#define defLambdaL1             (TYPE)0
#define defLambdaL2             (TYPE)0
```

Однако стоит помнить, что указанные здесь значения гиперпараметров являются лишь значениями по умолчанию. В процессе работы модели мы будем использовать переменные, которые при создании модели будут инициализированы этими значения. Но пользователь вправе указать другие значения без изменения кода библиотеки. Механизм такого процесса мы обсудим при построении классов и их методов.

3.4.2 Механизм описания структуры создаваемой нейронной сети

Мы уже определились, что будем строить универсальный конструктор для удобного создания нейронных сетей различной конфигурации. Следовательно, нам нужен некий механизм (интерфейс) для возможности передачи конфигурации модели для построения. Давайте подумаем, какую информацию нам нужно получить от пользователя для однозначного понимания, какую нейронную сеть предполагается построить.

Прежде всего нам нужно понимать, сколько слоев нейронов будет в нашей сети. Таких слоев должно быть как минимум два: входной слой исходных данных и выходной слой результатов. Дополнительно новая нейронная сеть может включать различное количество скрытых слоев. Их количество может быть различно, и мы не будем сейчас их ограничивать.

Для создания каждого слоя нейронной сети нам нужно знать количество нейронов в данном слое. Следовательно, помимо количества нейронных слоев пользователь должен указать количество нейронов в каждом слое.

Теперь давайте вспомним, что в предыдущем разделе мы определи константы для нескольких типов нейронных слоев, которые будут отличаться по типу нейронов. Чтобы понимать, какой именно слой хочет создать пользователь, нужно получить эту исходную информацию. Значит, пользователь должен иметь возможность ее указать для каждого создаваемого слоя.

Кроме того, мы рассматривали различные варианты функций активации. Какую из них нужно использовать при создании нейронов?

При создании универсального инструмента мы должны предоставить возможность выбора функции активации пользователю. Следовательно, добавляем функцию активации в список параметров для получения от пользователя.

Возникает еще один вопрос: будут ли все нейроны одного слоя использовать одну функцию активации? Или же будут варианты использования различных функций активации в рамках одного слоя? Я предлагаю остановиться на первом варианте, когда все нейроны одного слоя используют одну функцию активации.

Поясню свою позицию. Обсуждая приемы повышения сходимости нейронных сетей и, в частности, нормализацию данных, мы говорили о важности сопоставимости данных на входе нейронного слоя. Использование же различных функций активации с высокой долей вероятности приведет к дисбалансу данных. Это связано с природой самих функций активации. Вспомните, сигмоида возвращает данные в диапазоне от 0 до 1. Область значений гиперболического тангенса лежит в диапазоне от −1 до 1. А *ReLU* может вернуть значения от 0 до $+\infty$. Очевидно, что разные функции активации дадут сильно отличающиеся значения и лишь затруднят обучение и эксплуатацию нейронной сети.

Кроме того, с технической стороны тоже есть плюсы использования одной функции активации для всего нейронного слоя. В таком случае мы можем ограничиться одним целочисленным значением для хранения кода активации нейронов в слое независимо от количества нейронов. В то время как для хранения индивидуальных функций активации нам бы пришлось создавать целый вектор значений размером в число нейронов слоя.

Следующее, что нам потребуется знать при создании архитектуры нейронной сети, — это метод оптимизации весовых коэффициентов. Помните, в главе «Методы оптимизации нейронных сетей» мы рассмотрели шесть методов оптимизации. В предыдущей главе мы задали

перечисление для их идентификации. Теперь можно воспользоваться этим перечислением и предоставить пользователю выбрать один из них.

Почему нам важно знать метод оптимизации сейчас, на стадии создания нейронной сети, а не ее обучения? Тут все очень просто. Разные методы оптимизации требуют разное количество объектов для хранения информации, поэтому при создании нейронной сети нужно создать все необходимые объекты. При этом так как у нас есть технические ограничения по памяти вычислительной машины, мы должны рационально ее использовать и не создавать лишние объекты.

При создании таких слоев как нормализация и *Dropout* нам потребуется некоторая специфическая информация. Для нормализации нам потребуется размер выборки нормализации (*batch*), а для *Dropout* нужно указать вероятность «выкидывания» нейронов в процессе обучения.

Забегая немного вперед, скажу о том, что для некоторых типов нейронных слоев нам еще потребуются размер входного и выходного окна. А также размер шага от начала одного входного окна до начала следующего окна.

Чтобы облегчить пользователю процесс создания последовательно идущих одинаковых слоев, добавим еще параметр для указания такой последовательности.

Таким образом, у нас набрался десяток параметров, которые пользователь должен указать для каждого слоя. Добавим сюда же общее количество слоев для создания в нейронной сети. Все это мы хотим получить от пользователя перед созданием нейронной сети. Мы не будем сильно усложнять процесс передачи данных, и для описания одного нейронного слоя создадим класс ***CLayerDescription*** с элементами для хранения указанных параметров.

```
class CLayerDescription    :  public CObject
  {
public:
                     CLayerDescription(void);
                    ~CLayerDescription(void) {};
   //---
   int               type;         // Тип нейронного слоя
   int               count;        // Количество нейронов в слое
   int               window;       // Размер окна исходных данных
   int               window_out;   // Размер окна результатов
   int               step;         // Шаг окна исходных данных
   int               layers;       // Количество нейронных слоев
   int               batch;        // Размер пакета обновления матрицы весов
   ENUM_ACTIVATION_FUNCTION   activation;  // Тип функции активации
   VECTOR            activation_params[2]; // Массив параметров функции активации
   ENUM_OPTIMIZATION optimization; // Тип оптимизации матрицы весов
   TYPE              probability;  // вероятность маскирования, только Dropout
  };
```

Обратите внимание, что создаваемый класс наследуется от класса *CObject*, который является базовым классом для всех объектов в *MQL5*. Это небольшой момент, который мы будет эксплуатировать немного позже.

Конструктор класса мы не будем чем-либо усложнять, а зададим лишь некоторые значения по умолчанию. Здесь вы можете использовать любые свои значения. Я же рекомендую указать

наиболее часто используемые параметры. Это облегчит вам в последующем их указание в коде программы.

```
CLayerDescription::CLayerDescription(void)   :  type(defNeuronBase),
                                                count(100),
                                                window(100),
                                                step(100),
                                                layers(1),
                                                activation(AF_TANH),
                                                optimization(Adam),
                                                probability(0.1),
                                                batch(100)
  {
   activation_params = VECTOR::Ones(2);
   activation_params[1] = 0;
  }
```

Теперь вернемся к тому, почему важно было наследоваться от *CObject*. Здесь все довольно просто и прозаично: мы создали объект для описания одного нейронного слоя, но не всей нейронной сети. Мы еще не указали общее количество слоев и их последовательность.

Я решил не усложнять процесс и воспользоваться классом ***CArrayObj*** из стандартной библиотеки *MQL5*. Это класс динамического массива для хранения указателей на объекты *CObject* и их наследников. Следовательно, мы можем записать в него наши объекты описания нейронного слоя. Таким образом мы решаем вопрос с контейнером для хранения и передачи информации о нейронных сетях. Последовательность нейронных слоев будет соответствовать последовательности сохраненных описаний от входного слоя с нулевым индексом до выходного слоя.

На мой взгляд, это довольно простой и интуитивно понятный способ описания структуры нейронной сети. Но каждый читатель может воспользоваться своими наработками.

3.4.3 Базовый класс нейронной сети и организация процессов прямого и обратного проходов

Мы уже провели подготовительную работу по созданию констант и интерфейса для передачи архитектуры создаваемой нейронной сети. Продолжаем двигаться дальше. Сейчас я предлагаю перейти к созданию класса верхнего уровня ***CNet***, который будет выступать менеджером работы нашей нейронной сети.

Для выполнения данной работы мы создадим новый файл включаемой библиотеки *neuronnet.mqh* в подкаталоге нашей библиотеки. В нем мы соберем весь код нашего класса нейронной сети *CNet*. Далее будем создавать отдельный файл для каждого нового класса. Наименования файлов будут соответствовать именам классов — это позволит структурировать проект и довольно быстро получать доступ к коду отдельно взятого класса.

Мы не сможем сейчас написать полный код методов этого класса, так как в процессе их реализации нам нужно будет обращаться к классам нейронных слоев и их методам. На данный момент этих классов еще нет. Почему же я решил начать с создания объекта верхнего уровня, а не создал сначала объекты более низкого уровня? Здесь я решаю вопрос целостности структуры и стандартизации методов и интерфейсов передачи данных между отдельными блоками нашей нейронной сети.

Позже, при рассмотрении архитектурных особенностей нейронных слоев, вы сможете заметить различие в их функционале и отчасти в потоке информации. При решении задачи снизу мы рискуем получить довольно различные методы и интерфейсы, которые потом будет сложно увязать в единую систему. Я же, напротив, хочу сразу создать верхнеуровневый «скелет» нашей разработки и позже наполнять его функционалом. При этом сразу продумав архитектуру и функционал интерфейсов, мы просто будем вписывать новые архитектуры нейронных слоев в уже созданный информационный поток.

Давайте определимся с функционалом класса *CNet*. Первое, что должен выполнять данный класс — это непосредственно собрать нейронную сеть с полученной от пользователя архитектурой. Это можно сделать в конструкторе класса, а можно создать отдельный метод ***Create***. Я предпочел второй вариант. Использование базового конструктора класса без параметров позволит нам создавать «пустой» экземпляр класса, к примеру, для загрузки ранее обученной нейронной сети. Также это облегчит наследование класса для возможного последующего развития.

И раз уж мы затронули вопрос загрузки предварительно обученной сети, то отсюда вытекает следующий функционал класса: сохранение (***Save***) и загрузка (***Load***) нашей модели.

Будь то вновь созданные (сгенерированные) нейронные слои или загруженные из файла, нам нужно будет хранить их и работать с ними. При проработке и определении констант мы выделили отдельную константу для динамического массива хранения нейронных слоев. Экземпляр этого объекта мы добавим в переменные класса (***m_cLayers***).

Посмотрим в сторону непосредственной организации работы нейронной сети. Здесь мы должны реализовать алгоритмы прямого (***FeedForward***) и обратного прохода (***Backpropagation***). Выведем отдельным методом процесс обновления весовых коэффициентов ***UpdateWeights***.

Конечно, обновлять весовые коэффициенты можно и в методе обратного прохода, что на практике чаще всего и встречается. Но мы говорим об универсальном конструкторе. На момент написания кода мы не знаем, будет ли использоваться пакетная нормализация, размер пакета. Следовательно, нет четкого понимания, в какой момент нужно будет осуществить обновление весов.

Сложную задачу всегда легче решать по частям. Разделение процесса на более мелкие подпроцессы облегчает как написание кода, так и его отладку. Поэтому процесс обновления весовых коэффициентов я решил вынести отдельно.

Давайте вспомним методы оптимизации нейронов. Практически все методы используют коэффициент обучения, а некоторые требуют дополнительные параметры, такие как коэффициенты затухания. Мы также должны предоставить возможность пользователю указать их. При этом пользователь указывает один раз, а нам они потребуются на каждой итерации. Значит, нам их где-то надо сохранить. Добавим метод для указания параметров обучения (***SetLearningRates***) и переменные для хранения данных (***m_dLearningRate*** и ***m_adBeta***). Для коэффициентов затухания создадим вектор из двух элементов, что, на мой взгляд, сделает код более читабельным.

В процессе практического использования нейронной сети пользователю может потребоваться получение результатов обработки одних и тех же исходных данных несколько раз. Мы не должны ограничивать его в этом. Но чтобы каждый раз не осуществлять прямой проход, мы выведем отдельным методом ***GetResults*** возможность получения результатов последнего прямого прохода.

Кроме того, в процессе обучения и эксплуатации нейронной сети нам будет необходимо контролировать процесс точности и корректности данных прямого прохода. Основным показателем корректной работы нейронной сети является значение функции потерь.

Непосредственно расчет функции потерь будет осуществляться в методе обратного прохода ***Backpropagation***. Посчитанное значение функции потерь сохраним в переменную ***m_dNNLoss.*** Добавим метод ***GetRecentAverageLoss*** для вывода значения переменной по запросу пользователя.

Кстати о функции потерь. Выбор функции потерь для использования предоставляется пользователю. Значит, нам нужен метод для возможности получения ее от пользователя (***LossFunction***). Непосредственно расчет значения функции потерь будет осуществляться стандартными средствами матричных операций в *MQL5*. Здесь же мы создадим переменную для хранения типа функции потерь (***m_eLossFunction***).

При определении констант мы не стали отдельно создавать перечисление для методов регуляризации. Тогда мы договорились реализовать *Elastic Net* и управлять процессом через коэффициенты регуляризации. Я предлагаю добавить указание коэффициентов регуляризации в метод указания функции потерь. Ведь посмотрите, как растет количество методов класса. Поэтому вопрос не только в реализации нашего конструктора. Наоборот, при строительстве конструктора необходимо предусмотреть все возможные варианты использования. Это поможет сделать его более гибким.

В то же время практическое использование такого конструктора должно быть максимально легким и интуитивно понятным. Иными словами, мы должны предоставить пользователю такой интерфейс, который позволит максимально гибко настроить новую нейронную сеть при минимальном количестве итераций, требуемых от пользователя.

Заметим, что алгоритм работы слоев нормализации и *Dropout* отличаются в зависимости от режима использования (обучение или эксплуатация). Конечно, это можно было сделать отдельным параметром в методах прямого и обратного прохода, но здесь важно четкое соответствие работы прямого и обратного проходов. Проведение обратного прохода режима обучения после рабочего прямого прохода и наоборот могут только разбалансировать нейронную сеть, поэтому, чтобы не нагружать дополнительными контролями указанные методы, сделаем отдельную функции для указания и запроса режима работы ***TrainMode***.

Есть еще один момент, касающийся варианта работы нейронной сети, а точнее, выбора инструмента проведения вычислительных операций. Мы уже обсуждали тему использования технологии *OpenCL* для параллельных вычислений. Это позволит производить параллельное вычисление математических операций на *GPU* и ускорить вычисления в процессе работы нейронной сети. Для работы с *OpenCL* в *MQL5* предлагается класс *COpenCL* в стандартной библиотеке *OpenCL.mqh*.

В процессе работы с данным классом я решил немного дополнить его функционал, для чего создал новый класс ***CMyOpenCL*** наследником стандартного класса *COpenCL*. Наследование позволило мне дописать лишь код пары методов и при этом использовать всю мощь родительского класса.

Для использования класса *CMyOpenCL* добавим указатель на экземпляр класса ***m_cOpenCL***. Также добавим флаг ***m_bOpenCL***, который подскажет, включен ли функционал в нашей нейронной сети. Также добавим методы для инициализации функционала и управления им (***InitOpenCL, UseOpenCL***).

Не будем забывать, что мы планируем использование нейронных сетей для работы с таймсериями. Это накладывает определенный отпечаток на их работу. Помните график оценки корреляции исходных данных с временным сдвигом? С ростом временного сдвига падает влияние показателя на целевой результат. Это лишний раз подтверждает важность учета

позиции анализируемого показателя на линейке времени. Следовательно, нужно будет реализовать подобный механизм.

О самом методе мы поговорим чуть позже. А сейчас создадим экземпляр класса ***CPositionEncoder*** для реализации позиционного кодирования, флаг контроля активности функционала и объявим методы управления функцией.

К нашему списку добавим еще метод идентификации класса и получим нижеследующую структуру класса *CNet*.

```
class CNet  : public CObject
  {
protected:
   bool               m_bTrainMode;
   CArrayLayers*      m_cLayers;
   CMyOpenCL*         m_cOpenCL;
   bool               m_bOpenCL;
   TYPE               m_dNNLoss;
   int                m_iLossSmoothFactor;
   CPositionEncoder*  m_cPositionEncoder;
   bool               m_bPositionEncoder;
   ENUM_LOSS_FUNCTION m_eLossFunction;
   VECTOR             m_adLambda;
   TYPE               m_dLearningRate;
   VECTOR             m_adBeta;

public:
                      CNet(void);
                     ~CNet(void);
   //--- Методы создания объекта
   bool               Create(........);
   //--- Организация работы с OpenCL
   void               UseOpenCL(bool value);
   bool               UseOpenCL(void)          const { return(m_bOpenCL);          }
   bool               InitOpenCL(void);

   //--- Методы работы с позиционным кодированием
   void               UsePositionEncoder(bool value);
   bool               UsePositionEncoder(void) const { return(m_bPositionEncoder); }
   //--- Организация основных алгоритмов работы модели
   bool               FeedForward(........);
   bool               Backpropagation(........);
   bool               UpdateWeights(........);
   bool               GetResults(........);
   void               SetLearningRates(TYPE learning_rate, TYPE beta1 = defBeta1,
                                                           TYPE beta2 = defBeta2);
   //--- Методы функции потерь
   bool               LossFunction(ENUM_LOSS_FUNCTION loss_function,
                           TYPE lambda1 = defLambdaL1, TYPE lambda2 = defLambdaL2);
   ENUM_LOSS_FUNCTION LossFunction(void)       const { return(m_eLossFunction);    }
   ENUM_LOSS_FUNCTION LossFunction(TYPE &lambda1, TYPE &lambda2);
```

```
   TYPE              GetRecentAverageLoss(void) const { return(m_dNNLoss);        }
   void              LossSmoothFactor(int value)   { m_iLossSmoothFactor = value; }
   int               LossSmoothFactor(void)   const { return(m_iLossSmoothFactor);}
   //--- Управление режимом работы модели
   bool              TrainMode(void)          const { return m_bTrainMode;        }
   void              TrainMode(bool mode);
   //--- Методы работы с файлами
   virtual bool      Save(........);
   virtual bool      Load(........);
   //--- Метод идентификации объекта
   virtual int       Type(void)               const { return(defNeuronNet);       }
   //--- Получение указателей на внутренние объекты
   virtual CBufferType* GetGradient(uint layer)     const;
   virtual CBufferType* GetWeights(uint layer)      const;
   virtual CBufferType* GetDeltaWeights(uint layer) const;
  };
```

Можно заметить, что в объявлении ряда методов я оставил многоточие вместо указания параметров. Сейчас мы разберем методы класса и добавим недостающие данные.

Начнем с конструктора класса. В нем инициализируем начальными значениями переменные и создадим экземпляры используемых классов.

```
CNet::CNet(void)     :  m_bTrainMode(false),
                        m_bOpenCL(false),
                        m_bPositionEncoder(false),
                        m_dNNLoss(-1),
                        m_iLossSmoothFactor(defLossSmoothFactor),
                        m_dLearningRate(defLearningRate),
                        m_eLossFunction(LOSS_MSE)
  {
   m_adLambda.Init(2);
   m_adBeta.Init(2);
   m_adLambda[0] = defLambdaL1;
   m_adLambda[1] = defLambdaL2;
   m_adBeta[0]   = defBeta1;
   m_adBeta[1]   = defBeta2;
   m_cLayers     = new CArrayLayers();
   m_cOpenCL     = new CMyOpenCL();
   m_cPositionEncoder = new CPositionEncoder();
  }
```

В деструкторе класса очистим память, удалив экземпляры созданных ранее объектов.

```
CNet::~CNet(void)
  {
   if(!!m_cLayers)
      delete m_cLayers;
   if(!!m_cPositionEncoder)
      delete m_cPositionEncoder;
   if(!!m_cOpenCL)
      delete m_cOpenCL;
  }
```

Рассмотрим метод создания нейронной сети ***Create***. Выше я опустил параметры данного метода, а теперь предлагаю их обсудить.

В предыдущей главе был описан интерфейс передачи структуры нейронной сети в класс. Разумеется, его мы и будем передавать в данный метод. Но достаточно ли только этих данных или нет? С технической стороны вопроса, этих данных вполне достаточно для указания архитектуры нейронной сети. Для указания коэффициентов обучения и функции потерь мы предусмотрели дополнительные методы.

А если посмотреть на вопрос со стороны пользователя: насколько удобно использовать три метода для указания всех необходимых параметров при инициализации нейронной сети? На самом деле это вопрос личных привычек и предпочтений пользователя. Одни предпочитают использовать несколько методов с указанием одного или двух параметров и на каждом шаге контролировать процесс. Другие предпочтут в одной строчке кода «закинуть» все параметры в один метод, один раз проверить результат и идти дальше.

Когда мы работаем непосредственно с заказчиком, мы можем обсудить его предпочтения и сделать продукт удобный именно для него. Но при создании универсального продукта логично постараться удовлетворить предпочтениям всех возможных пользователей. Тем более, что пользователь может выбирать разные варианты в зависимости от поставленной задачи. Поэтому воспользуемся возможностью перегрузки функций и создадим несколько одноименных методов для удовлетворения всех возможных вариантов использования.

Первым создадим метод с минимальным количеством параметров, который будет получать только динамический массив описания архитектуры нейронной сети. В начале метода проверим действительность указателя на полученный в параметрах метода объект. Затем проверим количество нейронных слоев в переданном описании.

Мы уже говорили ранее, что слоев не может быть меньше двух, так как первый входной слой служит для записи исходных данных, а последний слой — для вывода результата работы нейронной сети. В случае получения отрицательного результата хотя бы одной проверки выходим из метода с результатом ***false***.

```
bool CNet::Create(CArrayObj *descriptions)
  {
//--- Блок контролей
   if(!descriptions)
      return false;
//--- Проверяем количество создаваемых слоев
   int total = descriptions.Total();
   if(total < 2)
      return false;
```

После успешного прохождения контролей инициализируем класс для работы с технологией *OpenCL*. В отличие от предыдущих проверок мы не будем возвращать ***false*** при ошибках инициализации. Мы лишь отключим данный функционал и продолжим работу в стандартном режиме. Такой подход реализован для возможности тиражирования готового продукта на различные вычислительные машины без изменения кода программы, что в общем увеличивает возможный круг потенциальных клиентов для распространения конечного продукта.

```
//--- Инициализируем объекты OpenCL
   if(m_bOpenCL)
      m_bOpenCL = InitOpenCL();
   if(!m_cLayers.SetOpencl(m_cOpenCL))
      m_bOpenCL = false;
```

Чтобы все объекты нашей нейронной сети работали в одном контексте *OpenCL*, передадим указатель на экземпляр класса *CMyOpenCL* в массив хранения нейронных слоев. Оттуда в последующем он будет передан каждому нейронному слою.

Затем организуем цикл с числом итераций равным числу слоев нашей сети. В нем поочередно будем перебирать все элементы динамического массива описания нейронных слоев. При этом будем проверять действительность объекта описания каждого слоя, а также соответствие указанных параметров целостности модели. В коде метода можно увидеть проверку специфических параметров различных типов нейронных слоев, с которыми мы познакомимся немного позже.

После этого вызовем метод для создания соответствующего слоя. Стоит отметить, что непосредственно создание нейронного слоя мы поручим методу создания элемента ***CreateElement*** динамического массива хранения нейронных слоев *m_cLayers*.

```
//--- Организовываем цикл для создания нейронных слоев
   for(int i = 0; i < total; i++)
     {
      CLayerDescription *temp = descriptions.At(i);
      if(!temp)
         return false;
      if(i == 0)
        {
         if(temp.type != defNeuronBase)
            return false;
         temp.window = 0;
        }

      else
        {
         CLayerDescription *prev = descriptions.At(i - 1);
         if(temp.window <= 0 || temp.window > prev.count ||
            temp.type == defNeuronBase)
           {
            switch(prev.type)
              {
               case defNeuronConv:
               case defNeuronProof:
                  temp.window = prev.count * prev.window_out;
                  break;
               case defNeuronAttention:
               case defNeuronMHAttention:
                  temp.window = prev.count * prev.window;
                  break;
               case defNeuronGPT:
                  temp.window = prev.window;
                  break;
               default:
                  temp.window = prev.count;
                  break;
              }

            switch(temp.type)
              {
               case defNeuronAttention:
               case defNeuronMHAttention:
               case defNeuronGPT:
                  break;
               default:
                  temp.step = 0;
              }
           }
        }
      if(!m_cLayers.CreateElement(i, temp))
         return false;
     }
```

В заключение метода инициализируем класс позиционного кодирования. Обратите внимание, что сам код каждой позиции остается неизменным на протяжении обучения и использования нейронной сети. Будут меняться элементы, но размеры входного слоя нейронов не меняются. Значит, мы можем сразу при создании сети пересчитать и сохранить код позиции для каждого элемента и в последующем использовать сохраненные значения вместо постоянного пересчета кода.

```
//--- Инициализируем объекты позиционного кодирования
   if(m_bPositionEncoder)
     {
      if(!m_cPositionEncoder)
        {
         m_cPositionEncoder = new CPositionEncoder();
         if(!m_cPositionEncoder)
            m_bPositionEncoder = false;
         return true;
        }
      CLayerDescription *temp = descriptions.At(0);
      if(!m_cPositionEncoder.InitEncoder(temp.count, temp.window))
         UsePositionEncoder(false);
     }
//---
   return true;
  }
```

При организации перегрузок метода ***Create*** мы не будем переписывать весь код, а лишь выполним работу нашего пользователя и осуществим вызов необходимых методов с полученными параметрами. Ниже приведены возможные варианты перегруженного метода.

```
bool CNet::Create(CArrayObj *descriptions,
                  TYPE learning_rate,
                  TYPE beta1,TYPE beta2,
                  ENUM_LOSS_FUNCTION loss_function,
                  TYPE lambda1,TYPE lambda2)
  {
   if(!Create(descriptions))
      return false;
   SetLearningRates(learning_rate,beta1,beta2);
   if(!LossFunction(loss_function,lambda1,lambda2))
      return false;
//---
   return true;
  }

bool CNet::Create(CArrayObj *descriptions,
                  ENUM_LOSS_FUNCTION loss_function,
                  TYPE lambda1,TYPE lambda2)
  {
   if(!Create(descriptions))
      return false;
   if(!LossFunction(loss_function,lambda1,lambda2))
      return false;
//---
   return true;
  }

bool CNet::Create(CArrayObj *descriptions,
                  TYPE learning_rate,
                  TYPE beta1,TYPE beta2)
  {
   if(!Create(descriptions))
      return false;
   SetLearningRates(learning_rate,beta1,beta2);
//---
   return true;
  }
```

При создании перегруженных методов не забудьте объявить все используемые варианты перегрузок метода в описании класса.

Двигаемся дальше. Поговорим о методе прямого прохода ***FeedForward***. В объявлении метода выше опущены параметры. Давайте подумаем, какие данные нам нужны для осуществления прямого прохода. Прежде всего нам нужны исходные данные. Они должны быть переданы в нейронную сеть извне. Добавляем в параметры динамический массив *CBufferType.* Данный класс мы создадим позже — он будет обслуживать все наши буферы данных.

В процессе прямого прохода исходные данные умножаются на весовые коэффициенты, которые хранятся в объектах нейронных слоев. Значит, нейронная сеть их уже знает. Полученные значения проходят через функцию активации. Используемые функции для каждого слоя указаны на стадии создания нейронной сети в описании архитектуры.

Таким образом, для осуществления прямого прохода нам на входе достаточно получить массив исходных данных.

В теле метода проверим действительность указателей на массив исходных данных и первого нейронного слоя нашей сети. Мы не будем создавать отдельный тип нейронного слоя для исходных данных. Вместо этого возьмем базовый полносвязный нейронный слой и запишем полученные исходные данные в буфер выходных (результирующих) значений нейронов. Таким образом, получим унификацию нейронных слоев.

```
bool CNet::FeedForward(const CBufferType *inputs)
  {
//--- Блок контролей
   if(!inputs)
      return false;
   CNeuronBase *InputLayer = m_cLayers.At(0);
   if(!InputLayer)
      return false;
```

Следующим шагом при необходимости проведем позиционирование исходных значений.

```
   CBufferType *Inputs = InputLayer.GetOutputs();
   if(!Inputs)
      return false;
   if(Inputs.Total() != inputs.Total())
      return false;
//--- Переносим исходные данные в нейронный слой
   Inputs.m_mMatrix = inputs.m_mMatrix;
//--- Применяем позиционное кодирование
   if(m_bPositionEncoder && !m_cPositionEncoder.AddEncoder(Inputs))
      return false;
   if(m_bOpenCL)
      Inputs.BufferCreate(m_cOpenCL);
```

На этом этап подготовки исходных данных можно считать завершенным. Перейдем непосредственно к прямому проходу: организуем цикл с последовательным перебором всех нейронных слоев нашей сети от первого до последнего, для каждого слоя вызовем одноименный метод прямого прохода. Обратите внимание, что цикл начинается со слоя с индексом 1. Нейронный слой с записанными исходными данными имеет индекс 0.

Еще один момент, на который следует обратить внимание. В процессе перебора для всех объектов нейронных слоев мы используем один класс *CNeuronBase*. Это наш базовый класс для нейронного слоя. Все остальные классы нейронных слоев будут его наследниками.

В дополнение создадим виртуальный метод *FeedForward*, который будет переопределен во всех других типах нейронных слоев. Такая реализация позволяет нам использовать базовый класс нейронного слоя и вызывать виртуальный метод прямого прохода. Работу по распределению и использованию метода прямого прохода конкретного типа нейронов выполнит за нас компилятор и система.

```
//--- Организовываем цикл с полным перебором всех нейронных слоев
//--- и вызовом метода прямого прохода для каждого из них
   CNeuronBase *PrevLayer = InputLayer;
   int total = m_cLayers.Total();
   for(int i = 1; i < total; i++)
     {
      CNeuronBase *Layer = m_cLayers.At(i);
      if(!Layer)
         return false;
      if(!Layer.FeedForward(PrevLayer))
         return false;
      PrevLayer = Layer;
     }
```

Здесь надо обратить внимание, что при использовании технологии *OpenCL* во время отправки кернела на выполнение осуществляется его постановка в очередь. И чтобы «подтолкнуть» его выполнение, необходимо инициировать считывание результатов операции. Ранее мы уже обсуждали необходимость минимизации процессов обмена данными между ОЗУ и контекстом *OpenCL*. Поэтому мы не будем считывать данные после постановки каждого кернела в очередь. Вместо этого отправим всю цепочку операций в очередь и только после завершения цикла перебора всех нейронных слоев запросим результат операций последнего нейронного слоя. Так как у нас данные передаются последовательно от одного слоя к другому, то потянется и вся очередь операций. Но не забывайте, что загрузка данных необходима только при использовании технологии *OpenCL*.

```
   if(m_bOpenCL)
      if(!PrevLayer.GetOutputs().BufferRead())
         return false;
//---
   return true;
  }
```

При прямом проходе мы получили некие расчетные данные. На необученной нейронной сети полученный результат будет довольно случайным. Мы же хотим, чтобы наша нейронная сеть выдавала результаты, максимально приближенные к реальным. И чтобы приблизиться к ним нам предстоит обучить нейронную сеть. Процесс обучения с учителем строится на итеративном подходе с постепенным подстраиванием весовых коэффициентов под правильные ответы. Как мы уже говорили ранее, этот процесс состоит из двух этапов: прямого и обратного прохода. Метод прямого прохода мы уже написали. Посмотрим на метод обратного прохода ***Backpropagation***.

Выше, при описании класса, я также опустил параметры этого метода. Посмотрите еще раз на алгоритм обратного прохода. Легко заметить, что из внешней системы нам нужны лишь правильные ответы. Поэтому в параметры метода добавим динамический массив правильных ответов. Но на входе метода мы получим лишь эталонные значения для выходного нейронного слоя. Поэтому нам предстоит посчитать градиент ошибки для каждого нейрона нашей сети. Исключение составляют лишь нейроны входного слоя — их значения подаются внешней системой и не зависят от состояния нейронной сети. Поэтому вычисление градиента ошибки исходных данных является излишней работой, не имеющей практической ценности и логического смысла.

В начале метода, как всегда, сделаем проверку данных для работы метода. В этом блоке мы проверим действительность полученного указателя на динамический массив эталонных значений и сравним размеры буфера результатов с полученным вектором эталонных значений.

После этого посчитаем значение функции потерь. Сам расчет функции потерь скрыт в стандартных матричных операциях *MQL5*. Алгоритм расчета значения функции был показан при рассмотрении возможных вариантов функции потерь. Проверим полученное значение функции потерь и посчитаем сглаженную ошибку за весь период обучения.

```
bool CNet::Backpropagation(CBufferType *target)
  {
//--- Блок контролей
   if(!target)
      return false;
   int total = m_cLayers.Total();
   CNeuronBase *Output = m_cLayers.At(total - 1);
   if(!Output ||Output.Total()!=target.Total())
      return false;
//--- Расчет значения функции потерь
   TYPE loss = Output.GetOutputs().m_mMatrix.Loss(target.m_mMatrix,
                                                  m_eLossFunction);

   if(loss == FLT_MAX)
      return false;
   m_dNNLoss = (m_dNNLoss < 0 ? loss :
                m_dNNLoss + (loss - m_dNNLoss) / m_iLossSmoothFactor);
```

В следующем блоке нашего метода обратного прохода мы доведем градиент ошибки до каждого нейрона нашей сети. Для этого сначала посчитаем градиент ошибки на выходном слое, а потом организуем обратный цикл. В переборе от выхода нейронной сети к ее входу для каждого нейронного слоя будем вызывать метод расчета градиента. О различиях в алгоритмах расчета градиента выходного и скрытого слоя нейронной сети мы поговорим чуть позже, при рассмотрении полносвязного нейронного слоя.

Тут же мы посчитаем, как должны измениться весовые коэффициенты нашей нейронной сети, чтобы она выдавала правильные результаты для текущего набора исходных данных. В последовательном переборе нейронных слоев для каждого будем вызывать метод расчета дельт.

```
//--- Расчет градиента ошибки на выходе нейронной сети
   CBufferType* grad = Output.GetGradients();
   grad.m_mMatrix = target.m_mMatrix;
   if(m_cOpenCL)
     {
      if(!grad.BufferWrite())
         return false;
     }
   if(!Output.CalcOutputGradient(grad, m_eLossFunction))
      return false;
//--- Организовываем цикл с перебором всех нейронных слоев в обратном порядке
   for(int i = total - 2; i >= 0; i--)
     {
      CNeuronBase *temp = m_cLayers.At(i);
      if(!temp)
         return false;
      //--- Вызываем метода распределения градиента ошибки через скрытый слой
      if(!Output.CalcHiddenGradient(temp))
         return false;
      //--- Вызываем метод распределения градиента ошибки до матрицы весов
      if(!Output.CalcDeltaWeights(temp, i == 0))
         return false;
      Output = temp;
     }
```

Аналогично, как и при прямом проходе, в случае использования технологии *OpenCL* нам необходимо загрузить результаты выполнения операций последнего кернела в очереди.

```
   if(m_cOpenCL)
     {
      for(int i = 1; i < m_cLayers.Total(); i++)
        {
         Output = m_cLayers.At(i);
         if(!Output.GetDeltaWeights() || !Output.GetDeltaWeights().BufferRead())
            continue;
         break;
        }
     }
//---
   return true;
  }
```

Цель обучения нейронной сети — не найти отклонения, а настроить ее для максимального правдоподобия результатов. Нейронная сеть настраивается путем подбора правильных весовых коэффициентов. Следовательно, после расчета дельт мы должны обновить весовые коэффициенты. По указанным выше причинам обновление весовых коэффициентов я вынес в отдельный метод ***UpdateWeights***.

При объявлении метода в описании класса параметры не указаны. Давайте подумаем: дельты для обновления весовых коэффициентов мы уже посчитали, а коэффициенты обучения и регуляризации заданы при инициализации нейронной сети. На первый взгляд, у нас есть все необходимое для обновления весов. Но посмотрите на дельты. При каждой итерации мы их будем складывать. Если для обновления коэффициентов использовать пакет некоего размера, то велика

вероятность получения завышенной дельты. В такой ситуации логично использовать среднюю дельту. Чтобы получить среднюю из суммы дельт пакета, достаточно разделить имеющуюся сумму на размер пакета. Конечно, с точки зрения математики размер пакета можно учесть в коэффициенте обучения. Если мы заблаговременно разделим коэффициент обучения на размер пакета, то итоговый результат не изменится.

$$\alpha \frac{\Delta}{Batchsize} = \frac{\alpha}{Batchsize}\Delta$$

Но это ручной контроль и, как всегда, вопрос предпочтений пользователя. Мы же дадим возможность использовать оба варианта: добавим в метод параметр для указания размера пакета и установим ему значение по умолчанию равным единице. Таким образом, пользователь может указать размер пакета в параметрах метода, а может вызвать метод без указания параметров. Тогда размер пакета приравняется к значению по умолчанию, и дельта будет скорректирована только на обучающий коэффициент.

Алгоритм метода довольно прост. Сначала проверим указанный размер пакета — он обязательно должен быть целым значением больше нуля. Далее организуем цикл по перебору всех нейронных слоев нашей сети с вызовом одноименного метода для каждого слоя. Сам же процесс обновления весов будет осуществляться на уровне нейронного слоя.

```
bool CNet::UpdateWeights(uint batch_size = 1)
  {
//--- Блок контролей
   if(batch_size <= 0)
      return false;
//--- Организовываем цикл перебора всех скрытых слоев
   int total = m_cLayers.Total();
   for(int i = 1; i < total; i++)
     {
      //--- Проверяем действительность указателя на объект нейронного слоя
      CNeuronBase *temp = m_cLayers.At(i);
      if(!temp)
         return false;
      //--- Вызываем метод обновления матрицы весов внутреннего слоя
      if(!temp.UpdateWeights(batch_size, m_dLearningRate, m_adBeta, m_adLambda))
         return false;
     }
//---
   return true;
  }
```

И конечно, у пользователя должна быть возможность получить результаты работы нейронной сети после выполнения прямого прохода. Возложим данный функционал на метод *GetResult*.

Какие данные извне должен получить метод? Рассуждая логически, функция должна не получить, а вернуть данные во внешнюю программу. Но мы не знаем, какими будут эти данные и в каком количестве. Зная возможные варианты функций активации нейронов, логично предположить, что на выходе каждого нейрона будет некое число. Количество таких значений будет равно количеству нейронов в выходном слое. Соответственно, оно будет известно на стадии генерации нейронной сети. Логичным выходом из сложившейся ситуации будет динамический массив соответствующего типа. Напомню, что выше для передачи данных в нашу модель мы использовали класс буфера данных *CBufferType*. Здесь мы воспользуемся аналогичным

объектом. Таким образом для обмена данными между основной программой и моделью мы всегда будем использовать один класс динамического массива.

В теле метода сначала получаем указатель на массив значений нейронов выходного слоя и проверяем действительность данного указателя. Затем проверим действительность указателя на динамический массив для записи результатов. Напомню, ссылку на последний массив мы получили в параметрах метода от внешней программы. Если указатель недействительный, то инициируем создания нового экземпляра класса буфера данных. После успешного создания нового буфера копируем в него значения с нейронов выходного слоя и выходим из метода.

```
bool CNet::GetResults(CBufferType *&result)
  {
   int total = m_cLayers.Total();
   CNeuronBase *temp = m_cLayers.At(total - 1);
   if(!temp)
      return false;
   CBufferType *output = temp.GetOutputs();
   if(!output)
      return false;
   if(!result)
     {
      if(!(result = new CBufferType()))
         return false;
     }
   if(m_cOpenCL)
      if(!output.BufferRead())
         return false;
   result.m_mMatrix = output.m_mMatrix;
//---
   return true;
  }
```

Нужно сказать, что в зависимости от сложности поставленной задачи нейронные сети могут сильно отличаться по сложности архитектуры и количеству синаптических связей. А от сложности сети сильно зависит время ее обучения. Каждый раз заново обучать нейронную сеть неэффективно, а в большинстве случаев просто невозможно. Поэтому однажды обученную нейронную сеть нужно сохранить и при следующем запуске загрузить все коэффициенты из файла. И лишь потом при необходимости можно дообучить нейронную сеть под текущие реалии.

За сохранение обученной нейронной сети отвечает метод ***Save***. Этот виртуальный метод создан в базовом классе *CObject*, он переопределяется в каждом новом классе. Я осознанно не стал сразу переписывать параметры метода из родительского класса. Дело в том, что там предполагается получение в параметрах хендла файла для записи объекта. Т.е. файл предварительно должен быть открыт во внешней программе, а после сохранения данных внешняя программа закрывает файл.

Иными словами, контроль за открытием и закрытием файла выносится из класса и возлагается на вызывающую программу. Такой подход удобен, когда объект является частью некоего большого проекта и позволяет записать последовательно все объекты проекта в один общий файл. И мы обязательно этим воспользуемся при сохранении данных объектов, составляющих нашу нейронную сеть.

Но когда мы говорим о верхнем уровне нашей программы, хотелось бы иметь один метод для сохранения всего проекта. Этот метод должен взять на себя работу по открытию и закрытию файла, перебор и сохранение всей информации, минимально необходимой для восстановления всей нейронной сети из файла. При этом мы не можем исключать возможность, что нейронная сеть будет лишь частью чего-то большего.

С учетом вышеизложенных мыслей мы создадим два одноименных метода: один будет в параметрах получать хендл файла по аналогии с методом родительского класса, а второму в параметрах будем передавать имя файла для записи данных.

Теперь давайте подумаем, какая минимальная информация нам нужна для полного восстановления обученной нейронной сети. Конечно, нам нужна архитектура сети, количество слоев и нейронов в них. Кроме того, нужны все весовые коэффициенты. Для этого нам потребуется сохранить весь массив нейронных слоев.

Но надо понимать, что обученная нейронная сеть будет корректно работать только в той среде, для которой она обучалась. Поэтому сохраним информацию о функции потерь и кодировке позиций.

Информацию об инструменте и таймфрейме я предлагаю записать в названии файла. Это позволит в последующем советнику быстро определить наличие предварительно обученной сети на диске. К тому же, достаточно будет изменить название файла, чтобы перенести и протестировать предварительно обученную нейронную сеть на другой инструмент или таймфрейм. В большинстве случаев дообучить нейронную сеть будет легче, чем обучать ее со случайных весовых коэффициентов.

Чтобы понимать, насколько обучена сохраненная в файле нейронная сеть, добавим последнее среднее значение функции потерь и коэффициент усреднения. Для легкого продолжения обучения сохраним параметры обучения и регуляризации и для полноты картины добавим флаг использования *OpenCL*.

Посмотрим на алгоритм метода с хендлом файла в параметрах. В начале метода проверим действительность полученного хендла файла для записи данных и указателей на экземпляры объектов функции потерь и динамического массива нейронных слоев.

```
bool CNet::Save(const int file_handle)
  {
   if(file_handle == INVALID_HANDLE ||
      !m_cLayers)
      return false;
```

Далее сохраним указанные выше параметры.

```
//--- Сохраняем константы
   if(!FileWriteInteger(file_handle, (int)m_bOpenCL) ||
      !FileWriteDouble(file_handle, m_dNNLoss) ||
      !FileWriteInteger(file_handle, m_iLossSmoothFactor) ||
      !FileWriteInteger(file_handle, (int)m_bPositionEncoder) ||
      !FileWriteDouble(file_handle, (double)m_dLearningRate) ||
      !FileWriteDouble(file_handle, (double)m_adBeta[0]) ||
      !FileWriteDouble(file_handle, (double)m_adBeta[1]) ||
      !FileWriteDouble(file_handle, (double)m_adLambda[0]) ||
      !FileWriteDouble(file_handle, (double)m_adLambda[1]) ||
      !FileWriteInteger(file_handle, (int)m_eLossFunction))
      return false;
```

Проверим флаг использования позиционного кодирования входящей последовательности и при необходимости вызовем метод сохранения экземпляра класса *CPositionEncoder*. В заключение метода вызовем метод сохранения динамического массива нейронных слоев. Детальнее с вызываемыми методами будем знакомиться по мере разбора содержащих их классов.

```
//--- Сохраняем объект позиционного кодирования при необходимости
   if(m_bPositionEncoder)
     {
      if(!m_cPositionEncoder ||
         !m_cPositionEncoder.Save(file_handle))
         return false;
     }
//--- Вызываем метод сохранения данных динамического массива нейронных слоев
   return m_cLayers.Save(file_handle);
  }
```

Алгоритм метода с именем файла в параметрах будет немного проще. Мы не будем переписывать весь алгоритм сохранения данных. Мы лишь организуем открытие файла для записи информации и передадим полученный хендл в рассмотренный выше метод. После завершения работы метода закроем файл.

Обратите внимание, что при получении пустого наименования файла в параметрах мы подменяем его на заданное по умолчанию имя файла и далее выполняем метод в стандартном режиме.

Также после выполнения функции открытия файла нам следовало бы проконтролировать успешность выполнения операции, проверив полученный хендл. Я намеренно исключил этот шаг, так как это первая операция в рассмотренном выше методе *Save*, а выполнение дважды одной операции только замедлит процесс.

```
bool CNet::Save(string file_name = NULL)
  {
   if(file_name == NULL || file_name == "")
      file_name = defFileName;
//---
   int handle = FileOpen(file_name, FILE_WRITE | FILE_BIN);
//---
   bool result = Save(handle);
   FileClose(handle);
//---
   return result;
  }
```

Для обратной операции загрузки данных нейронной сети из файла создадим два аналогичных метода ***Load*** с хендлом и именем файла в параметрах. И если алгоритм метода загрузки данных с именем файла в параметрах полностью идентичен соответствующему методу сохранения данных, то алгоритм второго метода немного усложняется за счет операций инициализации объектов.

В начале метода, как и при сохранении, поверяем действительность полученного хендла файла для загрузки данных.

```
bool CNet::Load(const int file_handle)
  {
   if(file_handle == INVALID_HANDLE)
      return false;
```

Затем загружаем все ранее сохраненные параметры нейронной сети. При этом следим, чтобы последовательность считывания данных строго соответствовала последовательности их записи.

```
//--- Считываем константы
   m_bOpenCL = (bool)FileReadInteger(file_handle);
   m_dNNLoss = FileReadDouble(file_handle);
   m_iLossSmoothFactor = FileReadInteger(file_handle);
   m_bPositionEncoder = (bool)FileReadInteger(file_handle);
   m_dLearningRate = (TYPE)FileReadDouble(file_handle);
   m_adBeta[0] = (TYPE)FileReadDouble(file_handle);
   m_adBeta[1] = (TYPE)FileReadDouble(file_handle);
   m_adLambda[0] = (TYPE)FileReadDouble(file_handle);
   m_adLambda[1] = (TYPE)FileReadDouble(file_handle);
   m_eLossFunction = (ENUM_LOSS_FUNCTION) FileReadInteger(file_handle);
```

Обратите внимание, что при сохранении данных объект позиционного кодирования мы записывали в файл только при включенной функции. Следовательно, сначала мы проверяем, была ли включена функция при сохранении данных, и только при необходимости запускаем процесс считывания метода позиционного кодирования. Предварительно проверяем наличие соответствующего созданного объекта. И если он до этого не был создан, то перед загрузкой данных инициируем создание экземпляра объекта.

```
//--- Загружаем объект позиционного кодирования
   if(m_bPositionEncoder)
     {
      if(!m_cPositionEncoder)
        {
         m_cPositionEncoder = new CPositionEncoder();
         if(!m_cPositionEncoder)
            return false;
        }
      if(!m_cPositionEncoder.Load(file_handle))
         return false;
     }
```

Для инициализации объекта работы с контекстом *OpenCL* мы не будем повторять весь код инициализации. Вместо этого воспользуемся соответствующим методом. Нам лишь надо осуществить его вызов и проконтролировать результат выполнения операций.

```
//--- Инициализируем объект работы с OpenCL
   if(m_bOpenCL)
     {
      if(!InitOpenCL())
         m_bOpenCL = false;
     }
   else
      if(!!m_cOpenCL)
        {
         m_cOpenCL.Shutdown();
         delete m_cOpenCL;
        }
```

Далее нам остается загрузить непосредственно нейронные слои модели и их параметры. Для загрузки данной информации нам было бы достаточно вызвать метод загрузки динамического массива нейронных слоев. Но прежде чем обратиться к методу класса, нам необходимо убедиться в действительности указателя на экземпляр класса. В противном случае мы рискуем получить критическую ошибку выполнения программы. Поэтому мы проверяем действительность указателя и при необходимости создаем новый экземпляр объекта динамического массива. Тут же мы передаем в объект действительный указатель объекта работы с контекстом *OpenCL*. Лишь после выполнения подготовительной работы вызываем метод загрузки динамического массива нейронных слоев.

```
//--- Инициализируем и загружаем данные динамического массива нейронных слоев
   if(!m_cLayers)
     {
      m_cLayers = new CArrayLayers();
      if(!m_cLayers)
         return false;
     }
   if(m_bOpenCL)
      m_cLayers.SetOpencl(m_cOpenCL);
//---
   return m_cLayers.Load(file_handle);
  }
```

Наверное, здесь надо пояснить, почему мы выполняем загрузку только динамического массива, а не всех нейронных слоев. Дело в том, что наш динамический массив нейронных слоев является контейнером, содержащим указатели на все объекты нейронных слоев модели. При сохранении массива были последовательно сохранены все нейронные слои, и теперь, при загрузке данных, также последовательно будут созданы объекты с сохранением указателей в массиве. Детальнее с этим механизмом мы познакомимся при рассмотрении методов данного класса.

Итак, мы разобрали основные методы нашего класса нейронной сети. В итоге, с учетом всего выше сказанного, его финальная структура примет нижеследующий вид.

```
class CNet  : public CObject
  {
protected:
   bool               m_bTrainMode;
   CArrayLayers*      m_cLayers;
   CMyOpenCL*         m_cOpenCL;
   bool               m_bOpenCL;
   TYPE               m_dNNLoss;
   int                m_iLossSmoothFactor;
   CPositionEncoder*  m_cPositionEncoder;
   bool               m_bPositionEncoder;
   ENUM_LOSS_FUNCTION m_eLossFunction;
   VECTOR             m_adLambda;
   TYPE               m_dLearningRate;
   VECTOR             m_adBeta;

public:
                      CNet(void);
                     ~CNet(void);
   //--- Методы создания объекта
   bool               Create(CArrayObj *descriptions);
   bool               Create(CArrayObj *descriptions, TYPE learning_rate,
                                                      TYPE beta1, TYPE beta2);
   bool               Create(CArrayObj *descriptions,
                ENUM_LOSS_FUNCTION loss_function, TYPE lambda1, TYPE lambda2);
   bool               Create(CArrayObj *descriptions, TYPE learning_rate,
                             TYPE beta1, TYPE beta2,
                ENUM_LOSS_FUNCTION loss_function, TYPE lambda1, TYPE lambda2);

   //--- Организация работы с OpenCL
   void               UseOpenCL(bool value);
   bool               UseOpenCL(void)          const { return(m_bOpenCL);         }
   bool               InitOpenCL(void);
   //--- Методы работы с позиционным кодированием
   void               UsePositionEncoder(bool value);
   bool               UsePositionEncoder(void) const { return(m_bPositionEncoder);}
   //--- Организация основных алгоритмов работы модели
   bool               FeedForward(const CBufferType *inputs);
   bool               Backpropagation(CBufferType *target);
   bool               UpdateWeights(uint batch_size = 1);
   bool               GetResults(CBufferType *&result);
   void               SetLearningRates(TYPE learning_rate, TYPE beta1 = defBeta1,
                                                           TYPE beta2 = defBeta2);
   //--- Методы функции потерь
   bool               LossFunction(ENUM_LOSS_FUNCTION loss_function,
                           TYPE lambda1 = defLambdaL1, TYPE lambda2 = defLambdaL2);
   ENUM_LOSS_FUNCTION LossFunction(void)       const { return(m_eLossFunction);    }
   ENUM_LOSS_FUNCTION LossFunction(TYPE &lambda1, TYPE &lambda2);
```

```
   TYPE              GetRecentAverageLoss(void) const { return(m_dNNLoss);        }
   void              LossSmoothFactor(int value)    { m_iLossSmoothFactor = value;}
   int               LossSmoothFactor(void)   const { return(m_iLossSmoothFactor);}
   //--- Управление режимом работы модели
   bool              TrainMode(void)          const { return m_bTrainMode;        }
   void              TrainMode(bool mode);
   //--- Методы работы с файлами
   virtual bool      Save(string file_name = NULL);
   virtual bool      Save(const int file_handle);
   virtual bool      Load(string file_name = NULL, bool common = false);
   virtual bool      Load(const int file_handle);
   //--- Метод идентификации объекта
   virtual int       Type(void)               const { return(defNeuronNet);      }
   //--- Получение указателей на внутренние объекты
   virtual CBufferType* GetGradient(uint layer)     const;
   virtual CBufferType* GetWeights(uint layer)      const;
   virtual CBufferType* GetDeltaWeights(uint layer) const;
  };
```

3.4.4 Динамический массив хранения нейронных слоев

Стоит несколько слов сказать о динамическом массиве хранения нейронных слоев ***CArrayLayers***. Как уже было озвучено ранее, он создан на основе стандартного класса массива объектов ***CArrayObj***.

В целом функционал родительского класса практически полностью покрывает наши требования к динамическому массиву. При рассмотрении исходного кода родительского класса в нем можно найти весь функционал работы динамического массива и обращения к его элементам. Также реализованы и методы для работы с файлами (записи и чтения массива). За это отдельное спасибо команде *MetaQuotes*.

При детальном рассмотрении алгоритма метода чтения массива из файла ***Load*** наше внимание обращает на себя метод создания нового элемента ***CreateElement***.

В предыдущем разделе, при рассмотрении метода чтения нейронной сети из файла, перед чтением данных мы создавали объект соответствующего класса. Указанный метод выполняет аналогичный функционал, но он не реализован в родительском классе. Это вполне понятно и объяснимо, ведь создатели класса не знали, для хранения каких объектов будет использоваться их массив, и не могли создать метод с генерацией неизвестного класса, поэтому оставили виртуальный метод для переопределения в пользовательском классе.

И мы, как потребители их продукта, создаем свой класс динамического массива, наследуя основной функционал от родительского класса. При этом переопределяем метод создания нового элемента массива.

```
class CArrayLayers   :  public CArrayObj
  {
protected:
   CMyOpenCL*        m_cOpenCL;
   int               m_iFileHandle;
public:
                     CArrayLayers(void) : m_cOpenCL(NULL),
                                          m_iFileHandle(INVALID_HANDLE)
                     { }
                    ~CArrayLayers(void) { };
   //---
   virtual bool      SetOpencl(CMyOpenCL *opencl);
   virtual bool      Load(const int file_handle) override;
   //--- method creating an element of array
   virtual bool      CreateElement(const int index) override;
   virtual bool      CreateElement(const int index,
                                   CLayerDescription* description);
   //--- method identifying the object
   virtual int       Type(void) override const { return(defArrayLayers); }
  };
```

Следует обратить внимание еще на один момент. Чтобы из метода родительского класса вызывался наш переопределенный метод, его определение должно полностью соответствовать определению метода родительского класса, включая параметры и возвращаемое значение. В этом, конечно, нет ничего сложного, но перед нами становится тот же вопрос, который стоял перед командой создателей родительского класса: какой объект создавать?

Мы знаем, что это будет объект нейронного слоя, но не знаем, какого типа. Мы можем сохранить тип необходимого нейронного слоя в файл перед записью содержимого самого объекта, но как нам прочитать его из файла, если метод не получает хендл файла для загрузки данных?

В то же время хендл файла мы передаем при вызове метода загрузки данных *Load*. Очевидно, нам надо переопределить и метод загрузки. Но мне бы не хотелось переписывать весь метод. Поэтому я добавил переменную m_iFileHandle, в которую при вызове метода Load сохраняю хендл файла загрузки данных. Затем вызываю аналогичный метод родительского класса.

```
bool CArrayLayers::Load(const int file_handle)
  {
   m_iFileHandle = file_handle;
   return CArrayObj::Load(file_handle);
  }
```

А теперь рассмотрим непосредственно сам метод создания нового нейронного слоя в динамическом массиве. В параметрах методу передается индекс создаваемого элемента. В начале метода мы проверяем, чтобы полученный индекс не был отрицательным, ведь индекс элемента динамического массива не может быть меньше нуля. Также проверим сохраненный хендл файла для загрузки — без него мы не сможем угадать тип создаваемого элемента.

Далее зарезервируем элемент в нашем массиве, прочитаем из файла тип создаваемого элемента и создадим экземпляр нужного нам типа. Не забудем проверить результат создания нового объекта, передать в новый элемент указатель на объект *OpenCL* и сохранить указатель на новый нейронный слой в наш массив. В заключение позаботимся, чтобы индекс нового элемента не превышал максимальное число элементов в массиве.

```
bool CArrayLayers::CreateElement(const int index)
  {
//--- блок проверки исходных данных
   if(index < 0 || m_iFileHandle==INVALID_HANDLE)
      return false;
//--- резервирование элемента массива под новый объект
   if(!Reserve(index + 1))
      return false;
//--- считываем из файла тип нужного объекта и создаем соответствующий нейронный слой
   CNeuronBase *temp = NULL;
   int type = FileReadInteger(m_iFileHandle);
   switch(type)
     {
      case defNeuronBase:
         temp = new CNeuronBase();
         break;
      case defNeuronConv:
         temp = new CNeuronConv();
         break;
      case defNeuronProof:
         temp = new CNeuronProof();
         break;
      case defNeuronLSTM:
         temp = new CNeuronLSTM();
         break;
      case defNeuronAttention:
         temp = new CNeuronAttention();
         break;
      case defNeuronMHAttention:
         temp = new CNeuronMHAttention();
         break;
      case defNeuronGPT:
         temp = new CNeuronGPT();
         break;
      case defNeuronDropout:
         temp = new CNeuronDropout();
         break;
      case defNeuronBatchNorm:
         temp = new CNeuronBatchNorm();
         break;
      default:
         return false;
     }
//--- контроль создания нового объекта
   if(!temp)
      return false;
```

```
//--- добавляем указатель на созданный объект в массив
   if(m_data[index])
      delete m_data[index];

   temp.SetOpenCL(m_cOpenCL);
   m_data[index] = temp;
//---
   return true;
  }
```

Ну и раз был создан новый класс, было решено добавить в него еще пару методов. Первое, что я добавил, — это аналогичный метод генерации нового элемента. Отличие в том, что новый метод создает новый слой по описанию, полученному в параметрах метода. Алгоритм метода практически полностью повторяет вышеприведенный, за исключением некоторых деталей.

```
bool CArrayLayers::CreateElement(const int index, CLayerDescription *desc)
  {
//--- блок проверки исходных данных
   if(index < 0 || !desc)
      return false;
//--- резервирование элемента массива под новый объект
   if(!Reserve(index + 1))
      return false;
//--- создаем соответствующий нейронный слой
   CNeuronBase *temp = NULL;
   switch(desc.type)
     {
      case defNeuronBase:
         temp = new CNeuronBase();
         break;
      case defNeuronConv:
         temp = new CNeuronConv();
         break;
      case defNeuronProof:
         temp = new CNeuronProof();
         break;
      case defNeuronLSTM:
         temp = new CNeuronLSTM();
         break;
      case defNeuronAttention:
         temp = new CNeuronAttention();
         break;
      case defNeuronMHAttention:
         temp = new CNeuronMHAttention();
         break;
      case defNeuronGPT:
         temp = new CNeuronGPT();
         break;
      case defNeuronDropout:
         temp = new CNeuronDropout();
         break;
      case defNeuronBatchNorm:
         temp = new CNeuronBatchNorm();
         break;
      default:
         return false;
     }
//--- контроль создания нового объекта
   if(!temp)
      return false;
```

```
//--- добавляем указатель на созданный объект в массив
   if(!temp.Init(desc))
      return false;
   if(m_data[index])
      delete m_data[index];
   temp.SetOpenCL(m_cOpenCL);
   m_data[index] = temp;
   m_data_total  = fmax(m_data_total, index + 1);
//---
   return true;
  }
```

Второй добавленный метод отвечает за передачу указателя на объект *OpenCL* во все ранее созданные слои нашей нейронной сети, ведь решение об использовании этой технологии может быть принято как до генерации нейронной сети, так и после. К примеру, нейронная сеть может быть создана и проверена на работоспособность без использования технологии OpenCL. Далее для ускорения процесса обучения может быть задействована данная технология.

Алгоритм метода довольно прост. Вначале мы проверяем, был ли ранее задан указатель, и при необходимости удаляем старый объект. Затем сохраняем новый указатель и запускаем цикл перебора элементов динамического массива. При этом каждому элементу массива передадим новый указатель на объект *OpenCL*.

```
bool CArrayLayers::SetOpencl(CMyOpenCL *opencl)
  {
//--- блок проверки исходных данных
   if(m_cOpenCL)
      delete m_cOpenCL;

   m_cOpenCL = opencl;
//--- передача указателя во все элементы массива
   for(int i = 0; i < m_data_total; i++)
     {
      if(!m_data[i])
         return false;
      if(!((CNeuronBase *)m_data[i]).SetOpenCL(m_cOpenCL))
         return false;
     }
//---
   return(!!m_cOpenCL);
  }
```

3.5 Описание структуры скрипта Python

Python — интерпретируемый язык программирования с минималистичным синтаксисом. Такой синтаксис позволяет быстро писать небольшие блоки кода и сразу проверять их работоспособность. Поэтому Python позволяет сосредоточиться на решении задачи, а не на программировании. Наверное, именно благодаря этой особенности Python получил такую популярность.

Несмотря на то, что интерпретируемые языки программирования работают медленнее компилируемых, в настоящее время Python стал наиболее популярным языков программирования

для создания и проведения экспериментов с нейронными сетями. Вопрос скорости выполнения решается с помощью использования различных библиотек, написанных, в том числе, и на компилируемых языках программирования. Благо Python имеет возможность легко расширяться и подключать библиотеки, написанные практически на всех доступных языках программирования.

Мы с вами тоже не будем выстраивать сложные алгоритмы, а воспользуемся готовыми решениями, среди которых есть библиотеки как для построения нейронных сетей, так и для трейдинга. Давайте для начала познакомимся с некоторыми из них.

Модуль ***os*** содержит функции для работы с операционной системой. Использование данной библиотеки позволяет создавать кросс-платформенные приложения, так как работа функций этого модуля не зависит от установленной операционной системы. Приведем лишь некоторые функции библиотеки *os*:

- *os.name* — возвращает имя операционной системы. В результате выполнения функции возможны следующие варианты: 'posix', 'nt', 'mac', 'os2', 'ce', 'java'.
- *os.environ* — функция для работы с переменными окружения, позволяющая изменять, добавлять и удалять переменные окружения.
- *os.path* — содержит целый ряд функций для работы с путями файлов и директорий.

Модуль ***Pandas*** представляет собой библиотеку для обработки и анализа данных. Библиотека предоставляет специальные структуры данных и операции для обработки числовых таблиц и временных рядов. Она позволяет проводить анализ и моделирование данных без использования специализированных для статистической обработки языков программирования, таких как *R* и *Octave*.

Пакет предназначен для очистки и первичной оценки данных по общим статистическим показателям. Он позволяет вычислять среднее значение, квантили и т. д. В то же время пакет нельзя назвать статистическим в полном смысле, однако создаваемые им наборы данных типов *DataFrame* и *Series* применяются в качестве входных в большинстве модулей анализа данных и машинного обучения *SciPy*, *Scikit-Learn* и других.

Объект *DataFrame* создан в библиотеке *Pandas*. Он предназначен для работы с индексированными массивами двумерных данных.

Кроме того, библиотека предоставляет:

- инструменты для обмена данными между структурами в памяти и файлами различных форматов;
- встроенные средства совмещения данных и способы обработки отсутствующей информации;
- переформатирование наборов данных, в том числе создание сводных таблиц;
- расширенные возможности индексирования и выборки из больших наборов данных;
- возможности группировки позволяют выполнять трехэтапные операции типа «разделение, изменение, объединение»;
- слияние и объединение различных наборов данных.

Библиотека дает возможность создания иерархического индексирования, что позволяет работать с данными высокой размерности в структурах меньшей размерности. Функции для работы с временными рядами позволяют формировать временные периоды и изменять интервалы. Библиотека оптимизирована для высокой производительности, наиболее важные части кода написаны на *Cython* и *C*.

Еще одна библиотека для работы с многомерными массивами — ***NumPy*** — представляет собой библиотеки с открытым исходным кодом. Основными возможностями данного модуля являются поддержка многомерных массивов (включая матрицы) и высокоуровневых математических функций, предназначенных для работы с многомерными массивами.

В библиотеке *NumPy* реализованы вычислительные алгоритмы в виде функций и операторов, которые оптимизированы для работы с многомерными массивами. Библиотека представляет возможность проведения векторных операций над данными. При этом все функции написаны на *C* и оптимизированы для максимальной производительности. В результате любой алгоритм, который может быть выражен в виде последовательности операций над массивами (матрицами) и реализованный с использованием *NumPy*, работает так же быстро, как эквивалентный код, выполняемый в *MATLAB*.

При этом *NumPy* можно рассматривать в качестве альтернативы использования *MATLAB*. Оба языка интерпретируемые и позволяют выполнять операции над массивами.

NumPy часто используется в качестве базы для работы с многомерными массивами в других библиотеках. В том числе представленный выше *Pandas* также использует библиотеку *NumPy* для низкоуровневых операций с массивами.

Модуль ***Matplotlib*** — это комплексная библиотека для создания статических, анимированных и интерактивных визуализаций. Она с легкостью позволяет визуализировать большие объемы данных.

Для создания моделей нейронных сетей мы будем использовать библиотеку ***TensorFlow***. Это комплексная платформа с открытым исходным кодом для машинного обучения. Имеет гибкую экосистему инструментов, библиотек и ресурсов сообщества, которая позволяет исследователям продвигать новейшие достижения в области машинного обучения, а разработчикам легко создавать и развертывать приложения на основе машинного обучения.

Библиотека позволяет создавать и обучать модели машинного обучения с помощью интуитивно понятных высокоуровневых API с активным исполнением, таких как *Keras*. Это обеспечивает немедленную интеграцию модели и облегчает ее отладку.

И конечно, для интеграции с терминалом *MetaTrader 5* мы будем использовать одноименную библиотеку *MetaTrader5*. Она предоставляет ряд функций для обмена данными с терминалом, в том числе функции получения рыночной информации и совершения торговых операций.

Для технического анализа данных можно воспользоваться библиотекой TA-lib, предоставляющей большое количество функций технических индикаторов.

Перед использованием библиотек нужно установить их в используемом окружении *Python*. Для этого в командной строке или *Windows PowerShell* с правами администратора системы нужно выполнить ряд команд:

- установка *NumPy*

```
pip install numpy
```

- установка *Pandas*

```
pip install pandas
```

- установка *Matplotlib*

```
pip install matplotlib
```

- установка *TensorFlow*

```
pip install tensorflow
```

- установка *Keras*

```
pip install keras
```

- установка библиотеки MetaTrader 5

```
pip install MetaTrader5
```

Переходя непосредственно к структуре нашего скрипта, создадим шаблон *template.py*. Скрипт будет состоять из нескольких блоков. Сначала нужно подключить необходимые библиотеки к нашему скрипту.

```
# импорт библиотек
import os
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow import keras
import matplotlib as mp
import matplotlib.pyplot as plt
import matplotlib.font_manager as fm
import MetaTrader5 as mt5
```

После обучения моделей мы будем строить графики визуализации процесса обучения и сопоставления работы различных моделей. Для стандартизации графиков зададим общие параметры их построения.

```
# устанавливаем параметры графиков результатов
mp.rcParams.update({'font.family':'serif',
                    'font.serif':'Clear Sans',
                    'axes.labelsize':'medium',
                    'legend.fontsize':'small',
                    'figure.figsize':[6.0,4.0],
                    'xtick.labelsize':'small',
                    'ytick.labelsize':'small',
                    'axes.titlesize': 'x-large',
                    'axes.titlecolor': '#333333',
                    'axes.labelcolor': '#333333',
                    'axes.edgecolor': '#333333'
                   })
```

Обучение и тестирование всех моделей будем осуществлять на одном наборе данных, который мы специально предварительно выгрузим в файл на локальном диске. Такой подход позволит нам исключить влияние несопоставимых данных и позволит оценить работу различных моделей нейронных сетей в одних условиях.

Следовательно, на следующем шаге мы загрузим исходные данные из файла в таблицу. Обратите внимание на следующий момент: поскольку *MetaTrader 5* ограничивает доступ к файлам из своих программ размерами песочницы, нужно указывать полный путь доступа к файлу исходных данных. Он будет находиться в каталоге *MQL5\Files* вашего терминала или его подкаталогах, если они были указаны при сохранении файла данных.

Вместо того, чтобы однозначно прописывать путь к песочнице терминала в коде нашей программы, мы просто запросим его из *MetaTrader 5* средствами предлагаемого API. Для этого мы

сначала подключаемся к установленному терминалу и проверяем результат выполнения данной операции.

```
# Подключаемся к терминалу MetaTrader 5
if not mt5.initialize():
    print("initialize() failed, error code =",mt5.last_error())
    quit()
```

После успешного подключения к терминалу запросим путь к песочнице и отключимся от терминала, потому что дальнейшие операции создания и обучения модели мы будем осуществлять средствами Python. Совершать какие-либо торговые операции в данном скрипте мы не планируем.

```
# Запрашиваем путь в песочницу
path=os.path.join(mt5.terminal_info().data_path,r'MQL5\Files')
mt5.shutdown()
```

В следующем небольшом блоке загрузки данных можно увидеть использование функций сразу трех вышеуказанных библиотек. С помощью функции *os.path.join* мы сцепляем путь к рабочему каталогу с именем файла обучающей выборки. Функцией *read_table* из библиотеки *Pandas* мы считываем и преобразуем содержимое *CSV*-файла в таблицу. Затем полученную таблицу преобразуем в двумерный массив с помощью функции библиотеки *NumPy*.

```
# Загрузка обучающей выборки
filename = os.path.join(path,'study_data.csv')
data = np.asarray( pd.read_table(filename,
                   sep=',',
                   header=None,
                   skipinitialspace=True,
                   encoding='utf-8',
                   float_precision='high',
                   dtype=np.float64,
                   low_memory=False))
```

Непосредственно считывание содержимого *CSV*-файла и преобразование строк в таблицу осуществляется функцией *read_table* из библиотеки *Pandas*. Данная функция имеет довольно много параметров для точной настройки методов преобразования строчных данных в требуемый числовой тип данных. Полное их описание можно найти в документации к библиотеке. Мы же опишем лишь используемые:

- *filename* — имя считываемого файла с указанием полного или относительного пути;
- *sep* — указываем используемый в файле разделитель данных;
- *header* — номера строк для использования в качестве имен столбцов и начала данных, при отсутствии заголовков указываем значении *None*;
- *skipinitialspace* — логический параметр указывает, пропускать ли пробелы после разделителя;
- *encoding* — указываем тип используемой кодировки;
- *float_precision* — определяет, какой преобразователь должен использоваться для значений с плавающей запятой;
- *dtype* — указывает конечный тип данных;
- *low_memory* — внутренняя обработка файла по частям, что приведет к меньшему использованию памяти при синтаксическом анализе.

В результате этих операций все данные обучающей выборки были загружены в объект двумерного массива типа *numpy.ndarray* из библиотеки *NumPy*. Среди загруженных данных есть элементы исходных данных и целевые значения. Но для обучения нейронной сети нам нужно отдельно подавать на ее вход исходные данные, а после прямого прохода сравнить полученный результат с целевыми значениями. Получается, что исходные данные и цели для нейронной сети разделены во времени и месту использования.

Следовательно, нам нужно разделить эти данные в отдельные массивы. Пусть каждая строка данных представляет отдельный паттерн данных, и последние два элемента строки содержат целевые точки данного паттерна. Функция *shape* покажет размер нашего массива, а значит с ее помощью мы можем определить размерности исходных данных и целевых значений. Только зная эти размерности, мы можем скопировать определенные выборки в новые массивы.

В блоке ниже мы разделим обучающую выборку на 2 таблицы. При этом мы разделяем только столбцы, полностью сохраняя структуру строк. Таким образом, мы получаем в одном массиве исходные данные, а в другом — целевые значения. Паттерны можно сопоставить с соответствующими целевыми значениями по номеру строки.

```
# Разделение обучающей выборки на исходные данные и цели
inputs=data.shape[1]-2
targerts=2
train_data=data[:,0:inputs]
train_target=data[:,inputs:]
```

Теперь, когда у нас есть данные для обучения, можно приступить к созданию модели нейронной сети. Модели мы будем создавать с помощью функции *Sequential* из библиотеки *Keras*.

```
# Создание модели нейронной сети
model = keras.Sequential([....])
```

Модель Sequential представляет собой линейный стек слоев. Можно создать модель Sequential, передав список слоев конструктору модели, а также можно добавлять слои с помощью метода *add*.

Прежде всего наша модель должна знать, какую размерность данных ожидать на входе. В связи с этим первый слой модели *Sequential* должен получать информацию о размерности входных данных. Все последующие слои производят автоматический расчет размерности.

Есть несколько способов указать размерность исходных данных:

- Передать аргумент *input_shape* первому слою.
- Некоторые 2D-слои поддерживают спецификацию размерности входных данных через аргумент *input_dim*. Некоторые 3D-слои поддерживают аргументы *input_dim* и *input_length*.
- Использовать специальный тип нейронного слоя для исходных данных *Input* с параметром *shape*, в котором указывается размер слоя.

```
# Создание модели нейронной сети
model = keras.Sequential([keras.Input(shape=inputs),
                          # Наполнить модель описанием нейронных слоев
                          ])
```

С типами предлагаемых нейронных слоев мы познакомимся по мере изучения их архитектуры. Сейчас давайте посмотрим на общие принципы построения и организации моделей.

После создания модели необходимо подготовить ее к обучению, настроить процесс. Этот функционал выполняется в методе *compile*, который имеет несколько параметров:

- *optimizer* — оптимизатор, может быть задан строковым идентификатором существующего оптимизатора или как экземпляр класса *Optimizer*;
- *loss* — функция потерь, может быть задана строковым идентификатором существующей функции потерь, или собственная функция;
- *metrics* — список показателей, которые модель должна оценивать во время обучения и тестирования, к примеру, для задачи классификации можно использовать ‘accuracy’;
- *loss_weights* — необязательный список или словарь, определяющий скалярные коэффициенты для взвешивания вкладов в потери различных выходных данных модели;
- weighted_metrics — список показателей, которые будут оценены и взвешены во время обучения и тестирования.

Для каждого параметра библиотека *Keras* предлагает свой список возможных значений, но при этом не ограничивает пользователя предложенными вариантами — для каждого параметра существует возможность добавления пользовательских классов и алгоритмов.

```
model.compile(optimizer='Adam',
              loss='mean_squared_error',
              metrics=['accuracy'])
```

Далее мы можем начать обучения созданной модели с помощью метода *fit*, который позволяет обучать модель с фиксированным количеством эпох. Данный метод имеет свои параметры для настройки процесса обучения.

- *x* — массив исходных данных;
- *y* — массив целевых результатов;
- *batch_size* — необязательный параметр, указывает количество наборов пар «исходные данные — целевые значения» до обновления матрицы весов;
- *epochs* — количество эпох обучения;
- *verbose* — необязательный параметр, указывает уровень детализации логирования обучения: 0 — без сообщений, 1 — индикатор выполнения, 2 — одна строка на эпоху, *auto* — автоматический выбор;
- *callbacks* — список обратных вызовов для применения во время обучения;
- *validation_split* — выделение части обучающей выборки для валидации, указывается в долях от 1,0;
- *validation_data* — отдельная выборка для валидации процесса обучения;
- *shuffle* — логическое значение, которое указывает на необходимость перемешивания данных обучающей выборки перед следующей эпохой;
- *class_weight* — необязательные индексы классов сопоставления словаря в значение веса, используемое для взвешивания функции потерь (только во время обучения);
- *sample_weight* — необязательный массив весов *NumPy* для обучающей выборки, используемый для взвешивания функции потерь (только во время обучения);
- *initial_epoch* — эпоха начала тренировки, может быть полезен для возобновления предыдущего тренировочного цикла;
- *steps_per_epoch* — общее количество пакетов перед объявлением одной эпохи завершенной и началом следующей, по умолчанию равен размеру обучающей выборки;

- *validation_steps* — общее количество пакетов из валидационной выборки перед остановкой при выполнении проверки в конце каждой эпохи, по умолчанию равен размеру валидационной выборки;
- *validation_batch_size* — количество образцов на партию валидации;
- *validation_freq* — целое число, указывает количество периодов обучения перед выполнением нового прогона валидации.

Конечно, мы не будем использовать полный набор параметров в первой же модели. Предлагаю остановиться на параметре *callbacks* — список обратных вызовов. Данный параметр предоставляет методы интерактивного взаимодействия с процессом обучения.

Его использование позволяет настроить получение оперативной информации о процессе обучения и управлять самим процессом. В частности можно накапливать средние значения показателей за эпоху или сохранять результаты каждой эпохи в файл *CSV*. Также можно отслеживать показатели обучения и уменьшать коэффициент обучения или вовсе остановить процесс обучения, когда отслеживаемый показатель перестал улучшаться. В то же время существует возможность добавления собственных классов обратного вызова.

Я предлагаю использовать ранний выход из процедуры обучения в том случае, если в течение пяти эпох не будет улучшения показателя функции ошибки.

```
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=5)
history = model.fit(train_data, train_target,
                    epochs=500, batch_size=1000,
                    callbacks=[callback],
                    verbose=2,
                    validation_split=0.2,
                    shuffle=true)
```

После завершения обучения сохраним обученную модель в файл на локальном диске. Для этого воспользуемся методами библиотек *Keras* и *os*.

```
# Сохранение обученной модели
model.save(os.path.join(path,'model.h5'))
```

Для наглядности и понимания процесса обучения выведем динамику изменения метрик в процессе обучения и валидации на графики. Здесь мы воспользуемся методами библиотеки *Matplotlib*.

```
# Отрисовка результатов обучения модели
plt.plot(history.history['loss'], label='Train')
plt.plot(history.history['val_loss'], label='Validation')
plt.ylabel('$MSE$ $Loss$')
plt.xlabel('$Epochs$')
plt.title('Dynamic of Models train')
plt.legend(loc='upper right')

plt.figure()
plt.plot(history.history['accuracy'], label='Train')
plt.plot(history.history['val_accuracy'], label='Validation')
plt.ylabel('$Accuracy$')
plt.xlabel('$Epochs$')
plt.title('Dynamic of Models train')
plt.legend(loc='lower right')
```

После обучения нам необходимо проверить работу нашей модели на тестовой выборке, ведь прежде, чем эксплуатировать модель в реальных условиях, нам необходимо знать, как она поведет себя на новых данных. Для этого мы загрузим тестовую выборку. Процедура загрузки данных полностью аналогична загрузке обучающей выборки — изменяем только имя файла.

```
# Загрузка тестовой выборки
test_filename = os.path.join(path,'test_data.csv')
test = np.asarray( pd.read_table(test_filename,
                   sep=',',
                   header=None,
                   skipinitialspace=true,
                   encoding='utf-8',
                   float_precision='high',
                   dtype=np.float64,
                   low_memory=false))
```

После загрузки данных разделим полученную таблицу на исходные данные и целевые метки, как и в случае с обучающей выборкой.

```
# Разделение тестовой выборки на исходные данные и цели
test_data=test[:,0:inputs]
test_target=test[:,inputs:]
```

Проверку качества работы обученной модели на тестовой выборке будем осуществлять с помощью метода *evaluate* из библиотеки *Keras*. В результате вызова указанного метода на выходе получаем значение функции потерь и метрик на тестовой выборке. Метод имеет ряд параметров для настройки процесса тестирования:

- *x* — массив исходных данных тестовой выборки;
- *y* — массив целей тестовой выборки;
- *batch_size* — размер пакета тестирования;
- *verbose* — режим детализации логирования процесса (0 — без логирования, 1 — индикация выполнения);
- *sample_weight* — необязательный параметр, используемый для взвешивания функции потерь;
- *steps* — общее количество шагов для объявления процесса тестирования завершенным;
- *callbacks* — список обратных вызовов, используемых в процессе обучения;

- *return_dict* — логическая переменная, которая определяет формат вывода результатов работы метода (*True* — в виде словаря «метрика - значение», *False* — в виде списка).

Большинство из указанных параметров необязательные, а также имеют значения по умолчанию. Для запуска процесса тестирования в большинстве случаев достаточно просто указать массивы данных.

```
# Проверка результатов модели на тестовой выборке
test_loss, test_acc = model.evaluate(test_data, test_target)
```

В заключение выведем результаты тестирования в журнал и отобразим построенные ранее графики.

```
# Вывод результатов тестирования в журнал
print('Model in test')
print('Test accuracy:', test_acc)
print('Test loss:', test_loss)

# Вывод созданных графиков
plt.show()
```

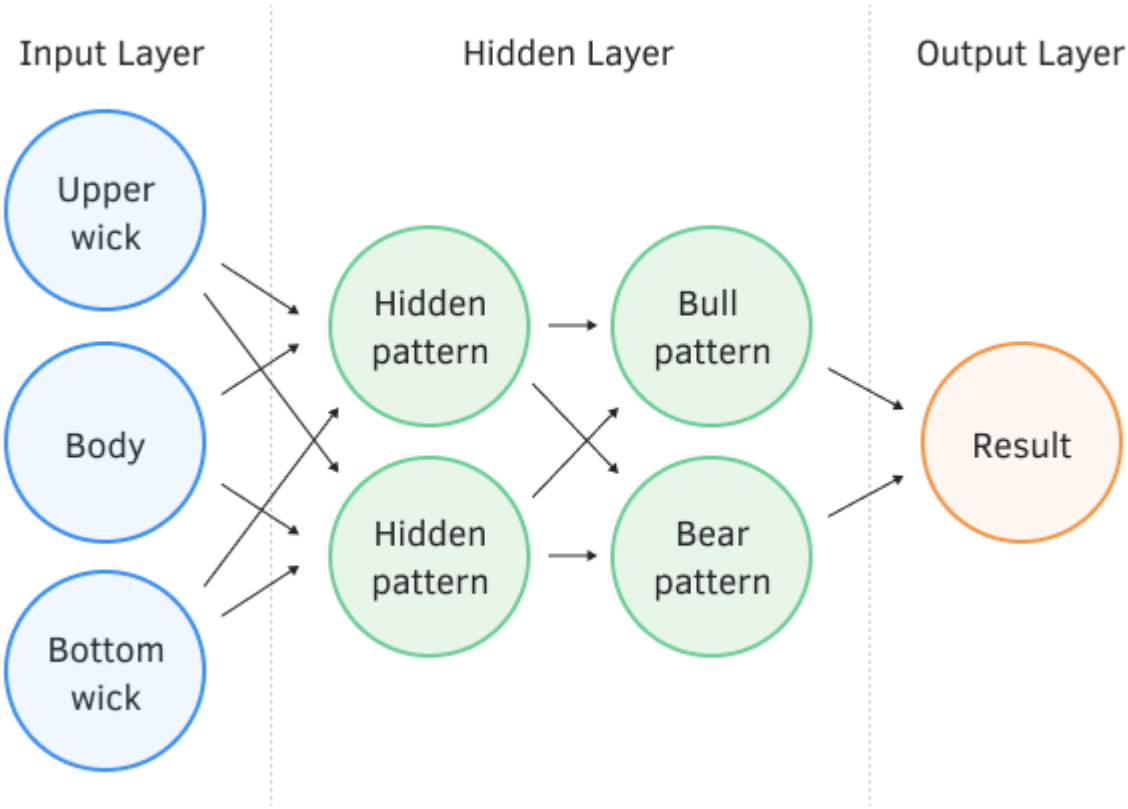
На этом базовый шаблон скрипта можно считать завершенным. Надо сказать, что при попытке запуска данного скрипта мы получим ошибку. И это не связано с ошибками в построении шаблона. Просто мы еще не дали описания нашей модели, а оставили пустой блок. В процессе рассмотрения различных решений нейронных сетей мы будем заполнять блок описания архитектуры модели и в полной мере сможем оценить работу нашего шаблона.

3.6 Полносвязный нейронный слой

В предыдущих разделах мы заложили основную логику организации и работы нейронной сети. Сейчас же мы переходим к конкретному ее наполнению. Начинаем построение непосредственных рабочих элементов — нейронных слоев и составляющих их нейронов. И начнем мы с построения полносвязного нейронного слоя.

Именно полносвязные нейронные слои составляли Перцепртон, созданный Фрэнком Розенблаттом в 1957 году. В данной архитектуре каждый нейрон слоя имеет связи со всеми нейронами предыдущего слоя. При этом каждая связь имеет свой весовой коэффициент. На рисунке ниже представлен перцептрон с двумя скрытыми полносвязными слоями. Выходной нейрон тоже можно представить как полносвязный слой с одним нейроном. Практически всегда на выходе нейронной сети мы будем использовать как минимум один полносвязный слой.

Можно сказать, что каждый нейрон оценивает общую картину входного вектора и реагирует на появление некоего паттерна. Благодаря подбору различных весовых коэффициентов выстраивается модель, в которой каждый нейрон реагирует на свой паттерн входного сигнала. Именно это свойство позволяет использовать полносвязный нейронный слой на выходе моделей-классификаторов.



Перцентрон имеет 2 срытых полносвязных слоя.

Если рассматривать полносвязный нейронный слой со стороны векторной математики, в представлении которой вектор входных значений представляет некую точку в *N*-мерном пространстве (где *N* — количество элементов во входном векторе), то каждый нейрон строит проекцию этой точки на своем векторе. При этом функция активации решает, передавать ли сигнал далее.

Здесь следует обратить внимание еще на момент смещения полученной проекции относительно начала координат. В то время как функция активации настроена на принятие решение в строгом диапазоне исходных данных, это смещение для нас является системной ошибкой. Для компенсации этого смещения на каждом нейроне вводится дополнительная константа, называемая ***bias***. На практике данная константа подбирается в процессе обучения вместе с весовыми коэффициентами. Для этого к вектору входного сигнала добавляется еще один элемент с постоянным значением 1, а подобранный весовой коэффициент при этом элементе и будет выполнять роль *bias*.

3.6.1 Описание архитектуры и принципов реализации полносвязного слоя

При построении базового класса нейронной сети и динамического массива хранения указателей на слои нейронов мы обозначили основные методы и интерфейсы для обмена данными между менеджером нашей нейронной сети и ее составляющими. Это и определяет основные публичные методы всех наших классов нейронных слоев. Предлагаю сейчас подвести небольшой итог упомянутых разделов. Выделим ключевые методы классов, которые нам еще предстоит написать, и их функционал.

Прежде всего надо сказать, что все объекты нейронных слоев должны быть наследниками базового класса *CObject*. Это основное требование для помещения указателей на экземпляры этих объектов в созданный нами динамический массив.

Придерживаясь общих принципов организации работы с объектами, в конструкторе класса мы инициализируем внутренние переменные и константы. В деструкторе будем выполнять очистку памяти: удаление всех внутренних экземпляров различных классов и очистку массивов.

Смотрим дальше. Метод *Init* получает в параметрах экземпляр класса *CLayerDescription* с описанием создаваемого нейронного слоя. Следовательно, в этом методе должна быть организована работа по созданию всей внутренней архитектуры для нормального функционирования нашего нейронного слоя. Нам потребуется создать несколько массивов для хранения данных.

Прежде всего это массив для записи состояний на выходе нейронов. Данный массив будет иметь размер равный количеству нейронов в нашем слое.

Также нам потребуется массив для хранения весовых коэффициентов. Это уже будет матрица, в которой размер первого измерения равен количеству нейронов в нашем слое, а размер второго измерения — на 1 больше размера массива исходных данных. Для полносвязного нейронного слоя массивом исходных данных являются выходные значения нейронов предыдущего слоя. Соответственно, размер второго измерения будет на 1 элемент больше размера предыдущего слоя. Добавленный элемент послужит для подбора *bias*-смещения.

Для обратного прохода нам потребуется массив для записи градиентов (отклонение расчетных значений от эталонных на выходе нейронов). Его размер будет соответствовать количеству нейронов в нашем слое.

Кроме того, в зависимости от метода обучения нам могут потребоваться одна или две матрицы для хранения накопленных моментов. Размеры этих матриц будут равны размеру матрицы весовых коэффициентов.

Мы не всегда будем производить обновление весовых коэффициентов после каждой итерации обратного прохода. Возможно обновление весов после полного прохода обучающей выборки или на основе некоторого пакета. Мы не будем сохранять промежуточные состояния всех нейронов и их входов. Напротив, после каждой итерации обратного прохода мы посчитаем необходимое изменение каждого весового коэффициента, как если бы мы осуществляли обновление весов на каждой итерации. Но вместо изменения весов будем суммировать полученные дельты в отдельный массив. При необходимости обновления просто возьмем среднее значение дельты за период и скорректируем веса. Для этого нам потребуется еще одна матрица размером равным матрице весовых коэффициентов.

Для всех массивов мы создадим специальный класс ***CBufferType***. Он будет наследоваться от базового класса ***CObject*** с добавлением необходимого функционала по организации работы буфера данных.

Помимо создания массивов и матриц нам предстоит их заполнить начальными значениями. Все массивы, кроме весовых коэффициентов, заполним нулями, а матрицу весов инициализируем случайными значениями.

Кроме массивов данных, наш класс также будет использовать локальные переменные. Нам потребуется сохранить параметры активации и оптимизации нейронов. Тип используемого метода оптимизации сохраним в переменную, а для функций активации создадим целую структуру отдельных классов с наследованием от одного базового класса.

Напомню, мы создаем универсальную платформу для создания нейронных сетей и их эксплуатации в терминале *MetaTrader 5*. Мы планируем дать пользователю возможность использовать многопоточные вычисления с помощью технологии *OpenCL*. Все объекты нашей

нейронной сети будут работать в одном контексте. Это позволит сократить затраты времени на излишнюю перегрузку данных. Непосредственно сам экземпляр класса для работы с технологией *OpenCL* будет создаваться в базовом классе нейронной сети, а указатель на созданный объект будет передан ко всем элементам нейронной сети. Следовательно, все объекты, составляющие нейронную сеть, в том числе и наш нейронный слой, должны иметь метод для получения указателя ***SetOpenCL*** и переменную для его хранения.

Прямой проход будет организован в методе ***FeedForward***. Единственным параметром данного метода будет указатель на объект *CNeuronBase* предыдущего слоя нейронной сети. От предыдущего слоя нам потребуются состояния на выходе нейронов, которые составят входящий поток данных. Для доступа к ним создадим метод ***GetOutputs***.

Обратный проход, в отличие от прямого, будет разделен на несколько методов:

- ***CalcOutputGradient*** — расчет градиента ошибки на выходном слое нейронной сети по эталонным значениям.
- ***CalcHiddenGradient*** — пропуск градиента ошибки через скрытый слой от выхода ко входу. В результате передадим градиенты ошибки на предыдущий слой. Для доступа к массиву градиентов предшествующего слоя нам потребуется метод для доступа к ним — ***GetGradients***.
- ***CalcDeltaWeights*** — расчет необходимого изменения весовых коэффициентов по результатам анализа последней итерации.
- ***UpdateWeights*** — метод непосредственного обновления весовых коэффициентов.

Ну и конечно, не забудем общие для всех объектов методы работы с файлами и идентификации ***Save***, ***Load*** и ***Type***.

В своей детализации объектов мы остановимся на классе нейронного слоя и не будем создавать отдельные объекты для каждого нейрона. На самом деле, тому есть целый ряд причин. Из того, что лежит на поверхности:

- Использование функции активации *Softmax* предполагает работу со всем нейронным слоем.
- Использование методов *Dropout* и *Layer Normalization* требует обработку данных всего нейронного слоя.
- Такой подход позволяет на базе матричных операций максимально эффективно организовать многопоточные вычисления.

Остановимся более подробно на матричных операциях и посмотрим, как это позволяет разделить операции по нескольким параллельным потокам. Рассмотрим небольшой пример из трех элементов на входе (вектор *Inputs*) и двух нейронов в слое. Оба нейрона имеют свои векторы весовых коэффициентов W_1 и W_2. При этом каждый вектор весов содержит по три элемента.

$$Inputs\{i_1, i_2, i_3\}$$
$$W_1\{w_{11}, w_{12}, w_{13}\}$$
$$W_2\{w_{21}, w_{22}, w_{23}\}$$

В соответствии с математической моделью нейрона нам необходимо поэлементно умножить вектор входных данных на вектор весовых коэффициентов, сложить полученные значения и применить к ним функцию активации. Примерно то же самое, за исключением функции активации, делает операция матричного умножения.

Напомню, умножения матриц — это операция, результатом которой является матрица. Элементы новой матрицы получаются путем суммы поэлементных произведений строк первой матрицы на столбцы второй матрицы.

Таким образом, чтобы получить сумму поэлементных произведений вектора входных данных на вектор весов одного из нейронов, нужно умножить вектор-строку исходных данных на вектор-столбец весовых коэффициентов.

$$Inputs * W_1 = [i_1 \quad i_2 \quad i_3] * \begin{bmatrix} w_{11} \\ w_{12} \\ w_{13} \end{bmatrix} = i_1 w_{11} + i_2 w_{12} + i_3 w_{13} = z_1$$

Данное правило применимо для любых матриц. Единственное условие — количество столбцов первой матрицы должно равняться количеству строк во второй матрице. Следовательно, мы можем собрать векторы весовых коэффициентов всех нейронов слоя в единую матрицу W, где каждый столбец будет представлять вектор весов отдельного нейрона.

$$Inputs * W = [i_1 \quad i_2 \quad i_3] * \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} = [z_1 \quad z_2]$$

Легко заметить, что вычисление любого элемента вектора Z осуществляется независимо от остальных элементов данного вектора. Соответственно, мы можем сразу загрузить в память матрицы исходных данных и весовых коэффициентов и затем в параллельных потоках одновременно посчитать значение всех элементов выходного вектора.

А можем пойти еще дальше и загрузить не один вектор исходных данных, а матрицу, строки которой будут представлять отдельное состояние системы. При работе с таймсериями каждая строка — это срез состояния системы в некий момент времени. В результате мы увеличим количество параллельных потоков операций и потенциально сократим время на обработку данных.

$$Inputs * W = \begin{bmatrix} i_{1t_1} & i_{2t_1} & i_{3t_1} \\ i_{1t_2} & i_{2t_2} & i_{3t_2} \\ i_{1t_3} & i_{2t_3} & i_{3t_3} \\ i_{1t_4} & i_{2t_4} & i_{3t_4} \end{bmatrix} * \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} = \begin{bmatrix} z_{1t_1} & z_{2t_1} \\ z_{1t_2} & z_{2t_2} \\ z_{1t_3} & z_{2t_3} \\ z_{1t_4} & z_{2t_4} \end{bmatrix}$$

И конечно, мы также можем использовать многопоточность для расчета значения функции активации на каждом независимом элементе матрицы *Z*. Исключением может быть использование функции активации *Softmax* из-за особенностей вычисления функции. Но и здесь возможна параллелизация вычислений на отдельных этапах вычисления функции.

3.6.2 Построение средствами MQL5

Приступая к реализации полносвязного нейронного слоя, следует учесть, что это будет базовый класс для всех последующих архитектурных решений нейронных слоев. Поэтому мы должны сделать его максимально универсальным с возможностью расширения функций. При этом расширения должны максимально легко вписываться в существующее решение.

Начнем с создания нашего базового класса нейронного слоя *CNeuronBase* с наследованием от класса *CObject*. Затем определим внутренние переменные класса:

- *m_cOpenCL* — указатель на экземпляр класса работы с технологией OpenCL;
- *m_cActivation* — указатель на объект функций активаций;
- *m_eOptimization* — тип метода оптимизации нейронов при обучении;
- *m_cOutputs* — массив значений на выходе нейронов;
- *m_cWeights* — массив весовых коэффициентов;
- *m_cDeltaWeights* — массив для накопления невыполненных обновлений весовых коэффициентов (суммарный градиент ошибки для каждого весового коэффициента после последнего обновления);
- *m_cGradients* — градиент ошибки на выходе нейронного слоя в результате последней итерации обратного прохода;
- *m_cMomenum* — в отличие от остальных переменных это будет массив из двух элементов для записи указателей на массивы накопления моментов.

Для облегчения доступа к переменным из классов-наследников все переменные будут объявлены в блоке *protected*.

В конструкторе класса проведем инициализацию вышеуказанных переменных параметрами по умолчанию. Я указал метод оптимизации *Adam* и *Swish* в качестве функции активации, вы же можете выбрать свой любимый метод оптимизации и функцию активации. Указатель на класс работы с *OpenCL* оставим пустым, создадим экземпляры для всех остальных используемых классов.

```
CNeuronBase::CNeuronBase(void)   : m_eOptimization(Adam)
  {
   m_cOpenCL = NULL;
   m_cActivation = new CActivationSwish();
   m_cOutputs = new CBufferType();
   m_cWeights = new CBufferType();
   m_cDeltaWeights = new CBufferType();
   m_cGradients = new CBufferType();
   m_cMomenum[0] = new CBufferType();
   m_cMomenum[1] = new CBufferType();
  }
```

Сразу создадим деструктор класса, чтобы не забыть про очистку памяти после работы класса.

```
CNeuronBase::~CNeuronBase(void)
  {
   if(!!m_cActivation)
      delete m_cActivation;
   if(!!m_cOutputs)
      delete m_cOutputs;
   if(!!m_cWeights)
      delete m_cWeights;
   if(!!m_cDeltaWeights)
      delete m_cDeltaWeights;
   if(!!m_cGradients)
      delete m_cGradients;
   if(!!m_cMomenum[0])
      delete m_cMomenum[0];
   if(!!m_cMomenum[1])
      delete m_cMomenum[1];
  }
```

Следующим создадим метода инициализации нейронного слоя. В параметрах метод получает экземпляр класса *CLayerDescription* с описанием создаваемого слоя. Чтобы не путаться в тонкостях алгоритма метода, предлагаю разбить его на отдельные логические блоки.

Начинается метод с блока проверки входящих параметров. Прежде всего проверяем действительность указателя на объект. Затем проверяем тип создаваемого слоя и количество нейронов в слое: в каждом слое должен быть хотя бы один нейрон, ведь с логической точки зрения построения нейронной сети слой без нейронов блокирует прохождение сигнала и парализует всю сеть. Обратите внимание, при проверке типа создаваемого слоя мы используем виртуальный метод *Type*, а не возвращаемую им константу *defNeuronBase*. Это очень важный момент для будущего наследования класса. Дело в том, что при использовании константы вызов такого метода для классов-наследников всегда бы возвращал *false* при попытке создать слой отличный от базового. Использование виртуального метода позволяет нам получить константу-идентификатор конечного класса-наследника, а проверка даст истинный результат сравнения заданного типа нейронного слоя и создаваемого объекта.

```
bool CNeuronBase::Init(const CLayerDescription *desc)
  {
//--- блок контроля исходных данных
   if(!desc || desc.type != Type() || desc.count <= 0)
      return false;
```

В следующем блоке проверим действительность ранее созданных буферов для записи исходящего из нейронного слоя потока данных и градиента к ним (при необходимости создаем новые экземпляры класса). Инициализируем массивы нулевыми значениями.

```
//--- создание буфера результатов
   if(!m_cOutputs)
      if(!(m_cOutputs = new CBufferType()))
         return false;
   if(!m_cOutputs.BufferInit(1, desc.count, 0))
      return false;
//--- создание буфера градиентов ошибки
   if(!m_cGradients)
      if(!(m_cGradients = new CBufferType()))
         return false;
   if(!m_cGradients.BufferInit(1, desc.count, 0))
      return false;
```

После этого проверим количество элементов входного сигнала. В случае использования нейронного слоя в качестве массива входящего сигнала у нас не будет предшествующих нейронных слоев и не потребуются остальные буферы для данных. Мы можем безболезненно их удалить и очистить память. Затем проверим действительность указателя на объект в *m_cOpenCL* и при положительном результате проверки создадим копию буфера данных в контексте *OpenCL*.

```
//--- удаление не используемых объектов для слоя исходных данных
   if(desc.window <= 0)
     {
      if(m_cActivation)
         delete m_cActivation;
      if(m_cWeights)
         delete m_cWeights;
      if(m_cDeltaWeights)
         delete m_cDeltaWeights;
      if(m_cMomenum[0])
         delete m_cMomenum[0];
      if(m_cMomenum[1])
         delete m_cMomenum[1];
      if(m_cOpenCL)
         if(!m_cOutputs.BufferCreate(m_cOpenCL))
            return false;
      m_eOptimization = desc.optimization;
      return true;
     }
```

Последующий код метода выполняется только при наличии предшествующих нейронных слоев. Создадим и инициализируем экземпляр метода функции активации. Этот процесс мы вынесли в отдельный метод *SetActivation*, который сейчас просто вызываем. Непосредственно алгоритм метода *SetActivation* рассмотрим немного позже.

```
//--- инициализация объекта функции активации
   VECTOR ar_temp = desc.activation_params;
   if(!SetActivation(desc.activation, ar_temp))
      return false;
```

Следующим шагом проведем инициализацию матрицы весовых коэффициентов. Определим количество элементов в матрице и инициализируем ее случайными значениями по методу Ксавье. В случае использования в качестве функции активации LReLU будем использовать метод Хе.

```
//--- инициализация объекта матрицы весов
   if(!m_cWeights)
      if(!(m_cWeights = new CBufferType()))
         return false;
   if(!m_cWeights.BufferInit(desc.count, desc.window + 1, 0))
      return false;
   double weights[];
   double sigma = (desc.activation == AF_LRELU ?
                  2.0 / (double)(MathPow(1 + desc.activation_params[0], 2)
                                                          * desc.window) :
                  1.0 / (double)desc.window);
   if(!MathRandomNormal(0, MathSqrt(sigma), m_cWeights.Total(), weights))
      return false;
   for(uint i = 0; i < m_cWeights.Total(); i++)
      if(!m_cWeights.m_mMatrix.Flat(i, (TYPE)weights[i]))
         return false;
```

Нам осталось провести инициализацию буферов дельт и моментов. Размер буферов будет равен размеру матрицы весов, инициализируем мы их нулевыми значениями. Следует помнить, что не все методы оптимизации одинаково используют матрицы моментов. Поэтому матрицы моментов мы будем инициализировать в зависимости от метода оптимизации. Лишние массивы очистим и удалим, чтобы освободить память для полезного использования.

```
//--- инициализация объекта накопления градиентов на уровне матрицы весов
   if(!m_cDeltaWeights)
      if(!(m_cDeltaWeights = new CBufferType()))
         return false;
   if(!m_cDeltaWeights.BufferInit(desc.count, desc.window + 1, 0))
      return false;
//--- инициализация объектов моментов
   switch(desc.optimization)
     {
      case None:
      case SGD:
         for(int i = 0; i < 2; i++)
            if(m_cMomenum[i])
               delete m_cMomenum[i];
         break;

      case MOMENTUM:
      case AdaGrad:
      case RMSProp:
         if(!m_cMomenum[0])
            if(!(m_cMomenum[0] = new CBufferType()))
               return false;
         if(!m_cMomenum[0].BufferInit(desc.count, desc.window + 1, 0))
            return false;
         if(m_cMomenum[1])
            delete m_cMomenum[1];
         break;

      case AdaDelta:
      case Adam:
         for(int i = 0; i < 2; i++)
           {
            if(!m_cMomenum[i])
               if(!(m_cMomenum[i] = new CBufferType()))
                  return(false);
            if(!m_cMomenum[i].BufferInit(desc.count, desc.window + 1, 0))
               return false;
           }
         break;

      default:
         return false;
         break;
     }
//--- сохранение метода оптимизации параметров
   m_eOptimization = desc.optimization;
   return true;
  }
```

В заключение метода сохраним заданный способ оптимизации весовых коэффициентов.

Метод *SetOpenCL* служит для сохранения указателя на объект работы с контекстом *OpenCL* и выглядит проще метода инициализации. Но и в нем есть свои нюансы. В отличие от всех

рассмотренных ранее методов, мы не завершаем работу метода при получении недействительного указателя на объект. Это связано с тем, что мы не вводим в каждый класс нейронного слоя флаг использования технологии *OpenCL*. Вместо этого мы используем один флаг в базовом классе нейронной сети. В свою очередь, для проверки использования технологии внутри класса мы можем проверить действительность указателя в переменной *m_cOpenCL*.

Следует обратить внимание на тот момент, что все объекты нейронной сети работают в одном контексте *OpenCL*. И до всех объектов доводится указатель на один и тот же объект класса *CMyOpenCL*. При подобном подходе, удаление экземпляра класса в одном из объектов нейронной сети сделает недействительным указатель во всех, использующих его, объектах. И флаг может не соответствовать текущему состоянию указателя. К тому же в случае отключения использования технологии мы оставляем возможность указания пустого значения указателя на объект.

Поэтому и код нашего метода можно условно разделить на две части. Первая часть кода будет отрабатывать при получении недействительного указателя на объект. В этом случае нам необходимо очистить все ранее созданные буфера данных в контексте *OpenCL*.

```
bool CNeuronBase::SetOpenCL(CMyOpenCL *opencl)
  {
   if(!opencl)
     {
      if(m_cOutputs)
         m_cOutputs.BufferFree();
      if(m_cGradients)
         m_cGradients.BufferFree();
      if(m_cWeights)
         m_cWeights.BufferFree();
      if(m_cDeltaWeights)
         m_cDeltaWeights.BufferFree();
      for(int i = 0; i < 2; i++)
        {
         if(m_cMomenum[i])
            m_cMomenum[i].BufferFree();
        }
      if(m_cActivation)
         m_cActivation.SetOpenCL(m_cOpenCL, Rows(), Cols());
      m_cOpenCL = opencl;
      return true;
     }
```

Вторая часть метода будет выполняться при получении действительного указателя на объект работы с контекстом *OpenCL*. Здесь мы организуем создание новых буферов данных в указанном контексте *OpenCL* для всех объектов текущего класса.

```
   if(m_cOpenCL)
      delete m_cOpenCL;
   m_cOpenCL = opencl;
   if(m_cOutputs)
      m_cOutputs.BufferCreate(opencl);
   if(m_cGradients)
      m_cGradients.BufferCreate(opencl);
   if(m_cWeights)
      m_cWeights.BufferCreate(opencl);
   if(m_cDeltaWeights)
      m_cDeltaWeights.BufferCreate(opencl);
   for(int i = 0; i < 2; i++)
     {
      if(m_cMomenum[i])
         m_cMomenum[i].BufferCreate(opencl);
     }

   if(m_cActivation)
      m_cActivation.SetOpenCL(m_cOpenCL, Rows(), Cols());
//---
   return(!!m_cOpenCL);
  }
```

Ранее мы говорили о выделении в отдельный метод процедуры инициализации функции активации. Предлагаю рассмотреть данный метод для завершения описания процесса инициализации нового объекта. Это один из немногих методов, в котором мы не организовываем блок проверки полученных данных. Проверка параметров функции активации не предоставляется возможным ввиду различия диапазона допустимых значений при использовании различных функций. В большинстве случаев диапазон их значений ограничивается только здравым смыслом и архитектурными требованиями модели.

Что же касается выбора функции активации, то он существует неявно, в виде перечисления допустимых значений. Но даже если пользователь подставит значение не из перечисления, то создавать объекты функции активации мы будем в теле оператора выбора *switch*. А значит, у нас будет неявный контроль типа функции активации, и при отсутствии указанного значения в функции выбора мы создадим базовый класс без функции активации.

Необходимость создания базового класса обусловлена сохранением работоспособности класса без использования функции активации в стандартном режиме. Как вы увидите позже, в некоторых случаях мы будем использовать нейронные слои без функций активации.

```
bool CNeuronBase::SetActivation(ENUM_ACTIVATION_FUNCTION function, VECTOR &params)
  {
   if(m_cActivation)
      delete m_cActivation;

   switch(function)
     {
      case AF_LINEAR:
         if(!(m_cActivation = new CActivationLine()))
            return false;
         break;

      case AF_SIGMOID:
         if(!(m_cActivation = new CActivationSigmoid()))
            return false;
         break;

      case AF_LRELU:
         if(!(m_cActivation = new CActivationLReLU()))
            return false;
         break;

      case AF_TANH:
         if(!(m_cActivation = new CActivationTANH()))
            return false;
         break;

      case AF_SOFTMAX:
         if(!(m_cActivation = new CActivationSoftMAX()))
            return false;
         break;

      case AF_SWISH:
         if(!(m_cActivation = new CActivationSwish()))
            return false;
         break;

      default:
         if(!(m_cActivation = new CActivation()))
            return false;
         break;
     }
```

После создания экземпляра объекта требуемой функции активации мы передаем в новый объект параметры функции и указатель на объект контекста *OpenCL*.

```
   if(!m_cActivation.Init(params[0], params[1]))
      return false;
   m_cActivation.SetOpenCL(m_cOpenCL, m_cOutputs.Rows(), m_cOutputs.Cols());
   return true;
  }
```

Операции прямого прохода будут реализованы в методе ***FeedForward***. В параметрах метод получает указатель на объект предыдущего слоя. Благодаря тому, что классы всех нейронных слоев планируется строить на основе одного базового класса, в параметрах метода мы можем

использовать класс базового нейронного слоя для получения указателя на предыдущий слой любого типа. При этом использование виртуальных методов доступа к внутренним объектам класса позволяет выстроить универсальный интерфейс без привязки к конкретному типу нейронного слоя.

В начале метода проверим действительность указателей на все используемые в методе объекты. Это наши исходные данные: полученный в параметрах указатель на предыдущий слой, а также содержащийся в нем буфер выходных состояний нейронов. Вместе с ними проверим указатели на матрицу весовых коэффициентов и буфер для записи результатов прямого прохода текущего слоя — буфер выходных состояний нейронов текущего слоя. Нелишним будет проверить и указатель на экземпляр класса для расчета значений функции активации.

```
bool CNeuronBase::FeedForward(CNeuronBase * prevLayer)
  {
//--- блок контролей
   if(!prevLayer || !m_cOutputs || !m_cWeights ||
      !prevLayer.GetOutputs() || !m_cActivation)
      return false;
   CBufferType* input_data = prevLayer.GetOutputs();
```

Затем проверим указатель на объект работы с *OpenCL*. Если указатель действительный — перейдем к блоку использования данной технологии. О нем мы поговорим чуть позже при рассмотрении организации процесса параллельных вычислений. При недействительном указателе на объект или его отсутствии перейдем к блоку вычислений стандартными средствами *MQL5*. Здесь мы сначала проверим соответствие размеров матриц и переформатируем матрицу исходных данных в вектор, добавив единичный элемент для *bias*-смещения. Выполним операцию матричного умножения на матрицу весовых коэффициентов. Результат запишем в буфер исходящего потока. Перед выходом из метода не забудем вычислить значения функции активации на выходе нейронного слоя.

```
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      if(m_cWeights.Cols() != (input_data.Total() + 1))
         return false;
      //---
      MATRIX m = input_data.m_mMatrix;
      if(!m.Reshape(1, input_data.Total() + 1))
         return false;
         m[0, m.Cols() - 1] = 1;
         m_cOutputs.m_mMatrix = m.MatMul(m_cWeights.m_mMatrix.Transpose());
        }

   else
     {
      //--- Здесь будет код обращения к OpenCL-программе
      return false;
     }
//---
   return m_cActivation.Activation(m_cOutputs);
  }
```

За прямым проходом идет обратный проход. Эту процедуру обучения нейронной сети мы разбиваем на составные части и создаем четыре метода:

- метод расчета градиента ошибки на выходе нейронной сети ***CalcOutputGradient***,
- метод распространения градиента через скрытый слой ***CalcHiddenGradient***,
- метод необходимого расчета корректирующих значений для весовых коэффициентов ***CalcDeltaWeights***,
- метод обновления матрицы весовых коэффициентов ***UpdateWeights***.

Мы будем двигаться по пути потока данных и рассмотрим алгоритм каждого метода.

В процессе обучения с учителем после прямого прохода расчетные значения на выходе нейронной сети сравниваются с эталонными значениями. В этот момент определяется отклонение на каждом нейроне выходного слоя. Эту операцию мы выполняем в методе ***CalcOutputGradient***. Алгоритм данного метода довольно прост: в параметрах метод получает массив эталонных значений и тип используемой функции потерь. В начале метода мы проверим действительность указателей на используемые объекты, а также соответствие размеров массивов.

```
bool CNeuronBase::CalcOutputGradient(CBufferType * target, ENUM_LOSS_FUNCTION loss)
  {
//--- блок контролей
   if(!target || !m_cOutputs || !m_cGradients ||
      target.Total() < m_cOutputs.Total() ||
      m_cGradients.Total() < m_cOutputs.Total())
      return false;
```

Далее, как и в методе прямого прохода, создадим разветвление алгоритма в зависимости от устройства проведения вычислений. Алгоритм с использованием технологии *OpenCL* будет рассмотрен в следующей главе, а сейчас посмотрим на построение процесса средствами *MQL5*.

Давайте посмотрим на процесс вычисления градиента ошибки на выходе нейронной сети. На первый взгляд, мы должны двигаться в сторону минимизации ошибки на каждом нейроне. То есть вычислять разницу между эталонным и расчетным значениями и минимизировать эту разницу. В таком случае мы получаем линейную зависимость ошибки и градиента. Это справедливо при использовании средней абсолютной ошибки в качестве функции потерь со всеми вытекающими преимуществами и недостатками.

Но говоря о функции потерь, мы с вами рассматривали и другие варианты, обсуждали их преимущества и недостатки. Но как нам воспользоваться их преимуществами? Ответ тут довольно прост. Нужно рассматривать функцию потерь и обучаемую модель как единую сложную функцию. В этом случае мы должны минимизировать не отклонение на каждом нейроне выходного слоя, а непосредственно значение функции потерь. И так же, как и при проведении градиента ошибки через нейронную сеть, мы определяем производную функции потерь и умножаем ее на отклонение значения функции потерь от нуля. При этом если для MAE и MSE мы можем взять в качестве ошибки только производную функции потерь и пренебречь умножением на значение функции потерь, так как это линейное масштабирование будет компенсировано коэффициентом обучения, то при использовании кросс-энтропии мы вынуждены осуществить умножение на значение функции потерь. Дело в том, что при равенстве целевого и расчетного значений функция потерь даст 0, а ее производная будет равна −1. И если мы производную не умножим на ошибку, то продолжим корректировать параметры модели при отсутствии ошибки.

При этом вовсе не обязательно полностью вычислять значение функции потерь. Кросс-энтропия в качестве функции потерь чаще всего используется при решении задач классификации. Следовательно, в качестве целевых значений мы ожидаем получить вектор, в котором только один элемент будет содержать единицу, а все остальные будут равны нулю. Для нулевых значений производная тоже будет равна нулю, а умножение на 1 не изменяет результат. Поэтому нам достаточно производную умножить логарифм расчетного значения. Именно логарифм от 1 даст 0, указывающий на отсутствие ошибки.

С учетом вышесказанного для вычисления соответствующего градиента ошибки на выходе модели воспользуемся переключателем *switch*, на котором создадим разветвление процесса в зависимости от используемой функции потерь. В случае отсутствия указанной функции потерь мы посчитаем простое отклонение расчетных результатов от целевых значений.

```
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      switch(loss)
        {
         case LOSS_MAE:
            m_cGradients.m_mMatrix = target.m_mMatrix - m_cOutputs.m_mMatrix;
            break;
         case LOSS_MSE:
            m_cGradients.m_mMatrix = (target.m_mMatrix - m_cOutputs.m_mMatrix) * 2;
            break;
         case LOSS_CCE:
            m_cGradients.m_mMatrix = target.m_mMatrix /
           (m_cOutputs.m_mMatrix + FLT_MIN) * MathLog(m_cOutputs.m_mMatrix) * (-1);
            break;

         case LOSS_BCE:
            m_cGradients.m_mMatrix = (target.m_mMatrix-m_cOutputs.m_mMatrix) /
               (MathPow(m_cOutputs.m_mMatrix, 2) - m_cOutputs.m_mMatrix + FLT_MIN);
            break;
         default:
            m_cGradients.m_mMatrix = target.m_mMatrix - m_cOutputs.m_mMatrix;
            break;
        }
     }
   else
      return false;
//---
   return true;
  }
```

После получения ошибки на выходе нейронной сети необходимо определить, какое влияние на нее оказывает каждый нейрон нашей сети. Для этого нужно слой за слоем распространить градиент ошибки до каждого нейрона. За организацию цикла перебора слоев нейронной сети отвечает менеджер сети — базовый класс нейронной сети *CNet*. Сейчас мы рассмотрим организацию процесса внутри одного нейронного слоя.

В параметрах метод ***CalcHiddenGradient*** получает указатель на предыдущий слой нейронной сети. Он нам потребуется для записи передаваемого градиента ошибки. В предыдущем методе мы определили ошибку на выходе нейрона, но выходное значение нейрона зависит от функции активации. Чтобы определить влияние каждого элемента исходных данных на конечный результат, нужно исключить влияние на ошибку функции активации. Для этого скорректируем градиент ошибки на производную функции активации. Эта операция, как и вычисление самой функции активации, вынесена в отдельный класс.

```
bool CNeuronBase::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
//--- корректировка входящего градиента на производную функции активации
   if(!m_cActivation.Derivative(m_cGradients))
      return false;
```

Далее идет блок проверок указателей используемых объектов. Сначала проверим действительность полученного указателя на предыдущий слой. Затем извлечем и проверим

указатели на буферы результатов и градиентов предыдущего слоя. Также проверим соответствие количества элементов в указанных буферах. Кроме того, проверим наличие достаточного количества элементов в матрице весов. Такое количество превентивных контролей необходимо для стабильной работы метода и исключения возможных ошибок при обращении к массивам данных.

```
//--- проверка буферов предыдущего слоя
   if(!prevLayer)
      return false;
   CBufferType *input_data = prevLayer.GetOutputs();
   CBufferType *input_gradient = prevLayer.GetGradients();
   if(!input_data || !input_gradient ||
      input_data.Total() != input_gradient.Total())
      return false;
//--- проверка соответствия размера буфера исходных данных и матрицы весов
   if(!m_cWeights || m_cWeights.Cols() != (input_data.Total() + 1))
      return false;
```

После успешного прохождения всех проверок переходим непосредственно к вычислительной части. Напомню, производная от произведения переменной на константу является константой. В данном случае производной по нейрону является соответствующий весовой коэффициент. Соответственно, влияние нейрона на результат является произведением градиента ошибки на выходе из функции на соответствующий весовой коэффициент. Сумму таких произведений мы и посчитаем для каждого нейрона предыдущего слоя. Полученные значения запишем в соответствующую ячейку буфера градиентов предыдущего слоя.

Как и в описанных выше методах мы осуществляем разделение алгоритма в зависимости от используемого вычислительного устройства. С алгоритмом реализации многопоточных вычислений мы познакомимся немного позже. Сейчас же рассмотрим реализацию алгоритма средствами *MQL5*. Как уже сказано выше, нам необходимо для каждого нейрона собрать сумму произведений градиентов ошибки зависящих от него нейронов на соответствующие весовые коэффициенты. Выполнение данной операции легко осуществляется с использованием операции матричного умножения. В данном случае достаточно умножить матрицу градиентов ошибки на матрицу весовых коэффициентов. Результат операции запишем в локальную матрицу.

Мы не можем сразу записать результат операции матрицу градиентов ошибки предыдущего слоя. Если вы посмотрите на метод прямого прохода, то увидите, как мы добавляли элемент *bias*-смещения. Соответственно, при умножении матриц мы получим результат с учетом ошибки на элементе смещения. Но предыдущий слой не ожидает это значение, и размер матрицы градиентов меньше. Поэтому мы сначала изменим размер матрицы, полученной в результате операции умножения, до требуемых размеров, а затем перенесем ее значения в матрицу градиентов предыдущего слоя.

Обратите внимание, что в данном методе мы не корректируем полученный на выходе предыдущего слоя градиент на производную функцию активации нейронов в предыдущем слое, ведь именно с аналогичной операции мы начинали данный метод. Поэтому если предыдущий слой является скрытым слоем нашей сети, то первое, что будет сделано при вызове рассмотренного метода на нижнем слое, — это корректировка градиента на производную функции активации. Задвоение операции приведет к ошибкам.

```
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      MATRIX grad = m_cGradients.m_mMatrix.MatMul(m_cWeights.m_mMatrix);
      if(!grad.Reshape(input_data.Rows(), input_data.Cols()))
         return false;
      input_gradient.m_mMatrix = grad;
     }
   else
      return false;
//---
   return true;
  }
```

Теперь у нас есть посчитанный градиент ошибки на каждом нейроне нашей сети. Данных достаточно для проведения обновления весовых коэффициентов. Но мы помним, что обновление весовых коэффициентов не всегда будет осуществляться после каждой итерации обратного прохода. Поэтому процесс обновления матрицы весовых коэффициентов мы разбили на два метода. В первом мы определим градиент ошибки для каждого весового коэффициента по аналогии с градиентом ошибке на нейроне предыдущего слоя, а во втором произведем корректировку матрицы весов.

Вычислять величину градиент ошибки для матрицы весовых коэффициентов будем в методе ***CalcDeltaWeights***. В параметрах метода, как и предыдущего, будет указатель на предшествующей слой нейронной сети, но теперь из него мы будем использовать не буфер градиентов, а массив выходных значений.

По аналогии с ранее рассмотренными методами метод начинается с блока проверок. За ним идет блок вычислений.

```
bool CNeuronBase::CalcDeltaWeights(CNeuronBase *prevLayer, bool read);
  {
//--- блок контролей
   if(!prevLayer || !m_cDeltaWeights || !m_cGradients)
      return false;
   CBufferType *Inputs = prevLayer.GetOutputs();
   if(!Inputs)
      return false;
```

В предыдущем методе мы уже скорректировали градиент на производную функции активации. Поэтому эту итерацию мы пропустим и перейдем сразу к непосредственному расчету градиента на весовых коэффициентах. Здесь, как в других методах, идет разветвление алгоритма по устройству вычислений. В блоке *MQL5* аналогично предыдущему методу мы воспользуемся матричным умножением, ведь, по существу, оба метода выполняют аналогичную операцию только для разных матриц. Но здесь есть несколько отличий.

Во-первых, в предыдущем методе мы убирали элемент *bias*-смещения. Сейчас же нам наоборот надо добавить единичный элемент к вектору результатов предыдущего слоя для определения градиента ошибки на соответствующем весовом коэффициенте.

Во-вторых, ранее мы умножали матрицу градиентов на матрицу весов. Теперь мы умножаем транспонированную матрицу градиентов ошибки на вектор результатов предыдущего слоя с элементом смещения.

Кроме того, мы перезаписывали градиент ошибки предыдущего слоя, а градиент весовых коэффициентов мы будем суммировать и тем самым накапливать градиент ошибки за весь период между операциями обновления весовых коэффициентов.

```
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      MATRIX m = Inputs.m_mMatrix;
      if(!m.Reshape(1, Inputs.Total() + 1))
         return false;
      m[0, Inputs.Total()] = 1;
      m = m_cGradients.m_mMatrix.Transpose().MatMul(m);
      m_cDeltaWeights.m_mMatrix += m;
     }
   else
      return false;
//---
   return true;
  }
```

В заключение процесса обратного прохода нам остается скорректировать матрицу весовых коэффициентов. Для выполнения этого функционала в нашем классе предусмотрен метод ***UpdateWeights***. Не будем забывать, что у нас возможны варианты с выбором метода оптимизации. Вопрос был решен простым и наглядным способом. В публичном методе обновления весовых коэффициентов реализована диспетчерская функция по выбору метода оптимизации в соответствии с выбором пользователя. Непосредственно процесс корректировки матрицы вынесен в отдельные методы, по одному методу на каждый вариант оптимизации.

```
bool CNeuronBase::UpdateWeights(int batch_size, TYPE learningRate,
                                VECTOR &Beta, VECTOR &Lambda)
  {
//--- блок контролей
   if(!m_cDeltaWeights || !m_cWeights ||
       m_cWeights.Total() < m_cDeltaWeights.Total() || batch_size <= 0)
      return false;
//---
   bool result = false;
   switch(m_eOptimization)
     {
      case None:
         result = true;
         break;

      case SGD:
         result = SGDUpdate(batch_size, learningRate, Lambda);
         break;
      case MOMENTUM:
         result = MomentumUpdate(batch_size, learningRate, Beta, Lambda);
         break;
      case AdaGrad:
         result = AdaGradUpdate(batch_size, learningRate, Lambda);
         break;
      case RMSProp:
         result = RMSPropUpdate(batch_size, learningRate, Beta, Lambda);
         break;
      case AdaDelta:
         result = AdaDeltaUpdate(batch_size, Beta, Lambda);
         break;
      case Adam:
         result = AdamUpdate(batch_size, learningRate, Beta, Lambda);
         break;
     }
//---
   return result;
  }
```

Алгоритмы каждого из методов оптимизации весовых коэффициентов уже были представлены при изучении их особенностей. Мы не будем их здесь дублировать, но вынесем в блок ***protected*** нашего базового класса нейронного слоя.

Выше мы уже рассмотрели реализацию операций прямого и обратного прохода в полносвязном нейронном слое. Но мы же не будем заново обучать нейронную сеть при каждом запуске. Поэтому нам нужны методы работы с файлами: записи и чтения данных состояния нейронного. Мы должны экономно относиться к нашим ресурсам, поэтому давайте подумаем какую информацию мы будем сохранять. Общее привило: нужно сохранять минимум информации, но ее должно быть достаточно для быстрого запуска и функционирования класса как без прерывания процесса. Давайте посмотрим на внутренние переменные класса и критически оценим необходимость сохранения их содержимого в файл.

- *m_cOpenCL* — указатель на экземпляр класса работы с технологией OpenCL, который отвечает за отдельный функционал, но не содержит дополнительной информации. *Не подлежит записи в файл.*
- *m_cActivation* — указатель на объект функций активаций. Тип функции активации задается пользователем при конструировании нейронной сети. Использование другой функции активации может привести к искажению результатов работы всей сети. *Сохраняем*.
- *m_eOptimization* — тип метода оптимизации нейронов при обучении, который задается пользователем при конструировании нейронной сети. Влияет на процесс обучения. *Сохраняем*.
- *m_cOutputs* — массив значений на выходе нейронов. Количество элементов задается архитектором нейронной сети. Содержимое перезаписывается при каждом прямом проходе. *Достаточно сохранить количество нейронов в слое и не сохранять весь массив.*
- *m_cWeights* — матрица весовых коэффициентов. Значение элементов формируется в процессе обучения нейронной сети. *Сохраняем*.
- *m_cDeltaWeights* — матрица для накопления не выполненных обновлений весовых коэффициентов (суммарный градиент ошибки для каждого весового коэффициента после последнего обновления). Значения накапливаются между обновлениями матрицы весов и обнуляются после корректировки весов. Размер массива равен матрице весовых коэффициентов. *Не подлежит записи в файл*.
- *m_cGradients* — градиент ошибки на выходе нейронного слоя в результате последней итерации обратного прохода. Содержимое перезаписывается при каждом обратном проходе. Размер массива равен буферу выходного сигнала. *Не подлежит записи в файл*.
- *m_cMomenum* — в отличие от остальных переменных, это будет массив из двух элементов для записи указателей на массивы накопления моментов. Использование буферов зависит от метода оптимизации. Содержимое накапливается в процессе обучения нейронной сети. *Сохраняем*.

После определения объема данных для записи в файл приступим к созданию метода записи в файл ***Save***. Этот виртуальный метод существует во всех классах-наследниках класса *CObject*. В параметрах метод получает хендл файла для записи.

В теле метода мы сначала проверим полученный хендл и действительность указателя на буфер результатов нейронного слоя. Как мы помним, нейронный слой может использоваться как с полным функционалом, так и нет. При использовании объекта в качестве слоя исходных данных мы удаляли все буферы, кроме буфера исходных данных. Поэтому наличие этого буфера является обязательным для нейронного слоя. Если какой-либо из контролей не пройден, выходим из метода с результатом false.

Далее записываем в файл тип нейронного слоя и размер буфера результатов. При этом не забываем проверить результат выполнения операций.

```
bool CNeuronBase::Save(const int file_handle)
  {
//--- блок контролей
   if(file_handle == INVALID_HANDLE)
      return false;
//--- запись данных буфера результатов
   if(!m_cOutputs)
      return false;
   if(FileWriteInteger(file_handle, Type()) <= 0 ||
      FileWriteInteger(file_handle, m_cOutputs.Total()) <= 0)
      return false;
```

После успешной записи размера буфера результатов мы проверяем действительность указателей на объекты функции активации и матрицы весовых коэффициентов. При отсутствии хотя бы одного объекта мы считаем текущий нейронный слой слоем исходных данных. В подтверждение этого записываем в файл 1 в качестве флага сохранения слоя исходных данных. В противном случае сохраняем 0, что будет свидетельствовать о сохранении нейронного слоя с полной функциональностью.

```
//--- проверка и запись флага слоя исходных данных
   if(!m_cActivation || !m_cWeights)
     {
      if(FileWriteInteger(file_handle, 1) <= 0)
         return false;
      return true;
     }
   if(FileWriteInteger(file_handle, 0) <= 0)
      return false;
```

Далее по методу оптимизации определяем количество необходимых для записи буферов моментов.

```
   int momentums = 0;
   switch(m_eOptimization)
     {
      case SGD:
         momentums = 0;
         break;
      case MOMENTUM:
      case AdaGrad:
      case RMSProp:
         momentums = 1;
         break;
      case AdaDelta:
      case Adam:
         momentums = 2;
         break;
      default:
         return false;
         break;
     }
```

Сразу же организовываем цикл проверки действительности указателей на буфера моментов.

```
   for(int i = 0; i < momentums; i++)
      if(!m_cMomenum[i])
         return false;
```

За блоком контролей идут операции непосредственной записи данных в файл. Сначала сохраним значение переменных, а следом вызовем методы записи в файл объектов, подлежащих сохранению.

```
//--- сохранение матрицы весовых коэффициентов, моментов и функции активации
   if(FileWriteInteger(file_handle, (int)m_eOptimization) <= 0 ||
      FileWriteInteger(file_handle, momentums) <= 0)
      return false;
   if(!m_cWeights.Save(file_handle) || !m_cActivation.Save(file_handle))
      return false;
   for(int i = 0; i < momentums; i++)
      if(!m_cMomenum[i].Save(file_handle))
         return false;
//---
   return true;
  }
```

Как видно из представленного кода, объекты не подлежащие сохранению мы просто пропустили. Но при загрузке данных из файла такой подход неприменим, так как даже пропущенные объекты необходимы для нормального функционирования нейронного слоя. Поэтому метод загрузки данных ***Load*** должен быть дополнен блоком инициализации пропущенных объектов. Посмотрим, как это реализовано.

Так же, как при записи в файл, в параметрах метод получает хендл файла с данными. Соответственно, в начале метода проверяем действительность полученного хендла.

```
bool CNeuronBase::Load(const int file_handle)
  {
//--- блок контролей
   if(file_handle == INVALID_HANDLE)
      return false;
```

Считывание данных из файла должно осуществляться в точном соответствии с последовательностью записи данных. Первым мы сохранили тип нейронного слоя и количество элементов в буфере в буфере результатов. Тип нейронного слоя будет считывать метод объекта верхнего уровня (динамического массива нейронных слоев), чтобы создать требуемый нейронный слой. В теле данного метода мы прочитаем количество элементов в буфере результата и инициализируем буфер соответствующего размера.

```
//--- загрузка буфера результатов
   if(!m_cOutputs)
      if(!(m_cOutputs = new CBufferType()))
         return false;
   int outputs = FileReadInteger(file_handle);
   if(!m_cOutputs.BufferInit(1, outputs, 0))
      return false;
```

И сразу создадим буфер градиентов аналогичного размера.

```
//--- создание буфера градиентов ошибки
   if(!m_cGradients)
      if(!(m_cGradients = new CBufferType()))
         return false;
   if(!m_cGradients.BufferInit(1, outputs, 0))
      return false;
```

Далее проверяем флаг загрузки нейронного слоя исходных данных. В случае загрузки такового удаляем неиспользуемые объекты и выходим из метода с положительным результатом.

```
//--- проверка флага слоя исходных данных
   int input_layer = FileReadInteger(file_handle);
   if(input_layer == 1)
     {
      if(m_cActivation)
         delete m_cActivation;
      if(m_cWeights)
         delete m_cWeights;
      if(m_cDeltaWeights)
         delete m_cDeltaWeights;
      if(m_cMomenum[0])
         delete m_cMomenum[0];
      if(m_cMomenum[1])
         delete m_cMomenum[1];
      if(m_cOpenCL)
         if(!m_cOutputs.BufferCreate(m_cOpenCL))
            return false;
      m_eOptimization = None;
      return true;
     }
```

Дальнейший код выполняется только при загрузке полностью функционального нейронного слоя. В начале этого блока считываем из файла метод оптимизации и количество используемых буферов моментов.

```
   m_eOptimization = (ENUM_OPTIMIZATION)FileReadInteger(file_handle);
   int momentums = FileReadInteger(file_handle);
```

После этого проверяем указатель на объект матрицы весов. В случае необходимости создаем новый экземпляр объекта и сразу вызываем метод загрузки буфера данных.

```
//--- создание объектов перед загрузкой данных
   if(!m_cWeights)
      if(!(m_cWeights = new CBufferType()))
         return false;
//--- загрузка данный из файла
   if(!m_cWeights.Load(file_handle))
      return false;
```

Затем считаем из файла тип используемой функции активации и инициализируем экземпляр соответствующего класса с помощью метода *SetActivation*. Параметры функции активации будут загружены путем вызова одноименного метода загрузки данных объекта функции активации.

```
//--- функция активации
   if(FileReadInteger(file_handle) != defActivation)
      return false;
   ENUM_ACTIVATION_FUNCTION activation =
                          (ENUM_ACTIVATION_FUNCTION)FileReadInteger(file_handle);
   if(!SetActivation(activation,VECTOR::Zeros(2)))
      return false;
   if(!m_cActivation.Load(file_handle))
      return false;
```

Аналогичным образом загрузим данные буферов моментов.

```
//---
   for(int i = 0; i < momentums; i++)
     {
      if(!m_cMomenum[i])
         if(!(m_cMomenum[i] = new CBufferType()))
            return false;
      if(!m_cMomenum[i].Load(file_handle))
         return false;
     }
```

После загрузки данных проведем инициализацию буфера *m_cDeltaWeights*. Буфер будем инициализировать нулевыми значениями. При этом размер буфера равен числу элементов в матрице весов.

Сначала проверим указатель на объект и при необходимости создадим новый. Затем запишем во все элементы буфера 0.

```
//--- инициализация оставшихся буферов
   if(!m_cDeltaWeights)
      if(!(m_cDeltaWeights = new CBufferType()))
         return false;
   if(!m_cDeltaWeights.BufferInit(m_cWeights.m_mMatrix.Rows(),
                                  m_cWeights.m_mMatrix.Cols(), 0))
      return false;
```

В заключение метода передадим текущий указатель m_cOpenCL во все внутренние объекты. Здесь мы не ставим проверку на действительность указателя. Напомню, все объекты нейронной сети работают в одном контексте OpenCL, поэтому даже недействительный указатель передаем в объекты.

```
//--- передача указателя на контекст OpenCL в объекты
   SetOpenCL(m_cOpenCL);
//---
   return true;
  }
```

В результате реализации всех вышеописанных методов финальная структура нашего класса приняла нижеследующий вид.

```
class CNeuronBase    :  public CObject
  {
protected:
   bool              m_bTrain;
   CMyOpenCL*        m_cOpenCL;
   CActivation*      m_cActivation;
   ENUM_OPTIMIZATION m_eOptimization;
   CBufferType*      m_cOutputs;
   CBufferType*      m_cWeights;
   CBufferType*      m_cDeltaWeights;
   CBufferType*      m_cGradients;
   CBufferType*      m_cMomenum[2];

   //---
   virtual bool      SGDUpdate(int batch_size, TYPE learningRate,
                               VECTOR &Lambda);
   virtual bool      MomentumUpdate(int batch_size, TYPE learningRate,
                               VECTOR &Beta, VECTOR &Lambda);
   virtual bool      AdaGradUpdate(int batch_size, TYPE learningRate,
                               VECTOR &Lambda);
   virtual bool      RMSPropUpdate(int batch_size, TYPE learningRate,
                               VECTOR &Beta, VECTOR &Lambda);
   virtual bool      AdaDeltaUpdate(int batch_size,
                               VECTOR &Beta, VECTOR &Lambda);
   virtual bool      AdamUpdate(int batch_size, TYPE learningRate,
                               VECTOR &Beta, VECTOR &Lambda);
   virtual bool      SetActivation(ENUM_ACTIVATION_FUNCTION function,
                               VECTOR &params);

public:
                     CNeuronBase(void);
                    ~CNeuronBase(void);
   //---
   virtual bool      Init(const CLayerDescription *description);
   virtual bool      SetOpenCL(CMyOpenCL *opencl);
   virtual bool      FeedForward(CNeuronBase *prevLayer);
   virtual bool      CalcOutputGradient(CBufferType *target,
                               ENUM_LOSS_FUNCTION loss);
   virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer);
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer);
   virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                               VECTOR &Beta, VECTOR &Lambda);
   virtual void      TrainMode(bool flag)         {  m_bTrain = flag;            }
   virtual bool      TrainMode(void)        const {  return m_bTrain;            }
   //---
   CBufferType       *GetOutputs(void)      const {  return(m_cOutputs);         }
   CBufferType       *GetGradients(void)    const {  return(m_cGradients);       }
   CBufferType       *GetWeights(void)      const {  return(m_cWeights);         }
   CBufferType       *GetDeltaWeights(void) const {  return(m_cDeltaWeights);    }
```

```
   virtual bool      SetOutputs(CBufferType* buffer, bool delete_prevoius = true);
   //--- methods for working with files
   virtual bool      Save(const int file_handle);
   virtual bool      Load(const int file_handle);
   //--- method of identifying the object
   virtual int       Type(void)              const { return(defNeuronBase);       }
   virtual ulong     Rows(void)              const { return(m_cOutputs.Rows());   }
   virtual ulong     Cols(void)              const { return(m_cOutputs.Cols());   }
   virtual ulong     Total(void)             const { return(m_cOutputs.Total());  }
  };
```

3.6.3 Класс функции активации

В процессе реализации базового класса нейронного слоя у нас остались несколько открытых вопросов. Один из них — это класс функции активации нейрона.

В класс функции активации я вынес операции вычисления функции активации и ее производной. Мы знаем, что есть различные варианты функций активации. В книге дан их не полный список — мы постарались рассмотреть функции, которые чаще других используются. Я не исключаю возможность появления новых, хорошо работающих функций активаций. И если вам потребуется добавить какую-либо функцию активации в эту библиотеку, то легче всего это будет сделать путем создания нового класса наследником некоего базового класса. Таким образом, переопределив пару методов непосредственного расчета функции и ее производной, изменения будут распространены по всем объектам нейронной сети, в том числе и ранее созданным.

В рамках этой логики было решено создать не один класс функции активации, который бы охватывал все рассмотренные ранее функции, а структуру классов, в которой бы каждый класс содержал алгоритм только одной функции активации. При этом во главе структуры стоял бы один базовый класс, который бы закладывал интерфейсы взаимодействия с другими объектами и служил бы объектом для обращения к методам из других объектов без привязки к конкретной функции активации.

Таким образом, создав единственное разветвление алгоритма при инициализации конкретного класса функции активации, мы уходим от проверки используемой функции на каждой итерации прямого и обратного проходов.

Родительский класс всех функций активации ***CActivation*** я решил наследоваться от класса ***CObject*** — базового класса всех объектов в *MQL5*.

Класс *CActivation* содержит лишь методы организации интерфейса и не описывает ни одну из функций активации. В свою очередь, для организации работы классов функции активации я определил следующие методы:

- *CActivation* — конструктор класса;
- *~CActivation* — деструктор класса;
- *Init* — передача параметров для расчета функции активации;
- *GetFunction* — получение используемой функции активации и ее параметров;
- *Activation* — вычисление значения функции активации по исходному значению;
- *Derivative* — производная от функции активации;
- *SetOpenCL* — запись указателя на объект работы с OpenCL;

- *Save* и *Load* — виртуальные методы для работы с файлами;
- *Type* — виртуальный метод идентификации класса.

Да и в целом класс выглядит намного проще ранее рассмотренных. В конструкторе класса зададим параметры функции активации по умолчанию.

```
CActivation::CActivation(void) : m_iRows(0),
                                 m_iCols(0),
                                 m_cOpenCL(NULL)
  {
   m_adParams = VECTOR::Ones(2);
   m_adParams[1] = 0;
  }
```

Обратите внимание, что при расчете производной некоторых функций активации достаточно значения самой функции активации. В других случаях нам потребуются значения до функции активации. Поэтому введем два указателя на буферы соответствующих данных:

- m_cInputs
- m_cOutputs

В теле данного класса создадим лишь один экземпляр буфера, а во второй переменной сохраним указатель на буфер вызывающего нейронного слоя. В связи с этим в деструкторе класса удалим только один объект.

```
CActivation::~CActivation(void)
  {
   if(!!m_cInputs)
      delete m_cInputs;
  }
```

В методе инициализации класса сохраним полученные параметры функции активации и создадим объект буфера данных. Надо сказать, что на данном этапе мы лишь создаем экземпляр класса, но инициализируем сам буфер, так как мы не знаем еще размеры необходимого буфера данных.

```
bool CActivation::Init(VECTOR &params)
  {
   m_adParams = params;
//---
   m_cInputs = new CBufferType();
   if(!m_cInputs)
      return false;
//---
   return true;
  }
```

В методе считывания параметров активации нет никаких подводных камней. Мы лишь возвращаем значение переменных.

```
ENUM_ACTIVATION_FUNCTION CActivation::GetFunction(VECTOR &params)
  {
   params = m_adParams;
   return GetFunction();
  }
```

Метод вычисления значений функции активации *Activation* в параметрах получает указатель на буфер результатов нейронного слоя. В данном буфере содержатся данные работы нейрона до функции активации. Нам необходимо активизировать полученные значения и перезаписать их в указанный буфер. Но мы же помним, что полученные значения могут понадобиться при вычислении производных некоторых функций. Поэтому мы осуществляем «жонглирование» указателями на объекты буферов, сохранив полученный указатель в переменной *m_cInputs*. В полученную в параметрах переменную и переменную *m_cOutputs* сохраним буфер из переменной *m_cInputs*. Текущий класс соответствует отсутствию функции активации, поэтому мы и не осуществляем никаких операций над полученными данными.

Однако здесь есть один нюанс. Так как мы не осуществляем никаких операций над полученными данными, то нам необходимо вернуть их вызывающей программе. При этом мы уже подменили буфер, который вернем. Поэтому мы проверяем используемую функцию активации, и если дальнейших действий над полученными данными не требуется, то вернем назад указатель на буфер и удалим лишний объект.

Казалось бы, получилось много ненужных действий в методе, который никоим образом не изменил данных. Но это наши небольшие инвестиции в функционал наследующих классов.

```
bool CActivation::Activation(CBufferType *&output)
  {
   if(!output || output.Total() <= 0)
      return false;
   m_cOutputs = m_cInputs;
   m_cInputs = output;
   output = m_cOutputs;
   if(GetFunction() == AF_NONE && output != m_cInputs)
     {
      delete output;
      output = m_cInputs;
     }
//---
   return true;
  }
```

В то же время метод вычисления производной функции активации в данном классе останется номинальным. Во всех случаях он будет возвращать положительное значение.

Метод активизации функционала многопоточных вычислений *SetOpenCL* в параметрах получает указатель на объект работы с контекстом *OpenCL* и размеры буфера результатов вызывающего нейронного слоя. Данные размеры нам понадобятся для инициализации и создания буфера в контексте.

В теле метода мы сохраняем полученные размеры и указатель, после чего инициализируем буфер данных указанного размера нулевыми значениями и создаем буфер в контексте *OpenCL*.

```
bool CActivation::SetOpenCL(CMyOpenCL *opencl, const ulong rows, const ulong cols)
  {
   m_iRows = rows;
   m_iCols = cols;
   if(m_cOpenCL != opencl)
     {
      if(m_cOpenCL)
         delete m_cOpenCL;
      m_cOpenCL = opencl;
     }
//---
   if(!!m_cInputs)
     {
      if(!m_cInputs.BufferInit(m_iRows, m_iCols, 0))
         return false;
      m_cInputs.BufferCreate(m_cOpenCL);
     }//---
   return(!!m_cOpenCL);
  }
```

Как видите, методы класса довольно просты. Нам осталось рассмотреть лишь методы работы с файлами. Их алгоритм также несложен. В теле метода сохранения данных *Save* мы как обычно проверяем хендл файла для записи данных, который получаем в параметрах, и сохраняем тип функции активации и значение параметров.

```
bool CActivation::Save(const int file_handle)
  {
   if(file_handle == INVALID_HANDLE)
      return false;
   if(FileWriteInteger(file_handle, Type()) <= 0 ||
      FileWriteInteger(file_handle, (int)GetFunction()) <= 0 ||
      FileWriteInteger(file_handle, (int)m_iRows) <= 0 ||
      FileWriteInteger(file_handle, (int)m_iCols) <= 0 ||
      FileWriteDouble(file_handle, (double)m_adParams[0]) <= 0 ||
      FileWriteDouble(file_handle, (double)m_adParams[1]) <= 0)
      return false;
//---
   return true;
  }
```

Метод загрузки данных Load в параметрах тоже получает хендл файла. В теле метода мы проверяем действительность полученного хендла и считываем значения констант. После этого инициализируем один буфер данных. При этом не забываем контролировать процесс выполнения операций.

```
bool CActivation::Load(const int file_handle)
  {
   if(file_handle == INVALID_HANDLE)
      return false;
   m_iRows = (uint)FileReadInteger(file_handle);
   m_iCols = (uint)FileReadInteger(file_handle);
   m_adParams.Init(2);
   m_adParams[0] = (TYPE)FileReadDouble(file_handle);
   m_adParams[1] = (TYPE)FileReadDouble(file_handle);
//---
   if(!m_cInputs)
     {
      m_cInputs = new CBufferType();
      if(!m_cInputs)
         return false;
     }
   if(!m_cInputs.BufferInit(m_iRows, m_iCols, 0))
      return false;
//---
   return true;
  }
```

Мы рассмотрели все методы базового класса функции активации *CActivation*. В целом получили следующую структуру класса.

```
class CActivation : protected CObject
  {
protected:
   ulong             m_iRows;
   ulong             m_iCols;
   VECTOR            m_adParams;
   CMyOpenCL*        m_cOpenCL;
   //---
   CBufferType*      m_cInputs;
   CBufferType*      m_cOutputs;

public:
                     CActivation(void);
                    ~CActivation(void) {if(!!m_cInputs) delete m_cInputs; }
   //---
   virtual bool      Init(VECTOR &params);
   virtual ENUM_ACTIVATION_FUNCTION  GetFunction(VECTOR &params);
   virtual ENUM_ACTIVATION_FUNCTION   GetFunction(void) { return AF_NONE; }
   virtual bool      Activation(CBufferType*& output);
   virtual bool      Derivative(CBufferType*& gradient) { return true;    }
   //---
   virtual bool      SetOpenCL(CMyOpenCL *opencl, const ulong rows,
                                                  const ulong cols);

   //--- методы работы с файлами
   virtual bool      Save(const int file_handle);
   virtual bool      Load(const int file_handle);
   //--- метод идентификации объекта
   virtual int       Type(void)             const { return defActivation; }
  };
```

Но как мы и говорили ранее, в данном классе заложен лишь интерфейс будущих классов различных функций активации. Чтобы добавить непосредственный алгоритм функции активации, необходимо создать новый класс с переопределением нескольких методов. К примеру, создадим класс линейной функции активации. Структура такого класса приведена ниже.

```
class CActivationLine   :  public CActivation
  {
public:
                     CActivationLine(void) {};
                    ~CActivationLine(void) {};
   //---
   virtual ENUM_ACTIVATION_FUNCTION   GetFunction(void) override
                                                 { return AF_LINEAR; }
   virtual bool      Activation(CBufferType*& output) override;
   virtual bool      Derivative(CBufferType*& gradient) override;
  };
```

Как можно заметить, новый класс *CActivationLine* публично наследуется от выше созданного базового класса функции активации *CActivation*. Конструктор и деструктор класса пустые. Все, что нам надо, так это переопределить три метода:

- *GetFunction* — получение используемой функции активации и ее параметров;

- *Activation* — вычисление значения функции активации по исходному значению;
- *Derivative* — производная от функции активации.

В методе *GetFunction* мы лишь меняем тип возвращаемой функции активации на соответствующий данному классу.

Метод *Activation* в параметрах получает указатель на буфер исходных данных аналогично методу родительского класса. В теле метода мы не проверяем полученный указатель, а просто вызываем метод родительского класса, в котором, как вы помните, осуществляется проверка полученного указателя и «жонглирование» указателями на буфера данных. После этого осуществляется разделение алгоритма на два потока: с использованием технологии *OpenCL* и без. С многопоточными операциями мы познакомимся немного позже, а в блоке операций без использования многопоточных операций мы лишь вызываем функцию активации для матрицы полученных значений с указанием типа функции активации *AF_LINEAR* и параметров функции.

```
bool CActivationLine::Activation(CBufferType*& output)
  {
   if(!CActivation::Activation(output))
      return false;
//---
   if(!m_cOpenCL)
     {
      if(!m_cInputs.m_mMatrix.Activation(output.m_mMatrix, AF_LINEAR,
                                         m_adParams[0], m_adParams[1]))
         return false;
     }
   else // блок OpenCL
     {
      return false;
     }
//---
   return true;
  }
```

В методе вычисления производной все еще прозаичней. В параметрах метод получает указатель на объект градиента ошибки. Полученные значения надо скорректировать на производную функции активации. Как вы знаете, производной линейной функции является ее коэффициент при переменной. Поэтому единственное, что нам надо сделать, это умножить полученный вектор градиентов на параметр функции активации с индексом 0.

```
bool CActivationLine::Derivative(CBufferType*& gradient)
  {
   if(!m_cInputs || !m_cOutputs ||
      !gradient || gradient.Total() < m_cOutputs.Total())
      return false;
//---
   if(!m_cOpenCL)
     {
      gradient.m_mMatrix = gradient.m_mMatrix * m_adParams[0];
     }
   else // блок OpenCL
     {
      return false;
     }
//---
   return true;
  }
```

Как можно заметить, механизм описания новой функции активации довольно несложен. Давайте аналогичным образом создадим класс для использования ReLU в качестве функции активации.

```
class CActivationLReLU : public CActivation
  {
public:
                     CActivationLReLU(void) { m_adParams[0] = (TYPE)0.3; };
                    ~CActivationLReLU(void) {};
   //---
   virtual ENUM_ACTIVATION_FUNCTION   GetFunction(void) override { return AF_LRELU; }
   virtual bool      Activation(CBufferType*& output) override;
   virtual bool      Derivative(CBufferType*& gradient) override;
  };
```

В функции активации нового класса мы также воспользуемся вызовом матричной функции активации с указанием соответствующего типа функции AF_LRELU.

```
bool CActivationLReLU::Activation(CBufferType*& output)
  {
   if(!CActivation::Activation(output))
      return false;
//---
   if(!m_cOpenCL)
     {
      if(!m_cInputs.m_mMatrix.Activation(output.m_mMatrix, AF_LRELU,m_adParams[0]))
         return false;
     }
   else // блок OpenCL
     {
      return false;
     }
//---
   return true;
  }
```

Аналогичный подход используем и в методе производной функции активации.

```
bool CActivationLReLU::Derivative(CBufferType*& gradient)
  {
   if(!m_cOutputs || !gradient ||
      m_cOutputs.Total() <= 0 || gradient.Total() < m_cOutputs.Total())
      return false;
//---
   if(!m_cOpenCL)
     {
      MATRIX temp;
      if(!m_cInputs.m_mMatrix.Derivative(temp, AF_LRELU,m_adParams[0]))
         return false;
      gradient.m_mMatrix *= temp;
     }

   else // блок OpenCL
     {
      return false;
     }
//---
   return true;
  }
```

Возможно, у читателя возникнет резонный вопрос, для чего нам создавать новые классы если мы используем матричные функции активации, заложенные в языке *MQL5*. Это сделано большей степенью для выработки единого подхода при использовании многопоточных технологий OpenCL и без них. В данные методы добавится код организации многопоточных вычислений в контексте *OpenCL*, а использование описанных классов позволяет сделать единый вызов алгоритмов для вычислений функции активации как средствами *MQL5*, так и с использованием многопоточных вычислений в контексте OpenCL.

3.7 Организация параллельных вычислений средствами OpenCL

В предыдущих главах мы уже познакомились с организацией работы полносвязного нейронного слоя средствами MQL5. Напомню, что в своей реализации мы использовали матричные операции для умножения вектора исходных данных на матрицу весовых коэффициентов. От одного нейронного слоя к другому сигнал идет последовательно, и мы не можем начать операции на последующем нейронном слое до полного завершения операций на предыдущем. В отличие от этого, результаты операций одного нейрона внутри слоя полностью не зависят от выполнения операций с другими нейронами этого же нейронного слоя. Следовательно, мы можем сократить затраты времени на обработку одного нейронного слоя, если сможем организовать параллельные вычисления. Чем больше будем обрабатывать нейронов одновременно, тем меньше составят временные затраты на обработку одного сигнала и обучение нейронной сети в целом.

Как мы уже обсуждали ранее, организовать параллельные вычисления нам поможет технология *OpenCL*. Конечно, это потребует от нас дополнительной работы по настройке процесса. Давайте подумаем, какие процессы мы будем переносить в *OpenCL*, чтобы это было максимально эффективно. Напомню, что из-за накладных расходов времени на передачу данных между устройствами реальный прирост производительности мы сможем получить только при большом количестве одновременных потоков операций.

Первое, что можно перенести, — это вычисление операций прямого прохода. Мы можем перенести в область параллельных вычислений выполнение операций по каждому отдельному нейрону. Сначала посчитаем взвешенную сумму входного сигнала для каждого нейрона, а потом по каждому нейрону вычислим функцию активации.

Операции обратного прохода мы тоже можем перенести в мир параллельных вычислений. Разберем этапы обратного прохода.

Отклонение расчетных значений от эталонных на выходном слое нейронной сети легко разделить на отдельные потоки по каждому нейрону.

Далее мы также можем по каждому нейрону скорректировать полученное отклонение на производную функции активации. В результате такой операции мы получим градиент ошибки перед функцией активации нейрона.

Следуя за процессом обратного прохода, на следующем этапе нам нужно распределить полученный градиент ошибки на нейроны предыдущего слоя. В полносвязном нейронном слое все нейроны предыдущего слоя имеют связи со всеми нейронами последующего слоя — в каждом значении вектора градиентов ошибки есть некая составляющая от каждого нейрона предыдущего слоя. Тут напрашивается два, казалось бы, равнозначных подхода:

- Мы можем создать потоки по каждому элементу вектора градиентов ошибки, в каждом потоке перебрать все нейроны предыдущего слоя и прибавить значение своей составляющей градиента ошибки.
- Можем наоборот разделить потоки по каждому нейрону предыдущего слоя и собрать составляющие градиента ошибки от предыдущего слоя.

При кажущейся равнозначности подходов первый вариант имеет несколько недостатков. Так как нам предстоит складывать составляющие градиентов ошибки от разных нейронов последующего слоя, то перед началом операций необходимо обнулить текущее значении вектора. Это дополнительные затраты времени и ресурсов. Кроме того, есть и технические моменты. Работа с глобальной памятью медленнее работы с приватной памятью потока, поэтому нам желательно собрать значения в быстрой памяти и один раз перенести в глобальную. Самое проблемное

место этого подхода в том, что в процессе работы велика вероятность возникновения ситуации, когда несколько потоков одновременно будут пытаться записать значения в один нейрон предыдущего слоя. А это крайне не желательно для нас.

По совокупности вышеизложенных факторов второй вариант становится более привлекательным для реализации.

Разделение на потоки следующих двух процессов (расчет дельт для корректировки весовых коэффициентов и непосредственно обновление матрицы весовых коэффициентов) не вызывает никаких вопросов, ведь каждый весовой коэффициент участвует только в одной связи двух нейронов и не влияет на остальные.

3.7.1 Создание программы OpenCL

Работу по переносу вычислений мы начнем с создания программы *OpenCL*. Выбор такого подхода весьма очевиден. Мы уже организовали процесс средствами MQL5. Следовательно, весь процесс операций нам ясен и понятен. Операции вычислений будут осуществляться в программе *OpenCL*. В основной программе нам предстоит организовать процесс передачи данных и вызов программы *OpenCL*. Последний процесс проще организовать, когда нам уже будет известно, какие данные и в какой *kernel* необходимо передать.

Код программы запишем в отдельный файл *opencl_program.cl*. Имя и расширения файла может быть любым, так как в последующем мы будем подгружать ее в коде основной программы в качестве ресурса. Я же использую расширение *.cl как общепринятое расширение для обозначения программ *OpenCL*. В целом использование стандартных расширений облегчает чтение проектов со сложной структурой файлов.

Для прямого прохода по аналогии с реализацией *MQL5* создадим *kernel PerceptronFeedForward*. Для проведения операций нам потребуются в качестве исходных данных вектор результатов предыдущего слоя (*inputs*) и матрица весовых коэффициентов (*weights*). Результат операций запишем в вектор результатов (*outputs*). Кроме массивов данных нам потребуются количество нейронов в предыдущем слое для контроля выхода за пределы массива.

Как мы уже обсудили ранее, количество потоков будет соответствовать количеству нейронов в слое. Поэтому в начале кернела мы вызовем функции *get_global_id*, которая вернет нам идентификатор потока. Воспользуемся этим значением в качестве порядкового номера обрабатываемого нейрона.

Матрица весовых коэффициентов в нашем буфере представляется в виде вектора, в котором последовательно идут *N* весовых коэффициентов первого нейрона, за ними — *N* весовых коэффициентов второго нейрона и так далее. *N* на 1 элемент больше количества нейронов в предыдущем слое, так как мы используем элемент *bias*-смещения. У нас уже есть порядковый номер нейрона и количество нейронов в предыдущем слое, поэтому мы можем определить смещение в векторе весовых коэффициентов до первого весового коэффициента нашего нейрона.

Далее создадим локальную переменную для сбора накопительной суммы произведений и инициируем ее значением коэффициента *bias*-смещения. После этого организовываем цикл и считаем сумму произведений исходного сигнала на весовые коэффициенты для конкретного нейрона нашего потока. Здесь следует обратить внимание, что в *OpenCL* существует поддержка векторных операций. Это особый вид операций, который позволяет микропроцессору за один цикл производить одну операцию сразу над несколькими значениями. В предложенной реализации я использовал векторы типа *double4*. Это вектор из четырех элементов типа *double*.

При этом для перевода данных из массива дискретных значений в вектор была создана функция *ToVect4*, которую мы рассмотрим немного позже. Для получения суммы произведений я использовал встроенную функцию *dot*. Она относится к векторным операциям и позволяет получить дискретное значение произведения двух векторов. Это позволило нам использовать в цикле шаг 4 и тем самым в 4 раза сократить количество итераций.

Следует отметить, что *double4* не единственный векторный тип данных, поддерживаемый *OpenCL*. При этом возможность их использования зависит от технических характеристик используемого оборудования. Тип *double4*, по моему личному мнению, является наиболее универсальным для использования на широком круге доступного оборудования. Создавая свои библиотеки, вы можете использовать другой тип данных, наиболее оптимальный для вашего оборудования.

После завершения итераций цикла мы сохраним накопленную сумму векторного произведения в соответствующий элемент буфера результатов.

Полный код кернела представлен ниже.

```
__kernel void PerceptronFeedForward(__global TYPE *inputs,
                                    __global TYPE *weights,
                                    __global TYPE *outputs,
                                    int inputs_total)
  {
   const int n = get_global_id(0);
   const int weights_total = get_global_size(0) * (inputs_total + 1);
   int shift = n * (inputs_total + 1);
   TYPE s = weights[shift + inputs_total];
   for(int i = 0; i < inputs_total; i += 4)
      s += dot(ToVect4(inputs, i, 1, inputs_total, 0),
               ToVect4(weights, i, 1, weights_total, shift));
   outputs[n] = s;
  }
```

Посмотрим на алгоритм функции *ToVect4*. В параметрах функция получает указатель на вектор дискретных значений, позицию первого элемента для копирования, шаг между двумя элементами для копирования, размер массива данных и сдвиг в массиве до первого копируемого элемента. Параметры позиции первого элемента и сдвиг имеют схожую функцию, но различны по контексту операций. Сдвиг определяет смещение в векторе данных от элемента с индексом 0. В случае функции прямого прохода это смещение до первого весового коэффициента обрабатываемого нейрона. В позиции первого элемента для копирования указывается элемент без учета шага между элементами. В указанном примере это номер нейрона предшествующего слоя. При рассмотрении примера с шагом в один элемент легко выразить один параметр через другой. Разница в использовании параметров будет более заметна при рассмотрении обратного прохода, когда шаг для копирования весовых коэффициентов будет равен размеру вектора весовых коэффициентов одного нейрона.

В начале функции инициализируем вектор результатов нулевыми значениями и проверим, чтобы шаг был не меньшее одного элемента. Затем проверим, сколько элементов мы можем взять из исходного массива данных от начальной позиции с учетом сдвига и шага. Эта операция необходима для предотвращения обращений за пределами массива. Далее заполняем векторы результатов доступными значениями, при этом недостающие элементы остаются нулевыми. Таким образом, размер исходного массива станет кратным четырем. При этом конечное значение вызывающих функций останется неизменным, а использование векторных операций позволит в целом сократить затраты времени на операции.

```
TYPE4 ToVect4(__global TYPE *array, int start, int step, int size, int shift)
  {
   TYPE4 result = (TYPE4)(0, 0, 0, 0);
   step = max(1, step);
   int st = start * step + shift;
   if(st < size)
     {
      int k = (size - shift + step - 1)  /  step;

      switch(k)
        {
         case 0:
            break;
         case  1:
            result = (TYPE4)(array[st], 0, 0, 0);
            break;
         case  2:
            result = (TYPE4)(array[st], array[st + step], 0, 0);
            break;
         case  3:
            result = (TYPE4)(array[st], array[st + step], array[st + 2 * step], 0);
            break;
         default:
         result = (TYPE4)(array[st], array[st + step], array[st + 2 * step], array[st
               break;
        }
     }
   return result;
  }
```

Для полного рассмотрения прямого прохода нам остается ознакомиться с функциями активации. В целом они повторяют аналогичные реализации в *MQL5*, за исключением оформления в виде кернела. К примеру, ниже представлен код реализации сигмоиды. В параметрах кернела подаются указатели на буферы исходных данных и результатов, а также параметры функции активации. В теле кернела мы сначала определяем идентификатор потока, который указывает на порядковый номер обрабатываемого элемента в буфере данных, а затем организовываем процесс вычислений функции активации. Как можно заметить, код вычисления значения функции сильно напоминает аналогичный код на *MQL5*, представленный в разделе описания функции активации.

```
__kernel void SigmoidActivation(__global TYPE* inputs,
                                __global TYPE* outputs,
                                const TYPE a, const TYPE b)
  {
   size_t i = get_global_id(0);
   outputs[i] = a / (1 + exp(-inputs[i])) - b;
  }
```

То же можно сказать и о реализации функции активации *Swish*.

```
__kernel void SwishActivation(__global TYPE* inputs,
                              __global TYPE* outputs,
                              const TYPE b)
  {
   size_t i = get_global_id(0);
   TYPE value = inputs[i];
   outputs[i] = value / (1 + exp(-b * value));
  }
```

Но есть некоторые сложности при реализации функции *Softmax*. Это связано со сложностью передачи данных между потоками для подсчета суммы всех значений вектора экспоненты в нейронном слое. Для решения вопроса было решено разделить процесс на несколько этапов. Вдобавок мы воспользуемся возможностью Work-group использовать общие массивы в локальной памяти.

В параметрах кернела, как и ранее, мы передаем указатели на буферы исходных данных и результатов и размер буфера исходных данных. В теле кернела мы сначала получаем все необходимы идентификаторы. Это глобальные идентификаторы в двух измерениях и идентификатор потока в локальной группе. О необходимости использования двух измерений в глобальном пространстве задач мы поговорим позже.

```
__kernel void SoftMaxActivation(__global TYPE* inputs,
                                __global TYPE* outputs,
                                const ulong total)
  {
   uint i = (uint)get_global_id(0);
   uint l = (uint)get_local_id(0);
   uint h = (uint)get_global_id(1);
   uint ls = min((uint)get_local_size(0), (uint)LOCAL_SIZE);
```

Затем мы создадим локальный массив, в котором каждому потоку предоставим один элемент для суммирования экспоненциальных значений.

Обратите внимание, что *OpenCL* не предоставляет возможности создания динамических массивов. Поэтому размер локального массива должен быть определен до компиляции программы. Значит, нам надо искать некий компромисс. Излишний размер ведет к неэффективному использованию памяти. А слишком малый размер массива ограничивает количество активных параллельных потоков. Конечно, решать такую задачу гораздо легче, когда известны параметры используемого устройства и архитектура модели. Следовательно, нам нужен механизм, позволяющий легко и быстро изменить данный параметр перед непосредственным компилированием программы. И для этого нет ничего лучше, как воспользоваться макроподстановкой. Точно также, как мы сделали для типа данных. В коде мы указываем константу *LOCAL_SIZE*, значение которой мы присваиваем в нашем файле констант *defines.mqh*.

Далее мы организовываем цикл, в котором каждый поток суммирует свою часть экспоненциальных значений. И записывает полученное значение в соответствующий элемент локального массива.

```
   __local TYPE temp[LOCAL_SIZE];
   uint count = 0;
   for(count = l; (count < total && l < ls); count += ls)
     {
      uint shift = h * total + count;
      temp[l] = (count > l ? temp[l] : 0) + exp(inputs[shift]);
     }
   barrier(CLK_LOCAL_MEM_FENCE);
```

После завершения операций цикла мы устанавливаем *barrier*, который позволяет синхронизировать все потоки локальной группы. Данная операция приостанавливает выполнение каждого потока в ожидание завершения итераций цикла всех потоков локальной группы.

После получения нескольких отдельных сумм, которые посчитал каждый отдельный поток, необходимо свести их в единую сумму. Для этого мы организуем еще один цикл. В его теле мы будем попарно суммировать значения локального массива. Фокус в том, что мы разделим весь локальный массив на две одинаковые части. Первая половина активных потоков прибавит к своему значению в локальном массиве значение из второй половины. На следующей итерации цикла мы еще уменьшим вдвое количество активных потоком и суммируем уже значения, полученные на предыдущей итерации цикла. Цикл повторяется до тех пор, пока в первом элементе массива не соберется сумма всех элементов.

Здесь мы вставляем *barrier* в теле цикла, так как до начала каждой последующей итерации необходимо окончание предыдущей итерации всеми потоками.

```
   count = ls;
   do
     {
      count = (count + 1) / 2;
      temp[l] += (l < count && (l + count) < ls ? temp[l + count] : 0);
      barrier(CLK_LOCAL_MEM_FENCE);
     }
   while(count > 1);
```

После получения суммы экспонент всех значений буфера мы можем посчитать значение каждого элемента после функции активации. Это мы и сделаем в следующем цикле.

```
//---
   TYPE sum=temp[0];
   for(count = l; count < total; count += ls)
     {
      uint shift = h * total + count;
      outputs[shift] = exp(inputs[shift]) / (sum + 1e-37f);
     }
  }
```

За прямым проходом идет обратный проход. Начинается он с определения градиентов ошибки на выходе нейронного слоя. Эту функцию выполняет кернел *CalcOutputGradient*. В параметрах кернел получает указатели на три вектора: первые два вектора целевых значений и результатов прямого прохода являются исходными данными для функции, а третий служит для записи результатов вычислений. Также в параметрах указывается используемая функция потерь. Код кернела полностью повторяет алгоритм ранее рассмотренного аналогичного метода,

написанного средствами *MQL5*. В нем мы также видим разветвление алгоритма в зависимости от используемой функции потерь.

```
__kernel void CalcOutputGradient(__global TYPE *target,
                                 __global TYPE *outputs,
                                 __global TYPE *gradients,
                                 int loss_function)
  {
   const int n = get_global_id(0);
   switch(loss_function)
     {
      case 0:
         gradients[n] = target[n] - outputs[n];
         break;
      case 1:
         gradients[n] = 2 * (target[n] - outputs[n]);
         break;
      case 2:
         gradients[n] = -target[n] /
                        (outputs[n] + 1e-37f) * log(outputs[n] + 1e-37f);
         break;
      case 3:
         gradients[n] = (target[n] - outputs[n]) /
                        (outputs[n] * (outputs[n] - 1) + 1e-37f);
         break;
      default:
         gradients[n] = target[n] - outputs[n];
         break;
     }
  }
```

Следующим этапом нашего процесса обратного прохода идет корректировка градиента ошибки на производную функции активации. Напомню, что функцию активации и все связанные с ней процессы мы вынесли в отдельный класс. Более того, каждая функция активации имеет свой класс. Для каждой функции активации мы создадим отдельный кернел. Аналогично мы создадим отдельный кернел для определения производной каждой функции активации.

При создании кернелов вычисления производных функций мы учитываем особенности каждой из них. К примеру, производной линейной функции активации всегда будет ее параметр *a*, и я не вижу смысла выносить это в отдельный кернел. Для корректировки градиента ошибки на производную данной функции мы можем воспользоваться кернелом прямого прохода, указав 0 вместо параметра *b*.

Похожая ситуация складывается при использовании LReLU. Здесь также используется линейная зависимость, но коэффициент линейности меняется от значения до функции активации. Поэтому надо создать новый кернел, который в параметрах будет получать указатели на три буфера данных и коэффициент утечки, в то время как кернел прямого прохода получал указатели только на два буфера и коэффициент.

В теле кернела мы, как всегда, определяем глобальный идентификатор потока. Он укажет нам обрабатываемый элемент в буферах данных. Затем мы проверяем значение соответствующего элемента до функции. Если число больше 0, то используем коэффициент 1. В противном случае воспользуемся коэффициентом утечки. На выбранный коэффициент умножим градиент ошибки,

полученный от последующего нейронного слоя. Значение операции запишем в соответствующий элемент буфера результатов.

```
__kernel void LReLUDerivative(__global TYPE* outputs,
                              __global TYPE* output_gr,
                              __global TYPE* input_gr,
                              const TYPE a)
  {
   size_t i = get_global_id(0);
   input_gr[i] = (outputs[i] > 0 ? (TYPE)1 : a) * output_gr[i];
  }
```

Значение производных *S*-образных функций, таких как сигмоида и гиперболический тангенс, легко посчитать через результат функции активации.

$$\frac{\partial\,\sigma(x)}{\partial x} = \sigma(x)(1-\sigma(x))$$

$$\frac{\partial \tanh(x)}{\partial x} = 1 - tanh^2(x)$$

Именно такой подход мы и заложим в алгоритм кернелов вычисления производных указанных функций. Общие же подходы к организации работы кернелов остаются прежние.

```
__kernel void SigmoidDerivative(__global TYPE* outputs,
                                __global TYPE* output_gr,
                                __global TYPE* input_gr,
                                const TYPE a, const TYPE b
                               )
  {
   size_t i = get_global_id(0);
   if(a == 0)
      input_gr[i] = 0;
   else
     {
      TYPE z = clamp(outputs[i] + b, (TYPE)0, a);
      input_gr[i] = z * (1 - z / a) * output_gr[i];
     }
  }

__kernel void TanhDerivative(__global TYPE* outputs, __global TYPE* output_gr,
                             __global TYPE* input_gr)
  {
   size_t i = get_global_id(0);
   input_gr[i] = (1 - pow(outputs[i], 2)) * output_gr[i];
  }
```

Для вычисления производной функции Swish нам уже потребуются значения как до активации, так и после. Математическая формула производной представлена ниже.

$$\frac{\partial f(x)}{\partial x} = \beta f(x) + \sigma(\beta x)(1-\beta f(x))$$

Как можно заметить, производная функции выражается через значение функции активации, значения сигмоиды и параметра функции активации β. Подставив формулу сигмоиды, получим нижеследующее выражение.

$$\frac{\partial f(x)}{\partial x} = \beta f(x) + \frac{1 - \beta f(x)}{1 + e^{-\beta x}}$$

Для реализации указанной формулы нам необходимо передать в кернел указатели на четыре буфера данных: соответствующие значения до и после активации, градиент ошибки от последующего слоя и буфер результатов.

В теле кернела мы вычислим значение производной функции по приведенной формуле и умножим полученное значение на градиент ошибки, полученный от последующего слоя. Результат операции запишем в соответствующий элемент буфера результатов.

```
__kernel void SwishDerivative(__global TYPE* outputs,
                              __global TYPE* output_gr,
                              __global TYPE* input_gr,
                              const TYPE b,
                              __global TYPE* inputs)
  {
   size_t i = get_global_id(0);
   TYPE by = b * outputs[i];
   input_gr[i] = (by + (1 - by) / (1 + exp(-b * inputs[i]))) * output_gr[i];
  }
```

Наиболее сложным выглядит кернел вычисления производной функции *Softmax*. Как и в случае *S*-образных функций, для вычисления производной функции *Softmax* достаточно значений самой функции активации. Только, как вы помните, для вычисления значения одного элемента вектора при прямом проходе мы использовали значения всех элементов вектора до функции активации (вычисление общей суммы экспонент). Следовательно, значение каждого элемента после активации зависит от всех элементов вектора до активации. А значит, каждый элемент до активации должен получить свою долю ошибки с каждого элемента на выходе нейронного слоя. В общем виде производная функции *Softmax* вычисляется по формуле ниже.

$$\frac{df(x_i)}{dx_i} = \begin{cases} i = j & f(x_i) * (1 - f(x_i)) \\ i \neq j & -f(x_j) f(x_i) \end{cases}$$

В параметрах кернела *SoftMaxDerivative* мы будем передавать указатели на три буфера данных: значения функции активации, градиент ошибки от последующего слоя и буфер результатов.

В теле кернела мы определяем глобальный идентификатор потока, который на этот раз нам указывает только элемент буфера результатов. Глобальный идентификатор потока во втором измерении используется при работе с матрицей, в которой функция *Softmax* применялась по строкам. В таком случае данный идентификатор поможет нам определить смещение до анализируемых данных.

Далее подготовим две частные переменные: в одной сохраним значение соответствующего элемента после активации, во второй будем накапливать суммарное значение градиентов ошибки.

После этого организуем цикл сбора градиентов ошибки со всех элементов на выходе нейронного слоя. Каждый конкретный градиент вычисляется по приведенной выше формуле.

После завершения итераций цикла накопленную сумму градиентов сохраним в соответствующий элемент буфера результатов.

```
__kernel void SoftMaxDerivative(__global TYPE* outputs,
                                __global TYPE* output_gr,
                                __global TYPE* input_gr)
  {
   size_t i = get_global_id(0);
   size_t outputs_total = get_global_size(0);
   size_t shift = get_global_id(1) * outputs_total;
   TYPE output = outputs[shift + i];
   TYPE result = 0;
   for(int j = 0; j < outputs_total; j++)
      result += outputs[shift + j] * output_gr[shift + j] *
                                    ((TYPE)(i == j ? 1 : 0) - output);
   input_gr[shift + i] = result;
  }
```

После корректировки градиента ошибки на производную функции активации нам предстоит распределить полученные значения на нейроны предыдущего нейронного слоя. Как уже было сказано выше, здесь мы разделим потоки по количеству нейронов в нижнем нейронном слое. Для каждого нейрона будем собирать градиенты со всех зависимых от него нейронов.

Распределение градиента ошибки через нейронный слой будет осуществляться в кернеле *CalcHiddenGradient*. На вход кернелу в параметрах подаются указатели на 3 массива:

- *weights* — матрица весовых коэффициентов;
- *gradients* — массив градиентов, скорректированных на производную функции активации;
- *gradient_inputs* — массив для записи градиентов предшествующего слоя.

Кроме того, в параметрах указывается количество нейронов в верхнем слое (размер массива *gradients*). Алгоритм построения кернела сильно напоминает метод прямого прохода — мы так же используем функции *dot* и *ToVect4*. Отличие в используемых массивах: если при прямом проходе мы брали входной сигнал и умножали на весовые коэффициенты, то сейчас мы градиент ошибки умножаем на весовые коэффициенты. Есть еще один момент в использовании функции *ToVect4* для матрицы весовых коэффициентов. Когда мы рассматривали эту функцию для прямого прохода, то говорили о схожей функции параметров первого элемента для копирования *start* и сдвига *shift*. Тогда мы использовали шаг в 1 элемент. Сейчас, перебирая элементы по массиву градиентов, мы будем подбирать соответствующие весовые коэффициенты. Но если при прямом проходе у нас нейроны и весовые коэффициенты шли по порядку, то при обратном проходе мы берем весовые коэффициенты поперек матрицы весов. В векторном выражении матрицы весов мы будем использовать шаг между двумя элементами для копирования на 1 элемент больше количества нейронов в предыдущем слое (элемент смещения *bias*). В то же время сдвиг будет равен порядковому номеру обрабатываемого нейрона нижнего слоя.

Можно заметить, что в параметрах мы не указываем количество нейронов в нижнем нейронном слое, но используем данное значение в шаге для считывания значений из матрицы весов. Получить указанное значение нам позволяет функция *get_global_size*, которая возвращает общее количество запущенных потоков нашего кернела. А так как мы запускали по одному потоку для каждого нейрона предшествующего слоя, то количество потоков в данном случае будет соответствовать количеству нейронов в слое. Здесь же мы считаем количество элементов в матрице весов путем умножения количества нейронов в слое на количество нейронов в предыдущем слое плюс элемент *bias*.

В остальном, мы также используем векторные операции, которые позволяют нам использовать цикл с шагом в 4 элемента.

```
__kernel void CalcHiddenGradient(__global TYPE *gradient_inputs,
                                 __global TYPE *weights,
                                 __global TYPE *gradients,
                                 int outputs_total)
  {
   const int n = get_global_id(0);
   const int inputs_total = get_global_size(0);
   int weights_total = (inputs_total + 1) * outputs_total;
//---
   TYPE grad = 0;
   for(int o = 0; o < outputs_total; o += 4)
      grad += dot(ToVect4(gradients, o, 1, outputs_total, 0),
                  ToVect4(weights, o, (inputs_total + 1), weights_total, n));
   gradient_inputs[n] = grad;
  }
```

Если посмотреть на алгоритм обратного распространения ошибки, то мы уже вышли на финишную прямую. После распространения градиента ошибки нам остается обновить матрицу весов. Но мы помним из реализации на MQL5, что матрица весов не будет обновляться после каждой итерации. Как и в указанной реализации, процесс обновления матрицы весов мы разделим на 2 этапа:

1. Накопление градиентов ошибки на некотором интервале.
2. Усреднение накопленного градиента и корректировка матрицы весов.

Собирать градиенты ошибок для каждого весового коэффициента будем в кернеле *CalcDeltaWeights*. В параметрах кернелу передаются указатели на массив данных на выходе предыдущего слоя, массив градиентов и массив для накопления дельт необходимых корректировок весовых коэффициентов.

Все весовые коэффициенты рассчитываются независимо, поэтому мы можем запустить вычисление ошибки по каждому весовому коэффициенту в отдельном потоке. Чтобы структура потоков была более наглядной и понятной, создадим пространство задач в двух измерениях. Первое измерение будет равно количеству нейронов текущего слоя, а второе — количеству нейронов предыдущего слоя.

В теле кернела мы определим размерность матрицы весов по пространству задач и положение анализируемого элемента в этой матрице. После этого определим смещение элемента в буфере результатов.

Не будем забывать, что мы накапливаем градиент ошибки до наступления момента непосредственного обновления весовых коэффициентов. Поэтому прибавим к ранее накопленной сумме произведение соответствующего градиента ошибки на элемент результатов предыдущего слоя.

Напомню, что мы используем дополнительный элемент *bias*-смещения. Для этого элемента используется постоянное значение входящего элемента равное единице. Мы его не учли при создании пространства задач, но мы просто обязаны также накапливать градиент ошибки по нему. С математической точки зрения производная умножения на 1 равна единице. А значит, градиент ошибки для данного элемента равен градиенту ошибки перед функцией активации соответствующего нейрона. Чтобы не повторять итерацию записи градиента ошибки веса *bias*-

смещения, данную итерацию мы будем осуществлять только для потока с индексом 0 во втором измерении.

```
__kernel void CalcDeltaWeights(__global TYPE *inputs,
                               __global TYPE *delta_weights,
                               __global TYPE *gradients)
  {
   const int n = get_global_id(0);
   const int outputs_total = get_global_size(0);
   const int i = get_global_id(1);
   const int inputs_total = get_global_size(1);
//---
   TYPE grad = gradients[n];
   int shift = n * (inputs_total + 1);
   delta_weights[shift + i] = inputs[i] * grad + delta_weights[shift + i];
   if(i == 0)
      delta_weights[shift + inputs_total] += grad;
  }
```

Теперь нам остается организовать процесс обновления матрицы весовых коэффициентов. Мы изучили и уже реализовали в основной программе несколько методов обновления весовых коэффициентов. При реализации данного процесса средствами MQL5 мы создали диспетчерский метод, который перенаправлял логическую цепочку операций в метод, соответствующий выбранному методу обновления весовых коэффициентов. Уже внутри этих методов мы определяем устройство для выполнения операций. Для поддержания целостности такого подхода нам предстоит по аналогии с основной программой создать несколько кернелов с реализацией всех используемых методов оптимизации матрицы весов.

Все кернелы создавались с использованием единого подхода и потому имеют много общих моментов. Но все же есть и различия, обусловленные особенностями каждого метода. Начнем рассмотрение кернелов с метода стохастического градиентного спуска.

$$w_{il}^{j} = w_{il}^{j} - \alpha \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^{j}} grad_{i}^{j},$$

В параметрах кернелу передаем указатели на матрицу накопленных градиентов и саму матрицу весовых коэффициентов. Кроме указателей на матрицы, параметры кернела содержат общее количество элементов в матрице весов, размер пакета для усреднения, обучающий коэффициент и параметры регуляризации.

Как и прежде, будем использовать векторные операции, поэтому число потоков будет в четыре раза меньше размера матрицы весов. В теле кернела сначала определим смещение в массиве для рабочих элементов нашего потока и загрузим их в векторные переменные. При считывании накопленных дельт сразу разделим полученные значения на размер пакета, что даст нам среднее значение градиента. После этого скорректируем веса на коэффициенты регуляризации и среднее значение накопленных дельт с учетом обучающего коэффициента.

В заключение вернем полученные значения в матрицу весовых коэффициентов, а массив накопленных дельт обнулим.

```
__kernel void SGDUpdate(__global TYPE *delta_weights,
                        __global TYPE *weights,
                        int total,
                        int batch_size,
                        TYPE learningRate,
                        TYPE Lambda1,
                        TYPE Lambda2
                       )
  {
   int start = 4 * get_global_id(0);
   TYPE4 delta4 = ToVect4(delta_weights, start, 1, total, 0);
   TYPE4 weights4 = ToVect4(weights, start, 1, total, 0);
   TYPE lr = learningRate / ((TYPE)batch_size);
   weights4 -= (TYPE4)(Lambda1) + Lambda2 * weights4;
   weights4 += (TYPE4)(lr) * delta4;
   D4ToArray(weights, weights4, start, 1, total, 0);
   D4ToArray(delta_weights, (TYPE4)(0), start, 1, total, 0);
  }
```

Следующим мы изучали метод накопленного импульса. В такой же последовательности будем создавать кернелы реализации методов. Алгоритм кернела *MomentumUpdate* очень напоминает рассмотренный выше кернел стохастического градиентного спуска. Основными отличиями являются введение дополнительного массива для хранения накопленного импульса обновления весовых коэффициентов и параметра сглаживания импульса.

$$\Delta_t = \alpha \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^j} grad_i^j + \beta \Delta_{t-1}$$

$$w_{il}^j = w_{il}^j - \Delta_t$$

В теле кернела мы, как и в предыдущем методе, считываем соответствующие значения массивов в векторные переменные. При этом усредняем накопленные градиент. Затем корректируем весовые коэффициенты на параметры регуляризации. После этого сначала обновим импульс изменения весового коэффициента с учетом среднего градиента и ранее накопленного импульса и только после этого скорректируем весовые коэффициенты на обновленный импульс. Перед выходом из кернела перенесем значения векторных переменных в соответствующие элементы матриц весов и моментов. Накопительный массив дельт обнулим.

```
__kernel void MomentumUpdate(__global TYPE* delta_weights,
                             __global TYPE* weights,
                             __global TYPE* momentum,
                             int total, int batch_size,
                             TYPE learningRate,
                             TYPE beta,
                             TYPE Lambda1, TYPE Lambda2)
  {
   int start = 4 * get_global_id(0);
//---
   TYPE4 delta4 = ToVect4(delta_weights, start, 1, total, 0) /
                                                    ((TYPE4)batch_size);
   TYPE4 weights4 = ToVect4(weights, start, 1, total, 0);
   TYPE4 momentum4 = ToVect4(momentum, start, 1, total, 0);
   weights4 -= (TYPE4)(Lambda1) + Lambda2 * weights4;
   momentum4 = (TYPE4)(learningRate) * delta4 + (TYPE4)(beta) * momentum4;
   weights4 += momentum4;
   D4ToArray(weights, weights4, start, 1, total, 0);
   D4ToArray(momentum, momentum4, start, 1, total, 0);
   D4ToArray(delta_weights, (TYPE4)(0), start, 1, total, 0);
  }
```

Метод оптимизации *AdaGrad* так же, как и метод накопленного импульса, использует один массив для накопления моментов. Но в отличие от предыдущего метода, суммировать будем квадраты градиентов и отсутствует коэффициент сглаживания.

$$G^j_{ilt} = G^j_{ilt-1} + (grad^j_{ilt})^2$$

$$w^j_{il} = w^j_{il} - \frac{\alpha}{\sqrt{G^j_{ilt} + \varepsilon}} * \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w^j_{il}} grad^j_{il},$$

Также изменился подход к использованию накопленного момента. Если в предыдущем методе мы использовали накопленный импульс для снижения хаотичности обновлений и поддержания плавности движений в направлении антиградиента, то в методе адаптивного градиента накопленный квадрат градиентов используется для снижения скорости обучения с каждой итерацией. Это и отражено в коде кернела, представленном ниже.

```
__kernel void AdaGradUpdate(__global TYPE* delta_weights,
                            __global TYPE* weights,
                            __global TYPE* momentum,
                            int total, int batch_size,
                            TYPE learningRate,
                            TYPE Lambda1, TYPE Lambda2)
  {
   int start = 4 * get_global_id(0);
//---
   TYPE4 delta4 = ToVect4(delta_weights, start, 1, total, 0) /
                                                    ((TYPE4)batch_size);
   TYPE4 weights4 = ToVect4(weights, start, 1, total, 0);
   TYPE4 momentum4 = ToVect4(momentum, start, 1, total, 0);
//---
   weights4 -= (TYPE4)(Lambda1) + Lambda2 * weights4;
   momentum4 = momentum4 + pow(delta4, 2);
   weights4 += learningRate / sqrt(momentum4 + 1.0e-37f);
   D4ToArray(weights, weights4, start, 1, total, 0);
   D4ToArray(momentum, momentum4, start, 1, total, 0);
   D4ToArray(delta_weights, (TYPE4)(0), start, 1, total, 0);
  }
```

Основная проблема метода адаптивного градиента заключается в постоянном накоплении квадрата градиента. При длительном обучении это может привести к снижению обучающего коэффициента до нуля и практической остановки обучения нейронной сети. Эта проблема решается в методе *RMSProp* введением коэффициента сглаживания для накопления квадратов градиента. Это позволяет нам ограничить рост накопленной суммы квадратов градиентов и тем самым ограничить снижение скорости обучения.

$$REMS(G_{il}^{j})_{t} = \gamma(grad_{il(t-k)}^{j})^{2} + (1-\gamma)REMS(G_{il}^{j})_{t-1}$$

$$w_{il}^{j} = w_{il}^{j} - \frac{\alpha}{\sqrt{REMS(G_{il}^{j}) + \varepsilon}} * \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^{j}} grad_{il}^{j},$$

В остальном алгоритм кернела повторяет ранее рассмотренные методы обновления весовых коэффициентов.

```
__kernel void RMSPropUpdate(__global TYPE* delta_weights,
                            __global TYPE* weights,
                            __global TYPE* momentum,
                            int total, int batch_size,
                            TYPE learningRate,
                            TYPE beta,
                            TYPE Lambda1, TYPE Lambda2)
  {
   int start = 4 * get_global_id(0);
//---
   TYPE4 delta4 = ToVect4(delta_weights, start, 1, total, 0) /
                                                    ((TYPE4)batch_size);
   TYPE4 weights4 = ToVect4(weights, start, 1, total, 0);
   TYPE4 momentum4 = ToVect4(momentum, start, 1, total, 0);
//---
   weights4 -= (TYPE4)(Lambda1) + Lambda2 * weights4;
   momentum4 = beta * momentum4 + (1 - beta) * pow(delta4, 2);
   weights4 += delta4 * learningRate / (sqrt(momentum4) + 1.0e-37f);
   D4ToArray(weights, weights4, start, 1, total, 0);
   D4ToArray(momentum, momentum4, start, 1, total, 0);
   D4ToArray(delta_weights, (TYPE4)(0), start, 1, total, 0);
  }
```

В методе обновления оптимизации *AdaDelta* авторы попытались отказаться от обучающего коэффициента. Но платой за это стало введение дополнительного буфера моментов и второго коэффициента сглаживания.

$$REMS(\delta w_{il}^{j})_{t} = \gamma_{w}(\delta w_{il(t-k)}^{j})^{2} + (1-\gamma_{w})REMS(\delta w_{il}^{j})_{t-1}$$

$$REMS(G_{il}^{j})_{t} = \gamma_{G}(grad_{il(t-k)}^{j})^{2} + (1-\gamma_{G})REMS(G_{il}^{j})_{t-1}$$

$$w_{il}^{j} = w_{il}^{j} - \frac{REMS(\delta w_{il}^{j})}{REMS(G_{il}^{j}) + \varepsilon} * \frac{\delta A_{ij}(S_{ij}(W_{ij}, Y'_{j-1}))}{\delta w_{il}^{j}} grad_{il}^{j},$$

В методе используются две экспоненциальные средние. Первая усредняет квадрат значений соответствующего весового коэффициента, а вторая, как и в двух предыдущих методах, — квадрат градиентов на этом весовом коэффициенте. Вместо коэффициента обучения используется отношение квадратных корней из двух указанных средних. В результате получаем метод, у которого с ростом абсолютного значения весового коэффициента растет и коэффициент обучения. В то же время рост абсолютного значения градиента ошибки ведет к снижению скорости обучения.

Рассмотрим реализацию данного метода в кернеле *AdaDeltaUpdate*. В параметрах кернелу передаются указатели на четыре массива данных:

- *delta_weights* — массив накопленных градиентов ошибки;
- *weights* — матрица весовых коэффициентов;
- *momentumW* — матрица экспоненциальных средних квадратов весовых коэффициентов;
- *momentumG* — матрица экспоненциальных средних квадратов градиентов ошибки.

Помимо указателей на массивы в параметрах кернела указываются размер массивов, размер пакета, два коэффициента экспоненциального сглаживания и параметры регуляризации.

В теле кернела мы определяем сдвиг до элементов, подлежащих обработке в данном потоке, и считываем нужные элементы из массивов в векторные переменные для последующей обработки. Следующим шагом скорректируем весовые коэффициенты на параметры регуляризации, обновим моменты весов и градиентов. После этого обновим сами весовые коэффициенты. В заключение запишем новые значения в массивы данных и обнулим массив накопленных градиентов.

```
__kernel void AdaDeltaUpdate(__global TYPE* delta_weights,
                             __global TYPE* weights,
                             __global TYPE* momentumW,
                             __global TYPE* momentumG,
                             int total, int batch_size,
                             TYPE beta1, TYPE beta2,
                             TYPE Lambda1, TYPE Lambda2)
  {
   int start = 4 * get_global_id(0);
//---
   TYPE4 delta4 = ToVect4(delta_weights, start, 1, total, 0) /
                                                    ((TYPE4)batch_size);
   TYPE4 weights4 = ToVect4(weights, start, 1, total, 0);
   TYPE4 momentumW4 = ToVect4(momentumW, start, 1, total, 0);
   TYPE4 momentumG4 = ToVect4(momentumG, start, 1, total, 0);
//---
   weights4 -= (TYPE4)(Lambda1) + Lambda2 * weights4;
   momentumW4 = beta1 * momentumW4 + (1 - beta1) * pow(weights4, 2);
   momentumG4 = beta2 * momentumG4 + (1 - beta2) * pow(delta4, 2);
   weights4 += delta4 * sqrt(momentumW4) / (sqrt(momentumG4) + 1.0e-37f);
   D4ToArray(weights, weights4, start, 1, total, 0);
   D4ToArray(momentumW, momentumW4, start, 1, total, 0);
   D4ToArray(momentumG, momentumG4, start, 1, total, 0);
   D4ToArray(delta_weights, (TYPE4)(0), start, 1, total, 0);
  }
```

Последним мы изучали метод адаптивной оценки моментов *Adam*. Напомню математические формулы этого метода.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) grad_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) grad_t^2$$

$$\hat{m} = \frac{m}{1 - \beta_1}$$

$$\hat{v} = \frac{v}{1 - \beta_2}$$

$$w_t = w_{t-1} - \alpha \frac{\hat{m}}{\sqrt{\hat{v}} + \varepsilon}$$

По сравнению с рассмотренными ранее методами они выглядят наиболее сложными. Но в них нет ничего страшного, и метод поддается реализации. Он так же, как и AdaDelta, использует два буфера для накопления моментов. В первом буфере накапливаем импульс градиентов, а во втором — импульс квадратов градиента. Оба буфера используют экспоненциальное сглаживание, но каждый использует свой коэффициент сглаживания. Кроме того, в метод возвращается коэффициент обучения.

Рассмотрим реализацию метода в кернеле *AdamUpdate*. В параметрах кернела, как всегда, мы будем передавать указатели на массивы данных:

- delta_weights — накопленные дельты градиентов;
- weights — матрица весовых коэффициентов;
- momentumM — матрица накопленных градиентов;
- momentumV — матрица накопленных квадратов градиентов.

Также в параметрах будем передавать размер массивов, размер пакета, коэффициент обучения, коэффициенты сглаживания и параметры регуляризации.

В начале кернела, как и в реализации предыдущих методов оптимизации, определим сдвиг до обрабатываемых элементов в массивах. Для исключения излишней путаницы мы будем синхронно использовать элементы массивов, то есть размер массивов и сдвиг до соответствующих элементов будет идентичен.

Скопируем обрабатываемые элементы массивов в векторные переменные. Как всегда, накопленные дельты сразу разделим на размер пакета и сохраним среднее арифметическое значения градиента ошибки.

Далее вычислим обновленные значения накопленных импульсов и скорректируем их, как было предложено авторами метода.

После проведения подготовительной работы скорректируем наши весовые коэффициенты. Как и ранее, сначала скорректируем с учетом параметров регуляризации, а затем — в сторону антиградиента по правилам метода оптимизации, то есть из текущего весового коэффициента вычтем произведение обучающего коэффициента и отношения первого момента градиента к квадратному корню из его второго момента.

И в заключение кернела сохраним полученные значения в соответствующие массивы. Не забываем обнулить массив накопленных дельт.

```
__kernel void AdamUpdate(__global TYPE* delta_weights,
                         __global TYPE* weights,
                         __global TYPE* momentumM,
                         __global TYPE* momentumV,
                         int total, int batch_size,
                         TYPE learningRate,
                         TYPE beta1, TYPE beta2,
                         TYPE Lambda1, TYPE Lambda2)
  {
   int start = 4 * get_global_id(0);
//---
   TYPE4 delta4 = ToVect4(delta_weights, start, 1, total, 0) /
                                                    ((TYPE4)batch_size);
   TYPE4 weights4 = ToVect4(weights, start, 1, total, 0);
   TYPE4 momentumM4 = ToVect4(momentumM, start, 1, total, 0);
   TYPE4 momentumV4 = ToVect4(momentumV, start, 1, total, 0);
//---
   momentumM4 = beta1 * momentumM4 + (1 - beta1) * delta4;
   momentumV4 = beta2 * momentumV4 + (1 - beta2) * pow(delta4, 2);
   TYPE4 m = momentumM4 / (1 - beta1);
   TYPE4 v = momentumV4 / (1 - beta2);
   weights4 -= (TYPE4)(Lambda1) + Lambda2 * weights4;
   weights4 += learningRate * m / (sqrt(v) + 1.0e-37f);
   D4ToArray(weights, weights4, start, 1, total, 0);
   D4ToArray(momentumM, momentumM4, start, 1, total, 0);
   D4ToArray(momentumV, momentumV4, start, 1, total, 0);
   D4ToArray(delta_weights, (TYPE4)(0), start, 1, total, 0);
  }
```

На данном этапе мы завершили работу по написанию программы *OpenCL*. Конечно, мы еще вернемся к этой работе при реализации других архитектурных решений нейронных слоев. Но в части полносвязного нейронного слоя эту работу можно считать завершенной. Мы сохраняем написанный нами код и переходим к реализации процессов обмена данными между основной программой и кернелами OpenCL, а также функций вызова кернелов. Эту работу нам предстоит выполнить на стороне основной программы.

3.7.2 Реализация функционала на стороне основной программы

Реализация функционала на стороне основной программы также потребует от нас некоторых знаний в организации процесса и трудозатрат. Начнем с подготовительной работы. Прежде всего в нашем *файле дефайнов* добавим загрузку написанной выше *OpenCL*-программы в качестве ресурса и присвоим его содержимое переменной типа string. Тут же добавим к программе заготовленные макроподстановки типа данных и размера локального массива.

```
#resource "opencl_program.cl" as string OCLprogram
//---
#define TYPE                            float
#define LOCAL_SIZE                      256
const string ExtType = StringFormat("#define TYPE %s\r\n"
                                    "#define TYPE4 %s4\r\n"
                                    "#define LOCAL_SIZE %d\r\n",
                                     typename(TYPE),typename(TYPE),LOCAL_SIZE);
#define cl_program                      ExtType+OCLprogram
```

При объявлении кернелов в основной программе функция *CLKernelCreate* возвращает хендл для его вызова. Для работы с технологией *OpenCL* мы будем использовать класс *CMyOpenCL*, который является наследником от стандартного класса *COpenCL*. В указанных классах реализованы массивы для хранения хендлов. Доступ к конкретному кернелу осуществляется по индексу в массиве. Чтобы облегчить работу с этими индексами и сделать код программы более читаемым, добавим в константы индексы для всех созданных выше кернелов. Для явного выделения в коде программы индекса кернела будем начинать все именованные константы кернелов с *def_k*.

```
//+------------------------------------------------------------------+
//| OpenCL Kernels                                                   |
//+------------------------------------------------------------------+
#define def_k_PerceptronFeedForward    0
#define def_k_LineActivation           1
#define def_k_SigmoidActivation        2
#define def_k_SigmoidDerivative        3
#define def_k_TANHActivation           4
#define def_k_TANHDerivative           5
#define def_k_LReLuActivation          6
#define def_k_LReLuDerivative          7
#define def_k_SoftMAXActivation        8
#define def_k_SoftMAXDerivative        9
#define def_k_SwishActivation          10
#define def_k_SwishDerivative          11
#define def_k_CalcOutputGradient       12
#define def_k_CalcHiddenGradient       13
#define def_k_CalcDeltaWeights         14
#define def_k_SGDUpdate                15
#define def_k_MomentumUpdate           16
#define def_k_AdaGradUpdate            17
#define def_k_RMSPropUpdate            18
#define def_k_AdaDeltaUpdate           19
#define def_k_AdamUpdate               20
```

Для указания параметров при вызове кернелов также используются индексы. Только теперь они не задаются явно — вместо этого используется порядковый номер в списке параметров кернела программы *OpenCL*. Все кернелы используют свой набор параметров, поэтому определим именованные константы для всех созданных кернелов. Чтобы не путаться между идентичными параметрами различных кернелов, в наименование константы включим указатель на соответствующий кернел. К примеру, константы параметров для кернела прямого прохода базового полносвязного слоя будут начинаться с *def_pff*.

```
//--- прямой проход перцептрона
#define def_pff_inputs                0
#define def_pff_weights               1
#define def_pff_outputs               2
#define def_pff_inputs_total          3
```

Аналогичным образом объявим константы для всех написанных кернелов.

```
//--- определение градиента ошибки слоя результатов
#define def_outgr_target               0
#define def_outgr_outputs              1
#define def_outgr_gradients            2
#define def_outgr_loss_function        3

//--- определение градиента ошибки скрытого слоя
#define def_hidgr_gradient_inputs      0
#define def_hidgr_weights              1
#define def_hidgr_gradients            2
#define def_hidgr_outputs_total        3

//--- определение градиента ошибки на уровне матрицы весов
#define def_delt_inputs                0
#define def_delt_delta_weights         1
#define def_delt_gradients             2

//--- оптимизация параметров стохастическим градиентным спуском
#define def_sgd_delta_weights          0
#define def_sgd_weights                1
#define def_sgd_total                  2
#define def_sgd_batch_size             3
#define def_sgd_learningRate           4
#define def_sgd_Lambda1                5
#define def_sgd_Lambda2                6

//--- оптимизация параметров методом моментов
#define def_moment_delta_weights       0
#define def_moment_weights             1
#define def_moment_momentum            2
#define def_moment_total               3
#define def_moment_batch_size          4
#define def_moment_learningRate        5
#define def_moment_beta                6
#define def_moment_Lambda1             7
#define def_moment_Lambda2             8

//--- оптимизация параметров методом AdaGrad
#define def_adagrad_delta_weights      0
#define def_adagrad_weights            1
#define def_adagrad_momentum           2
#define def_adagrad_total              3
#define def_adagrad_batch_size         4
#define def_adagrad_learningRate       5
#define def_adagrad_Lambda1            6
#define def_adagrad_Lambda2            7
```

```
//--- оптимизация параметров методом RMSProp
#define def_rms_delta_weights          0
#define def_rms_weights                1
#define def_rms_momentum               2
#define def_rms_total                  3
#define def_rms_batch_size             4
#define def_rms_learningRate           5
#define def_rms_beta                   6
#define def_rms_Lambda1                7
#define def_rms_Lambda2                8

//--- оптимизация параметров методом AdaDelta
#define def_adadelt_delta_weights      0
#define def_adadelt_weights            1
#define def_adadelt_momentumW          2
#define def_adadelt_momentumG          3
#define def_adadelt_total              4
#define def_adadelt_batch_size         5
#define def_adadelt_beta1              6
#define def_adadelt_beta2              7
#define def_adadelt_Lambda1            8
#define def_adadelt_Lambda2            9

//--- оптимизация параметров методом Adam
#define def_adam_delta_weights         0
#define def_adam_weights               1
#define def_adam_momentumM             2
#define def_adam_momentumV             3
#define def_adam_total                 4
#define def_adam_batch_size            5
#define def_adam_learningRate          6
#define def_adam_beta1                 7
#define def_adam_beta2                 8
#define def_adam_Lambda1               9
#define def_adam_Lambda2               10

//--- функции активации
#define def_activ_inputs               0
#define def_activ_outputs              1
#define def_activ_param_a              2
#define def_activ_param_b              3

//--- корректировка градиента на производную функции активации
#define def_deactgr_outputs            0
#define def_deactgr_gradients          1
#define def_deactgr_deact_gradient     2
#define def_deactgr_act_param_a        3
#define def_deactgr_act_param_b        4
```

Я специально выше представил полный набор, чтобы предоставить вам справочник констант. Он поможет в чтении и понимании кода наших последующих действий по внедрению технологии *OpenCL* в проект.

После описания констант перейдем к созданию классов, на которые будем возложена работа по обслуживанию инструментов *OpenCL*. Мы уже не раз о них упоминали. Настало времени больше узнать об их функционале.

Прежде всего, это класс *CMyOpenCL*. Он является наследником от класса *COpenCL* из стандартных библиотек *MQL5*. Стандартная библиотека написана хорошо и имеет достаточный функционал для организации работы. Но лично мне показался неудобным один момент: в процессе работы с буферами для обмена данными между основной программой и контекстом *OpenCL* используется подход аналогичный с другими объектами процесса — при создании буфера мы должны указать его индекс в общем массиве буферов. Это вполне рабочий вариант, когда у нас заранее известны все буферы и их количество. Но наш случай немного сложнее.

```
class CMyOpenCL    :  public COpenCL
  {
public:
                      CMyOpenCL(void)    {};
                     ~CMyOpenCL(void)    {};
   //--- initialization and shutdown
   virtual bool       Initialize(const string program, const bool show_log = true);
   //---
   template<typename T>
   int                AddBufferFromArray(T &data[], const uint data_array_offset,
                                         const uint data_array_count, const uint flags);
   int                AddBufferFromArray(MATRIX &data,
                                         const uint data_array_offset, const uint flags);
   int                AddBuffer(const uint size_in_bytes, const uint flags);
   bool               CheckBuffer(const int index);
   //---
   bool               BufferFromMatrix(const int buffer_index, MATRIX &data,
                                       const uint data_array_offset, const uint flags);
   bool               BufferRead(const int buffer_index, MATRIX &data,
                                                 const uint cl_buffer_offset);
   bool               BufferWrite(const int buffer_index, MATRIX &data,
                                                 const uint cl_buffer_offset);
  };
```

Ранее мы уже обсуждали, что в зависимости от выбранного метода обновления весовых коэффициентов у нас меняется количество используемых буферов для накопления моментов. Вдобавок к вышесказанному, мы не можем знать заранее, какое количество нейронных слоев будет использовать пользователь для решения своих задач. Поэтому мне потребовался динамический массив для хранения хендлов буферов данных. Эта задача была решена путем добавления небольшого метода *AddBufferFromArray*. Параметры данного метода аналогичны параметрам метода *BufferFromArray* родительского класса за исключением индекса буфера. В теле метода организован цикл по поиску пустых ячеек в массиве хранения хендлов буферов. Первая незанятая ячейка используется для создания буфера. В том случае когда в массиве нет свободных элементов, метод расширяет массив. Непосредственно создание буфера осуществляется вызовом указанного выше метода родительского класса.

В результат выполнения операций метод возвращает индекс созданного буфера. В случае возникновения ошибок в процессе выполнения операций метод вернет константу *INVALID_HANDLE*.

Хочется обратить внимание еще на тот момент, что метод создан по паттерну шаблона функции. Это позволяет использовать один метод для создания буферов разных типов данных.

```
template<typename T>
int CMyOpenCL::AddBufferFromArray(T &data[], const uint data_array_offset,
                                  const uint data_array_count, const uint flags
                                 )
  {
   int result=INVALID_HANDLE;
   for(int i=0; i<m_buffers_total; i++)
     {
      if(m_buffers[i]!=INVALID_HANDLE)
         continue;
      result=i;
      break;
     }
//---
   if(result<0)
     {
      if(ArrayResize(m_buffers,m_buffers_total+1)>0)
        {
         m_buffers_total=ArraySize(m_buffers);
         result=m_buffers_total-1;
         m_buffers[result]=INVALID_HANDLE;
        }
      else
         return result;
     }
//---
   if(!BufferFromArray(result,data,data_array_offset,data_array_count,flags))
      return INVALID_HANDLE;
//---
   return result;
  }
```

Созданный выше метод позволяет создавать буферы из массивов любых типов данных, однако он не применим при использовании матриц. Поэтому была создана перегрузка метода. Алгоритм метода остался без изменений.

```
int CMyOpenCL::AddBufferFromArray(MATRIX &data,
                                  const uint data_array_offset,
                                  const uint flags
                                 )
  {
//--- Поиск свободного элемента в динамическом массиве указателей
   int result = -1;
   for(int i = 0; i < m_buffers_total; i++)
     {
      if(m_buffers[i] != INVALID_HANDLE)
         continue;
      result = i;
      break;
     }
//--- Если свободный элемент не найден, добавляем новый элемент в массив
   if(result < 0)
     {
      if(ArrayResize(m_buffers, m_buffers_total + 1) > 0)
        {
         m_buffers_total = ArraySize(m_buffers);
         result = m_buffers_total - 1;

         m_buffers[result] = INVALID_HANDLE;
        }
      else
         return result;
     }
//--- Создаем буфер в контексте OpenCL
   if(!BufferFromMatrix(result, data, data_array_offset, flags))
      return -1;
   return result;
  }
```

Забегая немного вперед хочу сказать, что мы не всегда будем создавать буферы по готовым массивам. Иногда нам потребуется просто создать буфер в контексте *OpenCL* без создания его дубликата в основной памяти. Или, к примеру, определенный буфер используется только для получения результатов, и нам нет необходимости загружать его данные в контекст до выполнения операций. А мы уже говорили, что процесс копирования данных является дорогой операцией. И мы бы хотели минимизировать такие операции. Поэтому нам было бы проще просто создать буфер данных в контексте определенного размера без копирования данных. Для таких случаев мы создадим метод *AddBuffer*. Как можно заметить, алгоритм метода практически полностью повторяет методы предыдущего метода. Только в параметрах метод получает не массив, а размер буфера в байтах. В конце метода мы вызываем метод *BufferCreate*, который создаст буфер заданного размера в контексте *OpenCL*.

```
int CMyOpenCL::AddBuffer(const uint size_in_bytes, const uint flags)
  {
//--- Поиск свободного элемента в динамическом массиве указателей
   int result = -1;
   for(int i = 0; i < m_buffers_total; i++)
     {
      if(m_buffers[i] != INVALID_HANDLE)
         continue;
      result = i;
      break;
     }
//--- Если свободный элемент не найден, добавляем новый элемент в массив
   if(result < 0)
     {
      if(ArrayResize(m_buffers, m_buffers_total + 1) > 0)
        {
         m_buffers_total = ArraySize(m_buffers);
         result = m_buffers_total - 1;
         m_buffers[result] = INVALID_HANDLE;
        }

      else
         return result;
     }
//--- Создаем буфер в контексте OpenCL
   if(!BufferCreate(result, size_in_bytes, flags))
      return -1;
   return result;
  }
```

Также мы создали методы для чтения (*BufferRead*) и записи (*BufferWrite*) данных буфера контекста *OpenCL* в матрицу основной памяти. Алгоритм методов полностью идентичен. Для примера рассмотрим только метод чтения данных. В параметрах метод получает идентификатор буфера в динамическом массиве нашего класса, матрицу для записи данных и смещение в буфере контекста.

Прошу не путать идентификатор буфера в динамическом массиве класса и хендл буфера в контексте *OpenCL*. Работа класса построена таким образом, что внешней программе мы передаем только порядковый номер элемента в динамическом массиве нашего класса, который содержит хендл этого буфера. Поэтому при создании буфера в контексте с использованием класса внешняя программа не имеет прямого доступа к созданному буферу в контексте. Вся работа с буфером должна осуществляться через методы класса.

В теле метода мы сначала проверяем полученный идентификатор буфера на соответствие размеру нашего динамического массива. Затем мы проверяем действительность хендла указанного буфера. Кроме того, проверим действительность хендлов контекста и программы *OpenCL*. Только после успешного прохождения всех контролей вызываем функцию чтения данных из буфера. Не забываем на каждом шаге проверять результат выполнения операций. В завершение метода вернем логический результат выполнения операций.

```
bool CMyOpenCL::BufferRead(const int buffer_index, MATRIX &data,
                                   const uint cl_buffer_offset)
  {
//--- проверка параметров
   if(buffer_index < 0 || buffer_index >= m_buffers_total || data.Rows() <= 0)
      return(false);
   if(m_buffers[buffer_index] == INVALID_HANDLE)
      return(false);
   if(m_context == INVALID_HANDLE || m_program == INVALID_HANDLE)
      return(false);
//--- чтение данных буфера из контекста OpenCL
   if(!CLBufferRead(m_buffers[buffer_index], cl_buffer_offset, data))
      return(false);
//---
   return(true);
  }
```

Второй класс, который мы создадим и будем использовать для передачи данных между основной программой и контекстом *OpenCL,* — класс буфера данных *CBufferType*. Класс создан наследником базового класса *CObject*. Так как родительский класс является базовым, то нам необходимо воссоздать весь необходимый функционал.

Помимо создания новых методов в новом классе появились две новые переменные:

- *m_cOpenCL* — указатель на объект класса *CMyOpenCL*
- *m_myIndex* — индекс текущего буфера в динамическом массиве хранения хендлов буферов в классе *CMyOpenCL.*

Также появилась матрица *m_mMatrix* для хранения данных. Здесь мы немного отступили от общепринятых правил создания классов. Обычно принято ограничивать доступ к внутренним переменным, и все обращение к ним строится через методы класса. Каждый такой метод ограничивает степень свободы к внутренним переменным и требует дополнительные затраты времени на выполнение дополнительных операций метода. Конечно, такой подход позволяет полностью контролировать изменение состояний переменных. Но в построении нейронных моделей мы стараемся минимизировать затраты времени на каждой итерации, так как миллисекунды на одной итерации выражаются в значительные временные затраты из-за повторных обращений. Поэтому матрицу данных *m_mMatrix* мы объявили в публичном пространстве. Конечно, тот факт, что класс будет использоваться для хранения и передачи данных внутри нашего глобального проекта и все буферы будут частными или защищенными объектами других классов, минимизируем наши риски.

```
class CBufferType: public CObject
  {
protected:
   CMyOpenCL*        m_cOpenCL;     // объект контекста OpenCL
   int               m_myIndex;     // индекс буфера данных в контексте
public:
                     CBufferType(void);
                    ~CBufferType(void);
   //--- матрица данных
   MATRIX            m_mMatrix;
   //--- метод инициализации буфера начальными значениями
   virtual bool      BufferInit(const ulong rows, const ulong columns,
                                                      const TYPE value = 0);
   //--- создание нового буфера в контексте OpenCL
   virtual bool      BufferCreate(CMyOpenCL *opencl);
   //--- удаление буфера в контексте OpenCL
   virtual bool      BufferFree(void);
   //--- чтение данных буфера из контекста OpenCL
   virtual bool      BufferRead(void);
   //--- запись данных буфера в контекст OpenCL
   virtual bool      BufferWrite(void);
   //--- получение индекса буфера
   virtual int       GetIndex(void);
   //--- изменение индекса буфера
   virtual bool      SetIndex(int index)
                       {
                        if(!m_cOpenCL.BufferFree(m_myIndex))
                           return false;
                        m_myIndex = index;
                        return true;
                       }
   //--- копирование данных буфера в массив
   virtual int       GetData(TYPE &values[], bool load = true);
   virtual int       GetData(MATRIX &values, bool load = true);
   virtual int       GetData(CBufferType* values, bool load = true);
   //--- расчет среднего значения буфера данных
   virtual TYPE      MathMean(void);
   //--- операции с векторами
   virtual bool      SumArray(CBufferType* src);
   virtual int       Scaling(TYPE value);
   virtual bool      Split(CBufferType* target1, CBufferType* target2,
                                                    const int position);
   virtual bool      Concatenate(CBufferType* target1, CBufferType* target2,
                                 const int positions1, const int positions2);
   //--- методы работы с файлами
   virtual bool      Save(const int file_handle);
   virtual bool      Load(const int file_handle);
   //--- идентификатор класса
   virtual int       Type(void)              const { return defBuffer;             }
   //--- методы работы с матрицей данных
   ulong             Rows(void)              const { return m_mMatrix.Rows();      }
```

```
   ulong             Cols(void)              const { return m_mMatrix.Cols();       }
   uint              Total(void)             const { return (uint)(m_mMatrix.Rows() *
                                                                   m_mMatrix.Cols()); }
   TYPE              At(uint index)          const { return m_mMatrix.Flat(index);  }
   TYPE              operator[](ulong index) const { return m_mMatrix.Flat(index);  }
   VECTOR            Row(ulong row)                { return m_mMatrix.Row(row);     }
   VECTOR            Col(ulong col)                { return m_mMatrix.Col(col);     }
   bool              Row(VECTOR& vec,  ulong row)  { return m_mMatrix.Row(vec, row);}
   bool              Col(VECTOR& vec,  ulong col)  { return m_mMatrix.Col(vec, col);}
   bool              Activation(MATRIX& mat_out, ENUM_ACTIVATION_FUNCTION func)
                                   { return m_mMatrix.Activation(mat_out, func); }
   bool              Derivative(MATRIX& mat_out, ENUM_ACTIVATION_FUNCTION func)
                                   { return m_mMatrix.Derivative(mat_out, func); }
   bool              Reshape(ulong rows, ulong cols)
                                   { return m_mMatrix.Reshape(rows, cols);       }
//---
   bool              Update(uint index, TYPE value)
                     {
                      if(index >= Total())
                         return false;
                      m_mMatrix.Flat(index, value);
                      return true;
                     }

   bool              Update(uint row, uint col, TYPE value)
                     {
                      if(row >= Rows() || col >= Cols())
                         return false;
                      m_mMatrix[row, col] = value;
                      return true;
                     }
  };
```

Структура методов класса довольно разнообразна. Некоторые из них аналогичны матричным функциям и выполняют тот же функционал — созданы для работы с матрицей данных. Другие выполняют функционал взаимодействия с контекстом *OpenCL*. Рассмотрим некоторые из них подробнее.

В конструкторе класса мы лишь зададим начальные значения новых переменных. Они заполняются пустыми значениями.

```
CBufferType::CBufferType(void)  : m_myIndex(-1)
  {
   m_cOpenCL = NULL;
  }
```

В деструкторе класса мы выполним операции очистки памяти. Здесь мы очистим буфер в контексте OpenCL.

```
CBufferType::~CBufferType(void)
  {
   if(m_cOpenCL && m_myIndex >= 0 && m_cOpenCL.BufferFree(m_myIndex))
        {
         m_myIndex = -1;
         m_cOpenCL = NULL;
        }
  }
```

Метод инициализации буфера *BufferInit* мы уже использовали в конструкторе класса нейронного слоя. Основной функционал этого метода — создать матрицу заданного размера и заполнить его начальными значениями. Размер буфера и начальные значения указываются в параметрах метода. В рамках данного проекта мы будем заполнять массивы нулевыми значениями на этапе инициализации нейронной сети, а также обнулять буферы накопленных дельт после обновления матрицы весов.

```
bool CBufferType::BufferInit(ulong rows, ulong columns, TYPE value)
  {
   if(rows <= 0 || columns <= 0)
      return false;
   m_mMatrix = MATRIX::Full(rows, columns, value);
   if(m_cOpenCL)
     {
      CMyOpenCL *opencl=m_cOpenCL;
      BufferFree();
      return BufferCreate(opencl);
     }
//---
   return true;
  }
```

Следующим идет метод создания буфера в контексте *OpenCL*. В параметрах методу передается указатель на экземпляр класса *CMyOpenCL*, в контексте которого необходимо создать буфер.

Начинается метод с блока контролей. Сначала проверим действительность полученного указателя — в случае получения недействительного указателя удаляем буфер, ранее созданный в контексте *OpenCL*, и выходим из метода.

```
bool CBufferType::BufferCreate(CMyOpenCL *opencl)
  {
//--- блок проверки исходных данных
   if(!opencl)
     {
      BufferFree();
      return false;
     }
```

Затем проверяем соответствие ранее сохраненному указателю. Если указатели идентичны и индекс буфера уже сохранен, то мы не будем создавать новый буфер в контексте *OpenCL*, а лишь заново скопируем данные из матрицы в буфер обмена данных. Для этого вызовем метод *BufferWrite*. Данный метод обладает своим блоком проверок, с которым мы познакомимся немного позже, и возвращает логический результат операции. Выходим из метода с результатом метода записи данных в контекст *OpenCL*.

```
//--- если полученный указатель совпадает с ранее сохраненным,
//--- просто копируем содержимое буфера в память контекста
   if(opencl == m_cOpenCL && m_myIndex >= 0)
      return BufferWrite();
```

Дальнейший код метода будет выполняться только в том случае, если мы не вышли из метода при осуществлении предшествующих операций. Здесь мы проверяем действительность ранее сохраненного указателя на экземпляр класса *CMyOpenCL* и наличие индекса в динамическом массиве хранения хендлов буферов данных. При выполнении этого условия до продолжения операций мы должны очистить память и удалить существующий буфер с помощью метода *BufferFree*. Только после успешного удаления старого буфера у нас есть право на открытие нового. В противном случае неконтролируемое потребление ресурсов памяти приведет к ее дефициту и соответствующим последствиям.

```
//--- проверяем наличие ранее сохраненного указателя на контекст OpenCL
//--- при наличии удаляем буфер из неиспользуемого контекста
   if(m_cOpenCL && m_myIndex >= 0)
     {
      if(m_cOpenCL.BufferFree(m_myIndex))
        {
         m_myIndex = -1;
         m_cOpenCL = NULL;
        }
      else
         return false;
     }
```

В заключение метода мы инициируем создание нового буфера данных в указанном контексте. Для этого вызовем выше рассмотренный метод AddBufferFromArray. Полученный в ответ на вызов индекс сохраним в переменной m_myIndex. Если операция открытия буфера была успешной, то перед выходом сохраним полученный на вход метода указатель экземпляра класса CMyOpenCL.

```
//--- создаем новый буфер в указанном контексте OpenCL
   if((m_myIndex = opencl.AddBufferFromArray(m_mMatrix, 0, CL_MEM_READ_WRITE)) < 0)
      return false;
   m_cOpenCL = opencl;
//---
   return true;
  }
```

В рассмотренном методе мы использовали два новых метода: один для очистки буфера, второй для записи данных. За очистку буфера отвечает метод *BufferFree*. Алгоритм метода довольно прост. Вначале мы проверяем наличие сохраненных указателя на экземпляр класса *CMyOpenCL* и индекса в динамическом массиве буферов. При их наличии вызываем метод очистки буфера класса *CMyOpenCL* с указанием индекса буфера для удаления. При успешном удалении буфера из контекста очищаем указатель на экземпляр класса *CMyOpenCL* и переменную индекса буфера.

Надо сказать, что вызов данного метода очищает память и удаляет буфер только в контексте *OpenCL*. При этом сама матрица данных и ее содержимое остаются в оперативной памяти. Позже мы сможем эксплуатировать это свойство для более продуктивного использования памяти контекста *OpenCL*.

```
bool CBufferType::BufferFree(void)
  {
//--- проверяем наличие ранее сохраненного указателя на контекст OpenCL
//--- при наличии, удаляем буфер из неиспользуемого контекста
   if(m_cOpenCL && m_myIndex >= 0)
      if(m_cOpenCL.BufferFree(m_myIndex))
        {
         m_myIndex = -1;
         m_cOpenCL = NULL;
         return true;
        }
   if(m_myIndex >= 0)
      m_myIndex = -1;
//---
   return false;
  }
```

Далее предлагаю рассмотреть методы для передачи информации между основной программой и контекстом *OpenCL*. Эта работа выполняется в двух похожих методах — *BufferRead* и *BufferWrite*. Несмотря на разнонаправленность операций алгоритм методов идентичен. В начале методов организован контрольный блок, в котором проверяется действительность указателя на экземпляр класса *CMyOpenCL* и наличие индекса в динамическом массиве буферов. И только после успешного прохождения контрольного блока вызывается одноименный метод класса работы с контекстом OpenCL с указанием индекса буфера, матрицы и смещения в буфере *OpenCL*.

```
bool CBufferType::BufferRead(void)
  {
   if(!m_cOpenCL || m_myIndex < 0)
      return false;
//---
   return m_cOpenCL.BufferRead(m_myIndex, m_mMatrix, 0);
  }

bool CBufferType::BufferWrite(void)
  {
   if(!m_cOpenCL || m_myIndex < 0)
      return false;
//---
   return m_cOpenCL.BufferWrite(m_myIndex, m_mMatrix, 0);
  }
```

Отдельно созданы методы, позволяющие получить и напрямую указать индекс буфера в динамическом массиве хранения хендлов буферов *GetIndex* и *SetIndex*. Их код настолько прост, что я даже не стал их выносить за пределы блока объявления класса.

Мы добавили в класс три одноименных метода *GetData*. Все они выполняют один функционал — копирование данных матрицы в заданную структуру. Различие — в приемнике данных. Это может быть динамический массив, матрица или другой экземпляр класса *CBufferType*.

В первом случае в параметрах метода передается ссылка на массив и флаг, указывающий на необходимость считывания данных из контекста *OpenCL* перед копированием данных. Введение флага вынужденная мера. Как вы могли заметить при рассмотрении метода чтения данных из контекста, при отсутствии указателя на объект CMyOpenCL или индекса в динамическом массиве

буферов метод вернет значение *false*. Это заблокирует получения данных из массива без созданного в контексте OpenCL буфера. Введение флага позволяет управлять этим процессов.

В начале метода мы проверяем флаг и при необходимости считываем данные из контекста. Только потом изменяем размер массива-приемника и создаем цикл копирования данных. В завершение метод возвращает количество скопированных элементов.

```
int CBufferType::GetData(TYPE &values[], bool load = true)
  {
   if(load && !BufferRead())
      return -1;
   if(ArraySize(values) != Total() &&
      ArrayResize(values, Total()) <= 0)
      return false;
//---
   for(uint i = 0; i < Total(); i++)
      values[i] = m_mMatrix.Flat(i);
   return (int)Total();
  }
```

Два других метода построены по аналогичному алгоритму, но с учетом специфики объекта-приемника.

```
int CBufferType::GetData(MATRIX &values, bool load = true)
  {
   if(load && !BufferRead())
      return -1;
//---
   values = m_mMatrix;
   return (int)Total();
  }

int CBufferType::GetData(CBufferType *values, bool load = true)
  {
   if(!values)
      return -1;
   if(load && !BufferRead())
      return -1;
   values.m_mMatrix.Copy(m_mMatrix);
   return (int)values.Total();
  }
```

Теперь, когда мы подготовили константы и классы работы с контекстом *OpenCL*, мы можем продолжить работу по организации процесса непосредственно в классах нашей нейронной сети.

При создании методов нашего базового класса нейронной сети мы не дописали два метода, *UseOpenCL* и *InitOpenCL*. Как видно из названия методов, они предназначены для инициализации и управления процессом работы с технологией *OpenCL*. Первый служит для переключения режима работы, включает и выключает использование OpenCL. Второй инициализирует работу экземпляра класса *CMyOpenCL*.

Сделаем небольшой шаг назад и заполним эти пробелы. В параметрах методу *UseOpenCL* укажем новое состояние в виде логического значения. Использование логического значения для передачи бинарного состояния включения/выключения функции мне кажется интуитивно

понятным. Вполне логично использование значения *true* для включения функционала и *false* для его выключения.

В теле метода организуем разветвление алгоритма в зависимости от устанавливаемого состояния. При поступлении команды на отключение функционала проверим текущий указатель на экземпляр класса CMyOpenCL, который хранится в переменной *m_cOpenCL*. Если указатель недействительный, то функционал не был инициализирован ранее и нам нечего отключать. В таком случае просто обновим состояние флага использования технологии и выйдем из метода.

Если же функционал ранее уже был активирован, а сейчас поступил сигнал на его отключение, то запустим процесс очистки объекта и его удаление. После этого распространим новый (пустой) указатель по объектам нейронной сети, сохраним флаг и выйдем из метода.

```
void CNet::UseOpenCL(bool value)
  {
   if(!value)
     {
      if(!m_cOpenCL)
        {
         m_bOpenCL = value;
         return;
        }
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      if(!!m_cLayers)
         m_cLayers.SetOpencl(m_cOpenCL);
      m_bOpenCL = value;
      return;
     }
```

Дальнейшие операции будут выполняться только при включении функционала *OpenCL*. При поступлении сигнала на включение использования технологии *OpenCL* мы запускаем процесс создания и инициализации нового экземпляра класса *CMyOpenCL*, который вынесен в отдельный метод *InitOpenCL*.

Перед выходом из метода сохраняем новый флаг использования технологии *OpenCL* и распространяем указатель на новый объект по всем объектам нейронной сети. Для этого передадим новый указатель в объект динамического массива хранения слоев нейронной сети, а оттуда указатель по иерархической цепочке будет передан до каждого объекта нейронной сети.

```
//---
   if(!!m_cOpenCL)
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
     }
   m_bOpenCL = InitOpenCL();
   if(!!m_cLayers)
      m_cLayers.SetOpencl(m_cOpenCL);
   return;
  }
```

Непосредственно процесс создания нового экземпляра класса *CMyOpenCL* и его инициализация вынесены в отдельный метод *InitOpenCL*.

В начале метода мы проверяем наличие ранее сохраненного указателя на объект класса CMyOpenCL. И в этот момент возникает вопрос, что мы хотим сделать далее при наличии ранее запущенного объекта. Мы можем продолжить использовать ранее инициализированный экземпляр класса, а можем создать новый. Использование уже существующего объекта на данном этапе кажется менее трудозатратно. Но в таком случае нам может потребоваться дополнительный метод для перезапуска функционала в случае возникновения ошибки какого-либо рода. Это дополнительные трудозатраты, которые, вероятно, потребуют разработку дополнительной системы контролей по коду всего проекта.

Мы выбрали вариант принудительного перезапуска. Поэтому при наличии действительного указателя на ранее созданный экземпляр класса CMyOpenCL мы запускаем процесс удаления из памяти его содержимого, а следом и самого объекта. И только после очистки памяти запускаем процесс создания и инициализации нового объекта. Процесс создания контекста и программы OpenCL скрыт в методе *COpenCL::Initialize*. В параметрах этому методу мы передадим текстовую переменную, содержащую нашу программу. Помните, мы записали в нее код нашей программы из файлового ресурса?

```
bool CNet::InitOpenCL(void)
  {
//--- Удаляем созданные ранее объекты OpenCL
   if(!!m_cOpenCL)
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
     }
//--- Создаем новый объект для работы с OpenCL
   m_cOpenCL = new CMyOpenCL();
   if(!m_cOpenCL)
      return false;
//--- Инициализируем объект работы с OpenCL
   if(!m_cOpenCL.Initialize(cl_program, true))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
```

Далее укажем количество используемых кернелов и буферов. Выше мы объявили константы для 20 кернелов, каждый кернел использует не более 4 буферов данных. Я намеренно не указываю большое количество буферов на данном этапе, так как благодаря нашему новому методу при необходимости массив будет автоматически расширен при создании нового буфера данных. Количество же кернелов в программе статично и не зависит от архитектуры нейронной сети.

```
   if(!m_cOpenCL.SetKernelsCount(20))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
   if(!m_cOpenCL.SetBuffersCount(4))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
```

После этого мы инициализируем все кернелы программы и сохраним хендлы для их вызова в массив внутри объекта класса *CMyOpenCL*.

Мы не создаем сейчас все используемые буферы данных по одной простой причине — их количество зависит от архитектуры нейронной сети и может оказаться больше объема доступной в контексте OpenCL памяти. В случае ее недостатка можно использовать динамическое распределение памяти. Это подразумевает загрузку буферов по мере их необходимости и последующую очистку памяти, когда не планируется использование конкретного буфера данных. Но такой подход ведет к увеличению накладных расходов по копированию данных между основной памятью и контекстом *OpenCL*. Поэтому его использование оправдано только при недостатке памяти *GPU*.

Алгоритм создания кернелов идентичен. Я приведу лишь несколько для примера.

```
   if(!m_cOpenCL.KernelCreate(def_k_PerceptronFeedForward, "PerceptronFeedForward"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }

   if(!m_cOpenCL.KernelCreate(def_k_CalcOutputGradient, "CalcOutputGradient"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }

   if(!m_cOpenCL.KernelCreate(def_k_CalcHiddenGradient, "CalcHiddenGradient"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }

   if(!m_cOpenCL.KernelCreate(def_k_CalcDeltaWeights, "CalcDeltaWeights"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
```

Вот мы и подошли к этапу организации работы с контекстом OpenCL непосредственно в классе нейронного слоя. Напомню, что при создании многих методов класса мы делали разветвление алгоритма метода в зависимости от устройства выполнения операций. Тогда мы создали код организации процесса средствами MQL5 и оставили пробелы в организации процесса на стороне *OpenCL*. Давайте вернемся и заполним эти пробелы.

Начнем с метода прямого прохода. Ранее мы уже разбирали организацию операций средствами MQL5. Сейчас посмотрим на реализацию работы с контекстом *OpenCL*.

```
bool CNeuronBase::FeedForward(CNeuronBase * prevLayer)
  {
//--- блок контролей
   if(!prevLayer || !m_cOutputs || !m_cWeights ||
      !prevLayer.GetOutputs() || !m_cActivation)
      return false;
   CBufferType *input_data = prevLayer.GetOutputs();
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      if(m_cWeights.Cols() != (input_data.Total() + 1))
         return false;
      //---
      MATRIX m = input_data.m_mMatrix;
      if(!m.Reshape(1, input_data.Total() + 1))
         return false;
      m[0, m.Cols() - 1] = 1;
      m_cOutputs.m_mMatrix = m.MatMul(m_cWeights.m_mMatrix.Transpose());
     }
```

Вначале мы проверим наличие индекса буфера у массива исходных данных, матрицы весов и буфера результатов. Здесь логика проста. Если в параметрах метода мы получили указатель на массив данных с уже существующим буфером, то считаем что данные уже загружены в контекст *OpenCL*. Выше, при создании буфера данных в классе *CBufferType*, мы сразу создавали и буфер в контексте *OpenCL*. Следовательно, отсутствие индекса буфера может свидетельствовать об ошибке. Поэтому в подобном случаем мы завершаем метод с результатом *false*. Если же вы используете динамическое распределение памяти, то в этом месте вам нужно будет создать копии всех используемых на в данном кернеле буферов данных и скопировать их содержимое буферов исходных данных в контекст *OpenCL*.

```
   else // Блок OpenCL
     {
      //--- проверка буферов данных
      if(input_data.GetIndex() < 0)
         return false;
      if(m_cWeights.GetIndex() < 0)
         return false;
      if(m_cOutputs.GetIndex() < 0)
         return false;
```

Затем укажем параметры для кернела прямого прохода. Здесь для буферов мы укажем их индексы, а для дискретных параметров — конкретные значения.

```
      //--- передача аргументов кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_PerceptronFeedForward, def_pff_inputs,
                                                          input_data.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_PerceptronFeedForward, def_pff_weights,
                                                          m_cWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_PerceptronFeedForward, def_pff_outputs,
                                                          m_cOutputs.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_PerceptronFeedForward, def_pff_inputs_total,
                                                            input_data.Total()))
         return false;
```

В массиве NDRange укажем количество требуемых параллельных потоков по числу нейронов в текущем слое и запустим кернел на выполнение. Следует обратить внимание, что метод *Execute* не запускает выполнение кернела в буквальном смысле, а лишь осуществляет постановку в очередь для выполнения. Непосредственно запуск кернела осуществляется при попытке считать данные результатов его работы. Однако мы не будем загружать результат выполнения операций каждого кернела. Вместо этого мы выстроим очередь выполнения прямого прохода всей подели и загрузим результат только работы модели с последнего слоя. Это потянет всю очередь выполнения операций. Таким образом, мы снизим объем передаваемых данных и затраты времени на их загрузку.

В случае же использования динамического распределения памяти, после постановки кернела в очередь необходимо будет загрузить все изменения из контекста OpenCL в матрицы данных и удалить неиспользуемые буферы из контекста. Обратите внимание, что загружать нужно содержимое всех буферов, данные которых изменяются в процессе работы кернела.

```
      //--- постановка кернела в очередь выполнения
      uint off_set[] = {0};
      uint NDRange[] = {m_cOutputs.Total()};
      if(!m_cOpenCL.Execute(def_k_PerceptronFeedForward, 1, off_set, NDRange))
         return false;
     }
//---
   return m_cActivation.Activation(m_cOutputs);
  }
```

После выполнения вышеописанных операций мы вызываем метод активации нейронов необходимого класса функции активации и выходим из метода.

Также нужно дополнить код методов обратного прохода. В кернеле вычисления градиента на выходе нейронной сети используется три буфера: целевых значений, результатов последнего прямого прохода и для записи полученных градиентов. Их мы проверим в начале блока *OpenCL*.

```
bool CNeuronBase::CalcOutputGradient(CBufferType* target, ENUM_LOSS_FUNCTION loss)
  {
//--- блок контролей
   if(!target || !m_cOutputs || !m_cGradients ||
      target.Total() < m_cOutputs.Total() ||
      m_cGradients.Total() < m_cOutputs.Total())
      return false;

//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      switch(loss)
        {
         case LOSS_MAE:
            m_cGradients.m_mMatrix = target.m_mMatrix - m_cOutputs.m_mMatrix;
            break;
         case LOSS_MSE:
            m_cGradients.m_mMatrix = (target.m_mMatrix - m_cOutputs.m_mMatrix) * 2;
            break;
         case LOSS_CCE:
            m_cGradients.m_mMatrix=target.m_mMatrix/(m_cOutputs.m_mMatrix+FLT_MIN)*
                                   log(m_cOutputs.m_mMatrix) * (-1);
            break;
         case LOSS_BCE:
            m_cGradients.m_mMatrix = (target.m_mMatrix-m_cOutputs.m_mMatrix)/
                                   (MathPow(m_cOutputs.m_mMatrix,2) -
                                    m_cOutputs.m_mMatrix+FLT_MIN);
            break;
         default:
            m_cGradients.m_mMatrix = target.m_mMatrix - m_cOutputs.m_mMatrix;
            break;
        }
     }

   else // Блок OpenCL
     {
      //--- проверка буферов данных
      if(target.GetIndex() < 0)
         return false;
      if(m_cOutputs.GetIndex() < 0)
         return false;
      if(m_cGradients.GetIndex() < 0)
         return false;
```

Далее укажем их индексы в параметрах нашего кернела. Также в параметрах кернела укажем используемую функцию потерь.

```
         //--- передача аргументов кернелу
         if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcOutputGradient, def_outgr_target,
                                                                   target.GetIndex()))
            return false;
         if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcOutputGradient, def_outgr_outputs,
                                                          m_cOutputs.GetIndex()))
            return false;
         if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcOutputGradient,def_outgr_gradients,
                                                        m_cGradients.GetIndex()))
            return false;
         if(!m_cOpenCL.SetArgument(def_k_CalcOutputGradient, def_outgr_loss_function,
                                                                  (int)loss))
            return false;
```

Количество запускаемых независимых потоков операций равно количеству нейронов на выходе нашей модели.

Запускаем выполнение кернела и завершаем метод.

```
         //--- постановка кернела в очередь выполнения
         uint NDRange[] = { m_cOutputs.Total() };
         uint off_set[] = {0};
         if(!m_cOpenCL.Execute(def_k_CalcOutputGradient, 1, off_set, NDRange))
            return false;
        }
  //---
     return true;
    }
```

Процесс распределения градиента через скрытый слой к нейронам предыдущего слоя у нас разделен на два подпроцесса. В первом мы скорректируем градиент ошибки на производную функции активации, а во втором уже распределим значение градиента ошибки на нейроны предыдущего слоя в соответствии с их влиянием на конечный результат. Для каждого подпроцесса мы создали отдельный кернел. Корректировку градиента ошибки на производную функции активации мы вынесли в отдельный класс функции активации. Поэтому в методе *CalcHiddenGradient* нам предстоит организовать запуск только кернела распределения градиента ошибки в программе *OpenCL*.

```
bool CNeuronBase::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
//--- корректировка входящего градиента на производную функции активации
   if(!m_cActivation.Derivative(m_cGradients))
      return false;
//--- проверка буферов предыдущего слоя
   if(!prevLayer)
      return false;
   CBufferType *input_data = prevLayer.GetOutputs();
   CBufferType *input_gradient = prevLayer.GetGradients();
   if(!input_data || !input_gradient ||
      input_data.Total() != input_gradient.Total())
      return false;
//--- проверка соответствия размера буфера исходных данных и матрицы весов
   if(!m_cWeights || m_cWeights.Cols() != (input_data.Total() + 1))
      return false;
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      MATRIX grad = m_cGradients.m_mMatrix.MatMul(m_cWeights.m_mMatrix);
      grad.Reshape(input_data.Rows(), input_data.Cols());
      input_gradient.m_mMatrix = grad;
     }
```

В начале блока *OpenCL* мы, как и ранее, проверяем наличие ранее созданных буферов в контексте *OpenCL* для работы текущего кернела.

```
   else // Блок OpenCL
      {
       //--- проверка буферов данных
       if(m_cWeights.GetIndex() < 0)
          return false;
       if(input_gradient.GetIndex() < 0)
          return false;
       if(m_cGradients.GetIndex() < 0)
          return false;
```

После успешного прохождения контрольного блока передадим в кернел хендлы буферов и количество нейронов в слое.

```
      //--- передача аргументов кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcHiddenGradient,
                              def_hidgr_gradient_inputs, input_gradient.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcHiddenGradient, def_hidgr_weights,
                                                              m_cWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcHiddenGradient,def_hidgr_gradients,
                                                            m_cGradients.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_CalcHiddenGradient, def_hidgr_outputs_total,
                                                              m_cGradients.Total()))
         return false;
```

Количество потоков в данном случае будет равняться количеству нейронов в предыдущем слое. Их значение мы и запишем в первый элемент массива *NDRange*. Запустим выполнение операций кернела.

```
      //--- постановка кернела в очередь выполнения
      uint NDRange[] = {input_data.Total()};
      uint off_set[] = {0};
      if(!m_cOpenCL.Execute(def_k_CalcHiddenGradient, 1, off_set, NDRange))
         return false;
     }
//---
   return true;
  }
```

После распределения градиента ошибки по всем нейронам нашей сети в соответствии с их влиянием на конечный результат нам остается организовать процесс обновления матрицы весов. Этот процесс мы разделили на два подпроцесса. Причины такого решения мы уже не раз обсуждали. Но повторюсь, матрица весов не всегда будет обновляться после каждой итерации. Поэтому на каждой итерации мы считаем градиент ошибки для каждого весового коэффициента и складываем в отдельный буфер. По команде от основной программы корректируем матрицу весов на размер пакета, что даст нам среднеарифметическое значение из накопленного градиента ошибки.

Накопление градиентов ошибки осуществляется в методе *CalcDeltaWeights*. Для выполнения операций кернела этого метода нам потребуются три буфера:

- буфер результатов последнего прямого прохода предыдущего слоя,
- буфер градиентов текущего слоя,
- буфер для накопления градиентов весовых коэффициентов.

```
bool CNeuronBase::CalcDeltaWeights(CNeuronBase *prevLayer, bool read);
  {
//--- блок контролей
   if(!prevLayer || !m_cDeltaWeights || !m_cGradients)
      return false;
   CBufferType *Inputs = prevLayer.GetOutputs();
   if(!Inputs)
      return false;
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      MATRIX m = Inputs.m_mMatrix;
      m.Resize(1, Inputs.Total() + 1);
      m[0, Inputs.Total()] = 1;
      m = m_cGradients.m_mMatrix.Transpose().MatMul(m);
      m_cDeltaWeights.m_mMatrix += m;
     }
```

Вначале уже по традиции проверяем наличие используемых буферов в контексте *OpenCL*.

```
   else // Блок OpenCL
     {
      //--- проверка буферов данных
      if(m_cGradients.GetIndex() < 0)
         return false;
      if(m_cDeltaWeights.GetIndex() < 0)
         return false;
      if(Inputs.GetIndex() < 0)
         return false;
```

Передаем указатели на них в параметры кернела.

```
      //--- передача аргументов кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcDeltaWeights,
                              def_delt_delta_weights, m_cDeltaWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcDeltaWeights, def_delt_inputs,
                                                              Inputs.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_CalcDeltaWeights, def_delt_gradients,
                                                          m_cGradients.GetIndex()))
         return false;
```

В данном случае для запуска кернела мы будем использовать двухмерное пространство задач. В одном измерении укажем количество нейронов к текущем слое, а во втором — количество нейронов в предыдущем слое.

После завершения подготовительной работы запустим выполнение кернела.

Затем проверим флаг чтения данных и при необходимости загрузим результат операций из контекста.

И конечно, не забываем на каждом шаге контролировать процесс выполнения операций.

```
        //--- постановка кернела в очередь выполнения
        uint NDRange[] = {m_cGradients.Total(), Inputs.Total()};
        uint off_set[] = {0, 0};
        if(!m_cOpenCL.Execute(def_k_CalcDeltaWeights, 2, off_set, NDRange))
           return false;
        if(read && !m_cDeltaWeights.BufferRead())
           return false;
      }
  //---
     return true;
    }
```

Мы успешно двигаемся вперед в процессе создания нашего проекта. Для завершения работы над полносвязным нейроном нам остается описать подпроцесс обновления матрицы весов. Но в своем проекте мы решили реализовать несколько алгоритмом обновления весовых коэффициентов. Для каждого алгоритма обновления матрицы весов мы создали свой кернел. Добавим вызовы этих кернелов в соответствующие методы нашего класса.

Начнем мы с метода стохастического градиентного спуска. Реализация данного метода требует лишь двух буферов: накопленных дельт и матрицы весов. Наличие данных буферов мы проверяем в контексте *OpenCL*.

```
bool CNeuronBase::SGDUpdate(int batch_size, TYPE learningRate, VECTOR &Lambda)
  {
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      TYPE lr = learningRate / ((TYPE)batch_size);
      m_cWeights.m_mMatrix -= m_cWeights.m_mMatrix * Lambda[1] + Lambda[0];
      m_cWeights.m_mMatrix += m_cDeltaWeights.m_mMatrix * lr;
      m_cDeltaWeights.m_mMatrix.Fill(0);
     }
   else // Блок OpenCL
     {
      //--- проверка буферов данных
      if(m_cWeights.GetIndex() < 0)
         return false;
      if(m_cDeltaWeights.GetIndex() < 0)
         return false;
```

Затем передадим в параметры кернела указатели на них. Кроме того, нам нужно передать в кернел параметры обучения:

- размер пакета — *batch_size*
- коэффициент обучения — *learningRate*
- параметры регуляризации — вектор *Lambda*

```
      //--- передача аргументов кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_SGDUpdate, def_sgd_delta_weights,
                                                    m_cDeltaWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_SGDUpdate, def_sgd_weights,
                                                       m_cWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_SGDUpdate, def_sgd_total,
                                                      (int)m_cWeights.Total()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_SGDUpdate, def_sgd_batch_size, batch_size))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_SGDUpdate, def_sgd_learningRate,
                                                                learningRate))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_SGDUpdate, def_sgd_Lambda1, Lambda[0]))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_SGDUpdate, def_sgd_Lambda2, Lambda[1]))
         return false;
```

Определим количество запускаемых потоков. Их будет в 4 раза меньше количества элементов матрицы весов. Такой эффект достигается благодаря использованию векторных операций.

Немного слов об алгоритме определения количества потоков. Мы не можем просто разделить число нейронов на четыре, ведь мы не можем быть уверены, что число нейронов всегда будет кратно четырем. Но мы должны быть уверены, что количество потоков покроет все нейроны нашего слоя. Поэтому нам потребуется функция, аналогичная округлению до целого числа вверх. Но вместо этого мы воспользуемся свойством целочисленного деления отбрасывать дробную часть, то есть округлением вниз. Чтобы получить желаемый результат, перед делением на размер вектора мы прибавим к количеству нейронов число на 1 больше размера вектора. После такого небольшого математического трюка результатом целочисленного деления и будет требуемое число потоков. При использовании этого трюка нужно с особой аккуратностью подходить к типу используемых данных, ведь нужный эффект можно получить только тогда, когда все переменные операции целочисленные.

```
      //--- постановка кернела в очередь выполнения
      int NDRange[] = { (int)((m_cWeights.Total() + 3) / 4) };
      int off_set[] = {0};
      if(!m_cOpenCL.Execute(def_k_SGDUpdate, 1, off_set, NDRange))
         return false;
     }
   return true;
  }
```

После подготовительной работы запросим выполнение кернела.

В описании процесса обновления матрицы весов по методу накопленного импульса появляется дополнительный буфер для хранения моментов и коэффициент усреднения импульса. В остальном сохраняются принципы построения алгоритма, заложенные в предыдущем методе.

```
bool CNeuronBase::MomentumUpdate(int batch_size, TYPE learningRate,
                                 VECTOR &Beta, VECTOR &Lambda)
  {
   if(Beta[0] == 0)
      return SGDUpdate(batch_size, learningRate, Lambda);
//--- блок контролей
   if(!m_cMomenum[0])
      return false;
   if(m_cMomenum[0].Total() < m_cWeights.Total())
      return false;
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      TYPE lr = learningRate / ((TYPE)batch_size);
      m_cWeights.m_mMatrix -= m_cWeights.m_mMatrix * Lambda[1] + Lambda[0];
      m_cMomenum[0].m_mMatrix = m_cDeltaWeights.m_mMatrix * lr +
                                      m_cMomenum[0].m_mMatrix * Beta[0] ;
      m_cWeights.m_mMatrix += m_cMomenum[0].m_mMatrix;
      m_cDeltaWeights.m_mMatrix.Fill(0);
     }
   else // Блок OpenCL
     {
      //--- проверка буферов данных
      if(m_cWeights.GetIndex() < 0)
         return false;
      if(m_cDeltaWeights.GetIndex() < 0)
         return false;
      if(m_cMomenum[0].GetIndex() < 0)
         return false;
```

```
      //--- передача аргументов кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_MomentumUpdate,
                          def_moment_delta_weights, m_cDeltaWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_MomentumUpdate, def_moment_weights,
                                                        m_cWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_MomentumUpdate,
                                def_moment_momentum, m_cMomenum[0].GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_MomentumUpdate, def_moment_total,
                                                       (int)m_cWeights.Total()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_MomentumUpdate, def_moment_batch_size,
                                                                batch_size))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_MomentumUpdate, def_moment_learningRate,
                                                             learningRate))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_MomentumUpdate, def_moment_Lambda1,
                                                                Lambda[0]))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_MomentumUpdate, def_moment_Lambda2,
                                                                Lambda[1]))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_MomentumUpdate, def_moment_beta, Beta[0]))
         return false;
```

Задаем количество потоков в 4 раза меньше количества элементов в матрице весов и запускаем выполнение операций.

```
      //--- постановка кернела в очередь выполнения
      int NDRange[] = { (int)((m_cWeights.Total() + 3) / 4) };
      int off_set[] = {0};
      if(!m_cOpenCL.Execute(def_k_MomentumUpdate, 1, off_set, NDRange))
         return false;
     }
   return true;
  }
```

Также следует обратить внимание на такие детали, как используемые константы кернелов и их параметров. Несмотря на схожесть операций часто какая-то маленькая деталь, опечатка с какой-либо константой может привести к критической ошибке и остановке программы.

Перейдем к следующей реализации. Метод оптимизации *AdaGrad* реализован в методе *AdaGradUpdate* и соответствующем кернеле, который мы будем идентифицировать константой *def_k_AdaGradUpdate*. Чтобы исключить возможные ошибки при указании параметров, все константы параметров к данному кернелу начинаются с *def_adagrad_*. Как можно заметить, все наименования констант интуитивно понятны и имеют логическую связь. Это снижает риск возможной ошибки. Такой прием очень удобен при наличии большого количества констант.

Метод *AdaGrad,* как и метод накопительного импульса, использует буфер накопления моментов. Но в отличие от предыдущего метода здесь отсутствует коэффициент усреднения. На данном

этапе нам не важны различия в использовании параметров и буферов. Нас интересует лишь их наличие — использование буферов и параметров уже описано в кернеле программы *OpenCL*, а здесь мы организовываем процесс передачи данных из основной программы в контекст *OpenCL*.

Алгоритм организации процесса роботы с контекстом *OpenCL* в методе *AdaGradUpdate* аналогичен примененному в ранее описанных методах.

- Вначале проверяем наличие буферов в контексте *OpenCL*.
- Затем в кернел передаем указатели на буферы и параметры оптимизации.
- Запускаем выполнение кернела.

```
bool CNeuronBase::AdaGradUpdate(int batch_size, TYPE learningRate, VECTOR &Lambda)
  {
//--- блок контролей
   if(!m_cMomenum[0])
      return false;
   if(m_cMomenum[0].Total() < m_cWeights.Total())
      return false;
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      m_cWeights.m_mMatrix -= m_cWeights.m_mMatrix * Lambda[1] + Lambda[0];
      MATRIX delta = m_cDeltaWeights.m_mMatrix / ((TYPE)batch_size);
      MATRIX G = m_cMomenum[0].m_mMatrix = m_cMomenum[0].m_mMatrix + delta.Power(2);
      G = MathPow(MathSqrt(G) + 1e-32, -1);
      G = G * learningRate;
      m_cWeights.m_mMatrix += G * delta;
       m_cDeltaWeights.m_mMatrix.Fill(0);
    }

   else // Блок OpenCL
     {
      //--- проверка буферов данных
      if(m_cWeights.GetIndex() < 0)
         return false;
      if(m_cDeltaWeights.GetIndex() < 0)
         return false;
      if(m_cMomenum[0].GetIndex() < 0)
         return false;
```

```
      //--- передача аргументов кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AdaGradUpdate,
                           def_adagrad_delta_weights, m_cDeltaWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AdaGradUpdate, def_adagrad_weights,
                                                          m_cWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AdaGradUpdate, def_adagrad_momentum,
                                                      m_cMomenum[0].GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdaGradUpdate, def_adagrad_total,
                                                        (int)m_cWeights.Total()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdaGradUpdate, def_adagrad_batch_size,
                                                                 batch_size))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdaGradUpdate, def_adagrad_learningRate,
                                                               learningRate))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdaGradUpdate, def_adagrad_Lambda1,
                                                                  Lambda[0]))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdaGradUpdate, def_adagrad_Lambda2,
                                                                  Lambda[1]))
         return false;

      //--- постановка кернела в очередь выполнения
      int NDRange[] = { (int)((m_cWeights.Total() + 3) / 4) };
      int off_set[] = {0};
      if(!m_cOpenCL.Execute(def_k_AdaGradUpdate, 1, off_set, NDRange))
         return false;
     }
   return true;
  }
```

Метод оптимизации RMSProp по своему функционалу напоминает AdaGrad, но в нем добавляется коэффициент усреднения накопленного момента.

Действуем по отработанной схеме. Проверяем наличие буферов контекста *OpenCL*. Затем передаем в кернел указатели на буферы и параметры оптимизации. При этом отслеживаем соблюдение имени метода и используемых констант:

- метод ***RMS**PropUpdate*,
- константа кернела *def_k_**RMS**PropUpdate*,
- константы параметров *def_**rms**_…*

После указания параметров запускаем выполнение кернела.

```
bool CNeuronBase::RMSPropUpdate(int batch_size, TYPE learningRate,
                                VECTOR &Beta, VECTOR &Lambda)
  {
//--- блок контролей
   if(!m_cMomenum[0])
      return false;
   if(m_cMomenum[0].Total() < m_cWeights.Total())
      return false;
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      TYPE lr = learningRate;
      m_cWeights.m_mMatrix -= m_cWeights.m_mMatrix * Lambda[1] + Lambda[0];
      MATRIX delta = m_cDeltaWeights.m_mMatrix / ((TYPE)batch_size);
      MATRIX G = m_cMomenum[0].m_mMatrix = m_cMomenum[0].m_mMatrix * Beta[0] +
                                           delta.Power(2) * (1 - Beta[0]);
      G = MathPow(MathSqrt(G) + 1e-32, -1);
      G = G * learningRate;
      m_cWeights.m_mMatrix += G * delta;
      m_cDeltaWeights.m_mMatrix.Fill(0);
     }
   else // Блок OpenCL
     {
      //--- проверка буферов данных
      if(m_cWeights.GetIndex() < 0)
         return false;
      if(m_cDeltaWeights.GetIndex() < 0)
         return false;
      if(m_cMomenum[0].GetIndex() < 0)
         return false;
```

```
      //--- передача аргументов кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_RMSPropUpdate, def_rms_delta_weights,
                                                      m_cDeltaWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_RMSPropUpdate, def_rms_weights,
                                                        m_cWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_RMSPropUpdate, def_rms_momentum,
                                                     m_cMomenum[0].GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_RMSPropUpdate, def_rms_total,
                                                       (int)m_cWeights.Total()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_RMSPropUpdate, def_rms_batch_size,
                                                              batch_size))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_RMSPropUpdate, def_rms_learningRate,
                                                           learningRate))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_RMSPropUpdate, def_rms_Lambda1, Lambda[0]))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_RMSPropUpdate, def_rms_Lambda2, Lambda[1]))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_RMSPropUpdate, def_rms_beta, Beta[0]))
         return false;

      //--- постановка кернела в очередь выполнения
      int NDRange[] = { (int)((m_cWeights.Total() + 3) / 4) };
      int off_set[] = {0};
      if(!m_cOpenCL.Execute(def_k_RMSPropUpdate, 1, off_set, NDRange))
         return false;
     }
//---
   return true;
  }
```

Разработчики методе *AdaDelta* отказались от использования обучающего коэффициента, но заплатили за это введение дополнительного буфера моментов дополнительным коэффициентом усреднения. Соответственно, в данном кернеле мы будем использовать на один буфер больше.

При задании параметров кернела, как и раньше, следим за соблюдением наименования констант:

- метод ***AdaDelta****Update,*
- константа кернела *def_k_**AdaDelta**Update,*
- константы параметров *def_**adadelt**...*

Кроме того, для возможности безболезненного переноса построенной нейронной сети нам нужно проследить за соответствием использования буферов в части выполнения операций средствами *MQL5* и в контексте *OpenCL*. При использовании в рамках одной платформы изменение последовательности использования массивов моментов не окажет влияния. Как бы мы их ни называли, их содержимое будет соответствовать контексту использования. Но при переносе предварительно обученной нейронной сети на другую платформу велика вероятность получить неожиданные результаты. При этом следует помнить о назначении и функционале

массивов. Моменты используются только при обновлении матрицы весов в процессе обучения нейронной сети и не участвуют в прямом проходе. То есть влияние перепутанных буфером мы сможем увидеть только при попытке дообучить нейронную сеть. Но не стоит этим пренебрегать. При длительном использовании однажды построенной нейронной сети нам нужно будет периодически проводить ее дообучение. Это необходимо для поддержания актуальности весовых коэффициентов в нашем изменчивом мире.

Учтем вышесказанное и передадим в кернел указатели на загруженные буферы и параметры обучения.

Рассчитаем количество необходимых потоков и запустим кернел на выполнения.

```
bool CNeuronBase::AdaDeltaUpdate(int batch_size, VECTOR &Beta, VECTOR &Lambda)
  {
//--- блок контролей
   for(int i = 0; i < 2; i++)
     {
      if(!m_cMomenum[i])
         return false;
      if(m_cMomenum[i].Total() < m_cWeights.Total())
         return false;
     }
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      MATRIX delta = m_cDeltaWeights.m_mMatrix / ((TYPE)batch_size);
      MATRIX W = m_cMomenum[0].m_mMatrix = m_cMomenum[0].m_mMatrix * Beta[0] +
                                 m_cWeights.m_mMatrix.Power(2) * (1 - Beta[0]);
      m_cMomenum[1].m_mMatrix = m_cMomenum[1].m_mMatrix * Beta[1] +
                                                 delta.Power(2) * (1 - Beta[1]);
      m_cWeights.m_mMatrix -= m_cWeights.m_mMatrix * Lambda[1] + Lambda[0];
      W = MathSqrt(W) / (MathSqrt(m_cMomenum[1].m_mMatrix) + 1e-32);
      m_cWeights.m_mMatrix += W * delta;
      m_cDeltaWeights.m_mMatrix.Fill(0);
     }

   else // Блок OpenCL
     {
      //--- создание буферов данных
      if(m_cWeights.GetIndex() < 0)
         return false;
      if(m_cDeltaWeights.GetIndex() < 0)
         return false;
      if(m_cMomenum[0].GetIndex() < 0)
         return false;
      if(m_cMomenum[1].GetIndex() < 0)
         return false;
```

```
      //--- передача аргументов кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AdaDeltaUpdate,
                           def_adadelt_delta_weights, m_cDeltaWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AdaDeltaUpdate, def_adadelt_weights,
                                                          m_cWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AdaDeltaUpdate, def_adadelt_momentumW,
                                                       m_cMomenum[0].GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AdaDeltaUpdate, def_adadelt_momentumG,
                                                       m_cMomenum[1].GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdaDeltaUpdate, def_adadelt_total,
                                                         (int)m_cWeights.Total()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdaDeltaUpdate, def_adadelt_batch_size,
                                                                  batch_size))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdaDeltaUpdate, def_adadelt_Lambda1,
                                                                   Lambda[0]))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdaDeltaUpdate, def_adadelt_Lambda2,
                                                                   Lambda[1]))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdaDeltaUpdate, def_adadelt_beta1, Beta[0]))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdaDeltaUpdate, def_adadelt_beta2, Beta[1]))
         return false;

      //--- постановка кернела в очередь выполнения
      int NDRange[] = { (int)((m_cWeights.Total() + 3) / 4) };
      int off_set[] = {0};
      if(!m_cOpenCL.Execute(def_k_AdaDeltaUpdate, 1, off_set, NDRange))
         return false;
     }
//---
   return true;
  }
```

Наша работа по описанию операций, выполняемых в полносвязном нейронном слое, подходит к завершению. Осталось описать один метод обновления весовых коэффициентов. Как говорится, последний по списку, но не по значению метод обновления весовых коэффициентов *Adam*. Метод также использует два буфера моментов, но по сравнению с *AdaDelta* в него возвращается коэффициент обучения.

Повторим еще раз основные этапы нашего алгоритма и ключевые моменты контролей:

- проверяем наличие необходимых данных в памяти контекста *OpenCL;*
- передаем в кернел указатели на буферы данных и параметры обучения. При этом отслеживаем соответствие наименований Метод ***Adam****Update* → константа кернела *def_k_**Adam**Update* → константы параметров *def_**adam**_*...;

- контролируем соответствие использования буферов средствами *MQL5* и в контексте *OpenCL;*
- запускаем выполнение кернела.

```
bool CNeuronBase::AdamUpdate(int batch_size, TYPE learningRate,
                             VECTOR &Beta, VECTOR &Lambda)
  {
//--- блок контролей
   for(int i = 0; i < 2; i++)
     {
      if(!m_cMomenum[i])
         return false;
      if(m_cMomenum[i].Total() != m_cWeights.Total())
         return false;
     }
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      MATRIX delta = m_cDeltaWeights.m_mMatrix / ((TYPE)batch_size);
      m_cMomenum[0].m_mMatrix = m_cMomenum[0].m_mMatrix * Beta[0] +
                                                          delta * (1 - Beta[0]);
      m_cMomenum[1].m_mMatrix = m_cMomenum[1].m_mMatrix * Beta[1] +
                                             MathPow(delta,2) * (1 - Beta[1]);
      MATRIX M = m_cMomenum[0].m_mMatrix / (1 - Beta[0]);
      MATRIX V = m_cMomenum[1].m_mMatrix / (1 - Beta[1]);
      m_cWeights.m_mMatrix -= m_cWeights.m_mMatrix * Lambda[1] + Lambda[0];
      m_cWeights.m_mMatrix += M * learningRate  / MathSqrt(V);
      m_cDeltaWeights.m_mMatrix.Fill(0);
     }

   else // Блок OpenCL
     {
      //--- проверка буферов данных
      if(m_cWeights.GetIndex() < 0)
         return false;
      if(m_cDeltaWeights.GetIndex() < 0)
         return false;
      if(m_cMomenum[0].GetIndex() < 0)
         return false;
      if(m_cMomenum[1].GetIndex() < 0)
         return false;
```

```
      //--- передача аргументов кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AdamUpdate, def_adam_delta_weights,
                                                      m_cDeltaWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AdamUpdate, def_adam_weights,
                                                        m_cWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AdamUpdate, def_adam_momentumM,
                                                      m_cMomenum[0].GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AdamUpdate, def_adam_momentumV,
                                                      m_cMomenum[1].GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdamUpdate, def_adam_total,
                                                       (int)m_cWeights.Total()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdamUpdate, def_adam_batch_size,
                                                                batch_size))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdamUpdate, def_adam_Lambda1, Lambda[0]))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdamUpdate, def_adam_Lambda2, Lambda[1]))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdamUpdate, def_adam_beta1, Beta[0]))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdamUpdate, def_adam_beta2, Beta[1]))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AdamUpdate, def_adam_learningRate,
                                                            learningRate))
         return false;

      //--- постановка кернела в очередь выполнения
      int NDRange[] = { (int)((m_cWeights.Total() + 3) / 4) };
      int off_set[] = {0};
      if(!m_cOpenCL.Execute(def_k_AdamUpdate, 1, off_set, NDRange))
         return false;
     }
//---
   return true;
  }
```

Мы завершили описание процессов полносвязного нейронного слоя. Сейчас мы подошли к тому этапу, когда можно посмотреть на проделанную работу и оценить первые результаты. На самом деле уже созданных базовых классов достаточно, чтобы построить небольшую модель перцептрона из нескольких полносвязных слоев. Один из них будет выполнять роль приемника исходных данных (входной слой), последний нейронный слой будет выдавать результаты (выходной слой), а между ними будут скрытые слои.

3.8 Реализация модели перцептрона в Python

Для реализации модели полносвязного перцептрона на языке Python воспользуемся ранее созданным нами шаблоном. Как вы помните, в шаблоне мы оставили не заполненными описание нейронных слоев нашей модели.

```
# Создание модели нейронной сети
model = keras.Sequential([keras.Input(shape=inputs),
                          # Наполнить модель описанием нейронных слоев
                          ])
```

Для создания полносвязных слоев в нейронной сети библиотека *Keras* предоставляет класс layers.Dense. Внутри данного слоя реализуется операция:

$$Output = activation(dot(input, kernel) + bias)$$

где:

- ***activation*** — функция активации, задается в параметрах;
- ***input*** — массив исходных данных;
- ***kernel*** — матрица весов;
- ***dot*** — операция векторного умножения;
- ***bias*** — элемент смещения.

Для управления процессом создания нейронного слоя *Dense* имеет ряд параметров:

- *units* — размерность выходного пространства (количество нейронов в слое);
- *activation* — используемая функция активации;
- *use_bias* — необязательный параметр, указывающий на необходимость использования вектора элементов смещения;
- *kernel_initializer* — метод инициализации матрицы весов;
- *bias_initializer* — метод инициализации вектора элементов смещения;
- *kernel_regularizer* — метод регуляризации матрицы весов;
- *bias_regularizer* — метод регуляризации вектора смещения;
- *activity_regularizer* — метод регуляризации функции активации;
- *kernel_constraint* — функция ограничения матрицы весов;
- *bias_constraint* — функция ограничения вектора смещения.

Следует обратить внимание, что изменение параметров после первого обращения к слою невозможно.

Дополнительно к указанным выше параметрам *Dense* может принимать параметр *input_shape*, указывающий на размер массива входных данных. Параметр допустим только для первого слоя нейронной сети. В случае использования параметра создается входной слой для вставки перед текущим слоем. Операцию можно рассматривать как эквивалент явного определения входного слоя.

Реализацию своей первой модели нейронной сети мы начнем с копирования нашего шаблона скрипта в новый файл *perceptron.py*. В созданном файле создадим первую модель с одним скрытым слоем из 40 нейронов и 2 нейронами в слое результатов. В скрытом слое будем

использовать *Swish* в качестве функции активации. Нейроны выходного слоя будут активироваться гиперболическим тангенсом.

```
# Создание модели нейронной сети
model1 = keras.Sequential([keras.Input(shape=inputs),
                           keras.layers.Dense(40, activation=tf.nn.swish),
                           keras.layers.Dense(targerts, activation=tf.nn.tanh)
                          ])
```

В принципе этого достаточно для запуска обучения модели. Но мы с вами изучаем работу различных моделей и хотим понять влияние изменения архитектуры нейронной сети на способность модели обучаться и обобщать исходные данные. Поэтому я добавил еще две модели. В одной модели я добавил еще два скрытых слоя. В итоге получилась модель с тремя скрытыми слоями. Все три скрытых слоя полностью идентичны: имеют по 40 элементов и активируются функцией *Swish*. Первый и последний слои остались без изменений.

```
# Создание модели с тремя скрытыми слоями
model2 = keras.Sequential([keras.Input(shape=inputs),
                           keras.layers.Dense(40, activation=tf.nn.swish),
                           keras.layers.Dense(40, activation=tf.nn.swish),
                           keras.layers.Dense(40, activation=tf.nn.swish),
                           keras.layers.Dense(targerts, activation=tf.nn.tanh)
                          ])
```

И конечно, нам нужно будет повторить и все последующие шаги для каждой модели. Сначала подготовим модель к обучению с помощью метода *compile*.

```
model2.compile(optimizer='Adam',
               loss='mean_squared_error',
               metrics=['accuracy'])
```

После этого запустим процесс обучения модели и сохраним обученную модель.

```
history2 = model2.fit(train_data, train_target,
                      epochs=500, batch_size=1000,
                      callbacks=[callback],
                      verbose=2,
                      validation_split=0.2,
                      shuffle=True)
model2.save(os.path.join(path,'perceptron2.h5'))
```

Третью модель построим на базе второй с добавлением регуляризации. Для каждого нейронного слоя укажем в параметре *kernel_regularizer* объект класса *keras.regularizers.l1_l2* с параметрами *L1* и *L2*-регуляризации. Как видно из названия класса, мы будем использовать *ElasticNet*.

```
# Добавляем регуляризацию в модель с тремя скрытыми слоями
model3 = keras.Sequential([keras.Input(shape=inputs),
               keras.layers.Dense(40, activation=tf.nn.swish,
                  kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
               keras.layers.Dense(40, activation=tf.nn.swish,
                  kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
               keras.layers.Dense(40, activation=tf.nn.swish,
                  kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
               keras.layers.Dense(targerts, activation=tf.nn.tanh)
```

```
                              ])
```

Далее скомпилируем и обучим модель. Для всех трех моделей используются идентичные параметры обучения. Это позволит оценить непосредственно влияние архитектуры модели на результат обучения. При этом максимально исключим влияние прочих факторов.

```
model3.compile(optimizer='Adam',
               loss='mean_squared_error',
               metrics=['accuracy'])
history3 = model3.fit(train_data, train_target,
                      epochs=500, batch_size=1000,
                      callbacks=[callback],
                      verbose=2,
                      validation_split=0.2,
                      shuffle=True)
model3.save(os.path.join(path,'perceptron3.h5'))
```

Так как в данном скрипте мы обучаем не одну, а целых три модели, нам нужно подправить и блок визуализации. Выведем на один график результаты обучения всех трех моделей. Это продемонстрирует отличия в процессе обучения и валидации. Изменение внесем в блоки построения обоих графиков.

```
# Отрисовка результатов обучения трех моделей
plt.figure()
plt.plot(history1.history['loss'],
         label='Train 1 hidden layer')
plt.plot(history1.history['val_loss'],
         label='Validation 1 hidden layer')
plt.plot(history2.history['loss'],
         label='Train 3 hidden layers')
plt.plot(history2.history['val_loss'],
         label='Validation 3 hidden layers')
plt.plot(history3.history['loss'],
         label='Train 3 hidden layers vs regularization')
plt.plot(history3.history['val_loss'],
         label='Validation 3 hidden layer vs regularization')
plt.ylabel('$MSE$ $Loss$')
plt.xlabel('$Epochs$')
plt.title('Dynamic of Models train')
plt.legend(loc='lower left')

plt.figure()
plt.plot(history1.history['accuracy'],
         label='Train 1 hidden layer')
plt.plot(history1.history['val_accuracy'],
         label='Validation 1 hidden layer')
plt.plot(history2.history['accuracy'],
         label='Train 3 hidden layers')
plt.plot(history2.history['val_accuracy'],
         label='Validation 3 hidden layers')
plt.plot(history3.history['accuracy'],
         label='Train 3 hidden layers\nvs regularization')
plt.plot(history3.history['val_accuracy'],
         label='Validation 3 hidden layer\nvs regularization')
plt.ylabel('$Accuracy$')
plt.xlabel('$Epochs$')
plt.title('Dynamic of Models train')
plt.legend(loc='upper left')
```

После обучения в нашем шаблоне идет проверка работоспособности модели на тестовой выборке. Здесь нам тоже предстоит провести тестирование трех моделей в схожих условиях. Я пропущу блок загрузки тестовой выборки, так как он без изменений перешел из шаблона. Приведу лишь код непосредственного тестирования моделей.

```
# Проверка результатов моделей на тестовой выборке
test_loss1, test_acc1 = model1.evaluate(test_data,
                                        test_target,
                                        verbose=2)
test_loss2, test_acc2 = model2.evaluate(test_data,
                                        test_target,
                                        verbose=2)
test_loss3, test_acc3 = model3.evaluate(test_data,
                                        test_target,
                                        verbose=2)
```

Результаты тестирования в шаблоне выводились в журнал. Но сейчас мы имеем результаты тестирования трех моделей. Более наглядно будет сравнить результаты на графике. Для построения графиков воспользуемся средствами библиотеки *Matplolib*.

В данном случае мы будем отображать не динамику процесса, как ранее, а сравнивать величины. Следовательно, для отображения величин удобнее будет приметь столбчатую диаграмму. Для построения диаграмм библиотека предлагает метод *bar*. Этот метод в параметрах принимает два массива: в первом мы укажем метки сравниваемых параметров, а во втором — их величины. Для полноты картины добавим название графика и вертикальной оси с помощью методов *title* и *ylabel*, соответственно.

```
plt.figure()
plt.bar(
    ['1 hidden layer','3 hidden layers', '3 hidden layers\nvs regularization'],
    [test_loss1,test_loss2,test_loss3])
plt.ylabel('$MSE$ $Loss$')
plt.title('Result of test')

plt.figure()
plt.bar(
    ['1 hidden layer','3 hidden layers', '3 hidden layers\nvs regularization'],
    [test_acc1,test_acc2,test_acc3])
plt.ylabel('$Accuracy$')
plt.title('Result of test')
```

С работой скрипта познакомимся немного позже. В следующей главе мы подготовим данные для обучения и тестирования моделей.

3.9 Создание обучающей и тестовой выборки

Мы уже прошли довольно большой путь в создании нашей библиотеки для построения нейронных сетей. Мы завершили работу по построению базового диспетчерского класса нашей нейронной сети и создали все необходимое для построения полносвязного нейронного слоя. Нам предстоит еще много работы. Но уже сейчас мы можем построить свою первую нейронную сеть и проверить ее работоспособность на реальных данных. Так как у нас еще будет несколько реализаций различных архитектурных решений, для сравнения результатов работы моделей мы сделаем небольшой срез данных. Создадим две выборки данных: одну побольше для обучения нейронной сети, а вторую поменьше для проверки работы обученной нейронной сети.

Выделение отдельной выборки для обучения — это общепринятая практика. В процессе обучения нейронной сети подбираются такие весовые коэффициенты, чтобы нейронная сеть

максимально точно описывала обучающую выборку. При использовании достаточно большого количества весовых коэффициентов нейронная сеть способна выучить обучающую выборку до мельчайших деталей. Но при этом нейронная сеть теряет способность обобщения данных. В таком состояние нейронную сеть называют «*переобученной*». Выявить это на обучающей выборке невозможно. Но если сравнить результаты работы нейронной сети на обучающей выборке и на данных, не входящих в обучающую выборку, то различие результатов об этом четко скажет. Допускается небольшое ухудшение результатов на тестовой выборке, но оно не должно быть кардинальным. Конечно, данные в выборках должны быть сопоставимы. Чаще всего для этого берут общую совокупность доступных данных и случайным образом делят на две выборки в соотношении 70-80% для обучающей выборки и 20-30% для тестовой. В большинстве случаев нужно будет разделить генеральную совокупность на три подвыборки:

- обучающая 60%
- валидационная 20%
- тестовая 20%

Валидационная выборка используется для отбора лучших параметров обучения и архитектуры нейронной сети. Но мы не будем сейчас использовать валидационную выборку, поскольку на данном этапе мы бы хотели сравнить различные реализации при прочих равных условиях.

Для генерации выборок создадим скрипт *create_initial_data.mq5*. В параметрах скрипта укажем:

- период для загрузки данных в виде даты начала и окончания периода — за этот период мы загрузим с сервера исторические данные и данные индикаторов;
- таймфрейм, используемый для загрузки анализируемых данных;
- количество анализируемых исторических баров на один паттерн;
- наименование файлов для записи обучающей и тестовой выборок;
- укажем флаг нормализации данных.

Ранее мы много говорили о важности нормализации данных, подаваемых на вход нейронной сети. Сейчас мы можем на практике проверить, как нормализация данных сказывается на результатах обучения нейронной сети. Именно для оценки влияния этого фактора я ввел параметр нормализации данных. Здесь следует обратить внимание, что данные, подаваемые на вход нейронной сети, должны быть сопоставимы как в тестовой и обучающей выборке, так и при промышленной эксплуатации нейронной сети. Поэтому на практике нужно будет сохранить параметры нормализации и использовать их при нормализации данных, поступающих во время промышленной эксплуатации нейронной сети.

Напомню, что в разделе о выборе исходных данных для подачи на вход нейронной сети мы отобрали два индикатора: RSI и MACD. Их мы и будем использовать в процессе обучения нейронных сетей в рамках практических экспериментов этой книги.

Посмотрим на алгоритм скрипта. Вначале по аналогии со скриптами, рассмотренными при выборе исходных данных, мы подключим отобранные индикаторы к графику и получим хендлы доступа к данным индикаторов.

```
//+------------------------------------------------------------------+
//| Внешние параметры для работы скрипта                             |
//+------------------------------------------------------------------+
// Начало периода генеральной совокупности
input datetime Start = D'2015.01.01 00:00:00';
// Окончание периода генеральной совокупности
input datetime End = D'2020.12.31 23:59:00';
// Таймфрейм для загрузки данных
input ENUM_TIMEFRAMES TimeFrame = PERIOD_M5;
// Количество исторических баров в одном паттерне
input int      BarsToLine = 40;
// Имя файла для записи обучающей выборки
input string   StudyFileName = "study_data.csv";
// Имя файла для записи тестовой выборки
input string   TestFileName  = "test_data.csv";
// Флаг нормализации данных
input bool     NormalizeData = true;
//+------------------------------------------------------------------+
//| Начало программы скрипта                                         |
//+------------------------------------------------------------------+
void OnStart(void)
  {
//--- Подключение индикаторов к графику
   int h_ZZ = iCustom(_Symbol, TimeFrame, "Examples\\ZigZag.ex5", 48, 1, 47);
   int h_RSI = iRSI(_Symbol, TimeFrame, 12, PRICE_TYPICAL);
   int h_MACD = iMACD(_Symbol, TimeFrame, 12, 48, 12, PRICE_TYPICAL);
   double close[];
   if(CopyClose(_Symbol, TimeFrame, Start, End, close) <= 0)
      return;
```

После этого проверим действительность полученных хендлов и загрузим исторические данные индикаторов в динамические массивы. При этом следует обратить внимание на тот момент, что для индикатора ZigZag мы загрузим немного больше данных. Причиной тому являются особенности данного индикатора. Буфер данного индикатора указывает только на найденные экстремумы. В остальных случаях индикатор возвращает нулевые значения. Поэтому для последних анализируемых паттернов целевые значения могут быть за пределами анализируемого периода.

```
//--- Загружаем данные индикаторов в динамические массивы
   double zz[], macd_main[], macd_signal[], rsi[];
   datetime end_zz = End + PeriodSeconds(TimeFrame) * 500;
   if(h_ZZ == INVALID_HANDLE ||
      CopyBuffer(h_ZZ, 0, Start, end_zz, zz) <= 0)
     {
      PrintFormat("Error loading indicator %s data", "ZigZag");
      return;
     }
   if(h_RSI == INVALID_HANDLE ||
      CopyBuffer(h_RSI, 0, Start, End, rsi) <= 0)
     {
      PrintFormat("Error loading indicator %s data", "RSI");
      return;
     }
   if(h_MACD == INVALID_HANDLE ||
      CopyBuffer(h_MACD, MAIN_LINE, Start, End, macd_main) <= 0 ||
      CopyBuffer(h_MACD, SIGNAL_LINE, Start, End, macd_signal) <= 0)
     {
      PrintFormat("Error loading indicator %s data", "MACD");
      return;
     }
```

Кроме отобранных индикаторов загрузим цены закрытия свечи. Их мы будем использовать для определения направления движения цены до ближайшего экстремума и силы предстоящего движения.

После загрузки данных организуем процесс определения целевых значений на каждом шаге исторических данных. Для этого организуем обратный цикл и будем перебирать все значения индикатора ZigZag и в случае отличия значения от нуля будем сохранять его в переменную extremum. Параллельно будем перебирать значения цен закрытия и по отклонению последнего сохраненного экстремума от цены закрытия будем определять направление и силу предстоящего движения. Полученные значения сохраним в динамические массивы target1 и target2.

```
   int total = ArraySize(close);
   double target1[], target2[], macd_delta[], test[];
   if(ArrayResize(target1, total) <= 0 ||
      ArrayResize(target2, total) <= 0 ||
      ArrayResize(test, total) <= 0 ||
      ArrayResize(macd_delta, total) <= 0)
      return;
//--- Рассчитываем цели: направление и расстояние
//--- до ближайшего экстремума
   double extremum = -1;
   for(int i = ArraySize(zz) - 2; i >= 0; i--)
     {
      if(zz[i + 1] > 0 && zz[i + 1] != EMPTY_VALUE)
         extremum = zz[i + 1];
      if(i >= total)
         continue;
      target2[i] = extremum - close[i];
      target1[i] = (target2[i] >= 0 ? 1 : -1);
      macd_delta[i] = macd_main[i] - macd_signal[i];
     }
```

Здесь следует обратить внимание на тот момент, что на временном графике экстремум всегда должен быть после анализируемой цены закрытия. Поэтому цена закрытия берется от предыдущего бара по сравнению с последним проверенным значением индикатора ZigZag.

В том же цикле мы определим расстояние между основной и сигнальной линиями индикатора MACD и сохраним их в отдельный динамический массив macd_delta.

После расчета целевых показателей и расстояния между линиями индикатора MACD проведем нормализацию данных. Конечно, нормализацию мы будем проводить только в том случае, когда это требование указано пользователем в параметрах скрипта. Цель нормализации — преобразовать исходные данные таким образом, чтобы их значения были в диапазоне от −1 до 1 с центром в точке 0. Следует обратить внимание на ряд вводных, истекающих из особенностей самих индикаторов.

Индикатор RSI построен таким образом, что его значения нормализованы в диапазоне от 0 до 100. Следовательно, нам нет необходимости определять максимальное и минимальное значение данных этого индикатора для его нормализации. Поэтому алгоритм нормализации показаний данного индикатора ограничивается константой 50 — середина диапазона возможных значений индикатора. Формула нормализации значений будет следующей.

$$RSI_i^{Norm} = \frac{RSI_i - 50}{50}$$

Значения индикатора MACD не имеют верхней и нижней границы диапазона, но они центрированы относительно точки 0, так как по принципам построения индикатор отображает, находится ли цена выше или ниже скользящей средней. Аналогично можно сказать и о посчитанном нами расстоянии между основной и сигнальной линий индикатора. Сигнальная линия может быть как выше, так и ниже основной. Но в момент пересечения линий расстояние между ними равно 0. Поэтому для нормализации данных мы возьмем значение показателя и разделим на абсолютное значение максимального отклонения за анализируемый период.

$$MACD_i^{Norm} = \frac{MACD_i}{Max(Abs(MACD_{min}), MACD_{max})}$$

Здесь я хочу еще раз напомнить о сопоставимости данных для обучающей, тестовой выборки и данных промышленной эксплуатации. Если данные обучающей и тестовой выборки мы нормализуем сейчас, то для промышленной эксплуатации нам предстоит сохранить параметры нормализации всех трех показателей индикатора MACD.

После определения параметров нормализации организуем цикл по перебору и соответствующей корректировке исторических значений индикаторов.

Нормализуются только исходные данные, но не целевые значения.

```
//--- Нормализация данных
   if(NormalizeData)
     {
      double main_norm = MathMax(MathAbs(macd_main[ArrayMinimum(macd_main)]),
                                 macd_main[ArrayMaximum(macd_main)]);
      double sign_norm = MathMax(MathAbs(macd_signal[ArrayMinimum(macd_signal)]),
                                 macd_signal[ArrayMaximum(macd_signal)]);
      double delt_norm = MathMax(MathAbs(macd_delta[ArrayMinimum(macd_delta)]),
                                 macd_delta[ArrayMaximum(macd_delta)]);
      for(int i = 0; i < total; i++)
        {
         rsi[i] = (rsi[i] - 50.0) / 50.0;
         macd_main[i] /= main_norm;
         macd_signal[i] /= sign_norm;
         macd_delta[i] /= delt_norm;
        }
     }
```

Конечно, иногда может быть полезно нормализовать и целевые значения с той целью, чтобы подогнать их под значения определенной функции активации. Но в таком случае, как и при нормализации исходных данных, нужно сохранить параметры нормализации для дешифровки показаний нейронной сети в промышленной эксплуатации. Такие вопросы лежат на границе нейронной сети и основной программы, и решение их во многом зависит от поставленной задачи.

После подготовки генеральной совокупности нам предстоит разделить ее на обучающую и тестовую выборки. Общей практикой является «выдергивание» случайных записей из общей совокупности в тестовую выборку, а остаток идет в обучающую. Крайне не рекомендуется брать первые или последние подряд идущие паттерны для тестовой выборки. В первую очередь это связано с тем, что отдельно взятый небольшой срез данных больше подвержен влиянию локальных тенденций. Такая выборка не может быть репрезентативной для переноса оценки на большую совокупность. При извлечении же случайных записей из генеральной совокупности мы имеем гораздо большую вероятность извлечь для тестовой выборки максимально отличающиеся паттерны. Такая выборка будет максимально независимой от каких-либо локальных тенденций и более репрезентативной для оценки работы нейронной сети на глобальной совокупности данных. Однако следует сказать, что иногда все же берутся идущие подряд паттерны для тестовой выборки. Но это частные случаи, обусловленные особенностями архитектуры некоторых моделей.

Для разделения генеральной совокупности на обучающую и тестовую выборку мы создадим массив флагов test. Данный массив будет иметь размер нашей глобальной совокупности данных. Значение элементов будет указывать на направление использования паттерна:

- 0 — обучающая выборка
- 1 — тестовая выборка

Для бинарной классификации можно использовать и массив логических значений. Но когда нам понадобится добавить валидационную выборку, мы легко можем для нее использовать значение 2, а использование массива логических значений не дает нам такую возможность.

Наш массив флагов изначально будет инициализирован нулевыми значениями. Иными словами, мы определяем, что по умолчанию паттерн относится к обучающей выборке. Затем мы определяем количество паттернов для тестовой совокупности. И потом организовываем цикл по числу элементов для тестовой выборки с генерацией случайных значений внутри этого цикла. Генератор случайных значений должен возвращать целочисленное число в диапазоне от 0 до размера генеральной совокупности. В своем решении я воспользовался встроенной функцией *MQL5 MathRand* для генерации псевдослучайных чисел. Данная функция возвращает целочисленное значение в диапазоне от 0 до 32767. Но так как размер генеральной совокупности ожидается более 33 тыс. элементов я перемножил два случайных числа. Такой вариант способен генерировать более 1 млрд случайных значений. Чтобы привести полученное случайное число к размеру нашей генеральной совокупности мы сначала разделим полученное случайное число на квадрат от 32767, тем самым нормализуем случайное число в диапазоне от 0 до 1. Затем умножим на число элементов в нашей генеральной совокупности. Полученное число нам укажет на порядковый номер паттерна для тестовой выборки.

Нам остается только записать 1 в соответствующий элемент массива флагов. Но остается вероятность попадания дважды (а может и более раз) в один и тот же элемент массива флагов. Если мы не проконтролируем это, то с большой долей вероятности получим тестовую выборку меньше ожидаемой. Поэтому перед записью 1 в выбранный элемент массива флагов мы сначала проверяем его текущее состояние. И если в нем уже содержится 1, то мы уменьшаем счетчик итераций цикла на 1 и переходим к генерации следующего случайного числа. Таким образом, при попадании в один и тот же элемент счетчик итераций цикла не будет считаться, и мы получим на выходе тестовую выборку ожидаемого размера.

```
//--- Генерируем случайным образом индексы данных для тестовой выборки
   ArrayInitialize(test, 0);
   int for_test = (int)((total - BarsToLine) * 0.2);
   for(int i = 0; i < for_test; i++)
     {
      int t = (int)((double)(MathRand() * MathRand()) / MathPow(32767.0, 2) *
                    (total - 1 - BarsToLine)) + BarsToLine;
      if(test[t] == 1)
        {
         i--;
         continue;
        }
      test[t] = 1;
     }
```

На этом подготовительную работу можно считать завершенной. Остается только сохранить подготовленные данные в соответствующие файлы. Для записи данных мы открываем два файла для записи в соответствии с именами, указанными в параметрах скрипта. Логично было бы

создать бинарные файлы для записи числовых данных. Они занимают меньше места на диске, и работа с ними осуществляется быстрее. Но так как мы предполагаем загрузку данных из приложений написанных на других языках программирования, в частности из скриптов на *Python*, наиболее универсальным подходом будет использование *CSV*-файлов.

Мы открываем для записи два *CSV*-файла и сразу проверяем полученные хендлы для обращения к фалам. Ошибочные хендлы будут сигнализировать об ошибке открытия файла. Об этом выведем соответствующее сообщение в журнал терминала.

```
//--- Открываем для записи файл обучающей выборки
   int Study = FileOpen(StudyFileName, FILE_WRITE |
                                       FILE_CSV |
                                       FILE_ANSI, ",", CP_UTF8);
   if(Study == INVALID_HANDLE)
     {
      PrintFormat("Error opening file %s: %d", StudyFileName, GetLastError());
      return;
     }
//--- Открываем для записи файл тестовой выборки
   int Test = FileOpen(TestFileName, FILE_WRITE |
                                     FILE_CSV |
                                     FILE_ANSI, ",", CP_UTF8);
   if(Test == INVALID_HANDLE)
     {
      PrintFormat("Error opening file %s: %d", TestFileName, GetLastError());
      return;
     }
```

После успешного открытия фалов организовываем цикл с перебором всех элементов генеральной совокупности. Обратите внимание, что цикл начинается не с нулевого элемента, а с элемента, соответствующего количеству баров на паттерн. Ведь для полной записи паттерна мы должны указать данные нескольких предыдущих свечей. Разделение на обучающую и тестовую выборку будем осуществлять на этапе записи в файл. Проверяя значение соответствующего элемента массива флагов, будем заменять хендл файла для записи паттерна хендлом файла правильной выборки. Непосредственно запись паттерна в файл вынесена в отдельную функцию, которую мы посмотрим чуть позже. Для отслеживания процесса выведем в комментарии на графике процент выполнения записи в файл.

По завершении цикла очистим комментарии на графике, закроем файлы и выведем в журнал информацию об именах и пути записи файлов.

```
//--- Запись выборок в файлы
   for(int i = BarsToLine - 1; i < total; i++)
     {
      Comment(StringFormat("%.2f%%", i * 100.0 / (double)(total - BarsToLine)));
      if(!WriteData(target1, target2, rsi, macd_main, macd_signal, macd_delta, i,
                                     BarsToLine, (test[i] == 1 ? Test : Study)))
        {
         PrintFormat("Error to write data: %d", GetLastError());
         break;
        }
     }
//--- Закрываем файлы
   Comment("");
   FileFlush(Study);
   FileClose(Study);
   FileFlush(Test);
   FileClose(Test);
   PrintFormat("Study data saved to file %s\\MQL5\\Files\\%s",
               TerminalInfoString(TERMINAL_DATA_PATH), StudyFileName);
   PrintFormat("Test data saved to file %s\\MQL5\\Files\\%s",
               TerminalInfoString(TERMINAL_DATA_PATH), TestFileName);
  }
```

Для записи информации о паттерне в файл создадим функцию *WriteData*. В параметрах функции передадим указатели на массивы исходных и целевых данных, порядковый номер последнего бара паттерна в массивах данных, количество баров для анализа одного паттерна и хендл файла для записи данных. Указание последнего бара паттерна вместо первого сделано в попытке приблизить построение паттерна к реальным условиям эксплуатации нейронной сети. Работая с таймсериями биржевых котировок в реальном времени, мы в текущем моменте всегда находимся на последнем известном баре. Мы анализируем информацию с нескольких последних баров, которые уже являются историей, и пытаемся понять наиболее вероятное предстоящее ценовое движение. Так и здесь, указанный в параметрах бар для нас является «текущим моментом». Мы берем перед ним указанное количество баров, и все это составляет для нас анализируемый паттерн. На основе него наша нейронная сеть должна определить вероятное ценовое движение и его силу.

```
//+------------------------------------------------------------------+
//| Функция записи паттерна в файл                                   |
//+------------------------------------------------------------------+
bool WriteData(double &target1[], // Буфер 1 целевых значений
               double &target2[], // Буфер 2 целевых значений
               double &data1[],   // Буфер 1 исторических данных
               double &data2[],   // Буфер 2 исторических данных
               double &data3[],   // Буфер 2 исторических данных
               double &data4[],   // Буфер 2 исторических данных
               int cur_bar,       // Текущий бар окончания паттерна
               int bars,          // Количество исторических баров
                                  // в одном паттерне
               int handle)        // Хендл файла для записи
  {
```

Информацию по паттерну сначала соберем в строковую переменную типа *string*. При этом не забываем между значениями элементов вставлять разделительный знак. Разделительный знак должен соответствовать разделительному знаку, указанному при открытии *CSV*-файла. Сбор данных в строковую переменную — вынужденный компромисс. Дело в том, что функция записи в текстовый файл *FileWrite* имеет ограничение в 63 параметра для записи, а запись каждого вызова завершается символом конца строки. Итак, пред нами возникли две проблемы:

1. Указывая все данные о паттерне в рамках одного вызова функции *WriteData*, при использовании 4 показателей на 1 бар мы сможем описать не более 15 свечей.
2. Мы должны собрать информацию обо всех барах одномоментно.

Мы не можем использовать цикл для перебора значений массива. Нужно руками прописать все записываемые элементы в параметрах функции записи данных в файл. Использование строковой переменной помогает решить указанные вопросы. Мы можем в простом цикле собрать все значения в одну текстовую строку. При этом у нас нет ограничения по количеству включаемых параметров. Конечно, во время сбора показателей в строку нам предстоит вставить между ними разделитель, тем самым имитируя строку файла *CSV*. Да и записывать в файл мы будем уже собранную строку один раз. Следовательно, функция вставит символ конца строки в конце записи один раз. Таким образом, весь паттерн в нашем файле будет записан в одну строку.

```
   //--- Проверка хендла файла
      if(handle == INVALID_HANDLE)
        {
         Print("Invalid Handle");
         return false;
        }
   //--- Определяем индекс первой записи исторических данных паттерна
      int start = cur_bar - bars + 1;
      if(start < 0)
        {
         Print("Too small current bar");
         return false;
        }

   //--- Проверяем корректность индекса данных и записываемых в файл
      int size1 = ArraySize(data1);
      int size2 = ArraySize(data2);
      int size3 = ArraySize(data3);
      int size4 = ArraySize(data4);
      int sizet1 = ArraySize(target1);
      int sizet2 = ArraySize(target2);
      string pattern = (string)(start < size1 ? data1[start] : 0.0) + "," +
                       (string)(start < size2 ? data2[start] : 0.0) + "," +
                       (string)(start < size3 ? data3[start] : 0.0) + "," +
                       (string)(start < size4 ? data4[start] : 0.0);
      for(int i = start + 1; i <= cur_bar; i++)
        {
         pattern = pattern + "," + (string)(i < size1 ? data1[i] : 0.0) + "," +
                                   (string)(i < size2 ? data2[i] : 0.0) + "," +
                                   (string)(i < size3 ? data3[i] : 0.0) + "," +
                                   (string)(i < size4 ? data4[i] : 0.0);
        }
      return (FileWrite(handle, pattern,
                       (double)(cur_bar < sizet1 ? target1[cur_bar] : 0),
                       (double)(cur_bar < sizet2 ? target2[cur_bar] : 0)) > 0);
    }
```

В итоге мы получаем структурированный *CSV*-файл, в котором между каждыми двумя соседними элементами строит разделительный знак, а каждая строка представляет собой отдельный паттерн для анализа данных.

Следует также добавить, что для предотвращения ошибки выхода за пределы массива перед каждым обращением к массивам данных мы проверяем значения индекса на соответствие размерам массива. В случае некорректного индекса вместо показателя записывается 0. Напомню, что в процессе работы алгоритма нейрона все значения входящего вектора показателей умножаются на весовые коэффициенты, а полученные произведения складываются в общую сумму. Умножение любого весового коэффициента на 0 всегда возвращает 0. Поэтому показатели с нулевым значением не оказывают прямого влияния на результат работы нейрона. Конечно, здесь можно говорить о косвенном влиянии. Ведь возможна ситуация, когда именно вклада данного показателя в общую сумму не хватает для активации нейрона. Но это меньшее из зол, и мы принимаем эти риски.

Наверное, надо сказать, что для будущих тестов наших моделей мы сразу создадим два набора обучающих данных:

- обучающие выборки с ***ненормированными*** исходными данными мы запишем в файлы *study_data_not_norm.csv* и *test_data_not_norm.csv*;
- обучающие выборки с ***нормированными*** исходными данными мы запишем в файлы *study_data.csv* и *test_data.csv*.

Для создания указанных выборок данных для бучения мы воспользуемся вышеописанным скриптом из файла create_initial_data.mq5. Его мы запустим два раза для сбора одних и тех же исторических данных, но при этом изменим названия файлов для записи данных и «Флаг нормализации данных».

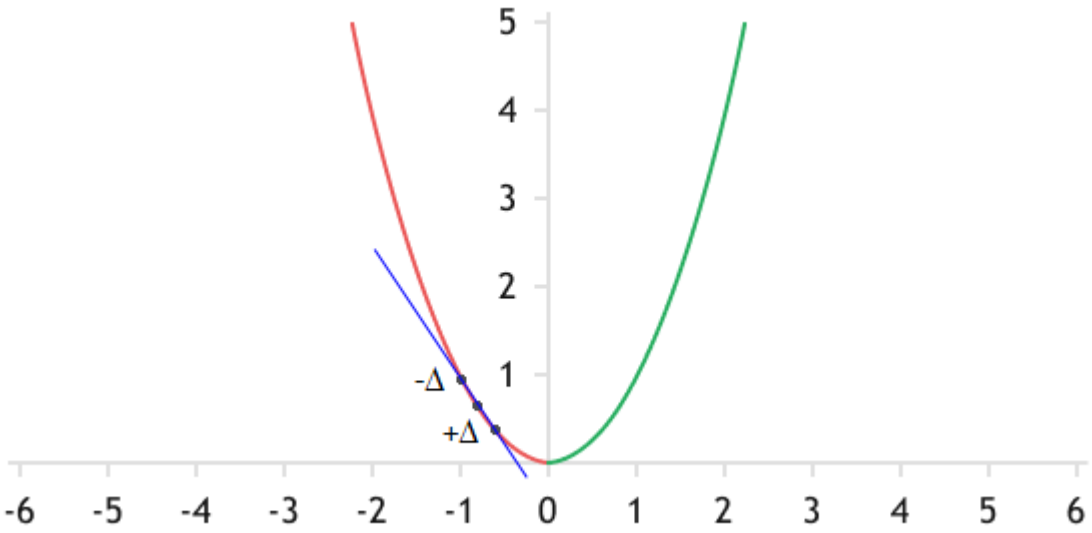
3.10 Проверка корректности распределения градиента

Вот наступил то момент, когда мы с вами соберем первую нейронную сеть средствами *MQL5*. Но не буду вас сильно обнадеживать — первая наша нейронная сеть не будет ничего анализировать и прогнозировать. Она выполнит контрольную функцию и проверит корректность ранее проделанной работы. Дело в том, что перед тем как приступить непосредственно к обучению нейронной сети, нам с вами необходимо проверить корректность распределения градиента ошибки по нейронной сети. Думаю, не вызывает вопросов тот момент, что от корректности реализации указанного процесса очень сильно зависит и результат всего обучения нейронной сети. Ведь именно градиент ошибки на каждом весовом коэффициенте определяет величину и направление изменения последнего.

Для проверки корректности распределения градиента воспользуемся тем фактом, что для определения производной функции у нас есть два варианта:

- *Аналитический* — определение градиента функции на основе ее производной первого порядка. Этот метод и реализован в нашем методе обратного прохода.
- *Эмпирический* — при прочих равных условиях изменяется значение одного показателя и оценивается его влияние на конечный результат функции.

Напомню геометрический смысл градиента — это наклон касательной к графику функции в текущей точке. Он показывает, как изменится значение функции при изменении значения параметра.



Геометрический смысл градиента — это наклон касательной к графику функции в текущей точке

Для построения линии нам необходимо две точки. Следовательно, за две несложные итерации мы можем найти эти точки на графике функции. Нам нужно сначала прибавить небольшое число к текущему значению параметра и посчитать значение функции без изменения прочих параметров. Это будет первая точка. Повторим итерацию, только теперь отнимем от текущего значения то же самое число и получим вторую точку. Прямая через эти две точки будет с некоторой долей погрешности приближаться к искомой касательной. И чем меньше будет число, используемое для изменения параметра, тем меньше будет эта погрешность. На этом и строится *эмпирический* способ определения градиента.

Если этот способ настолько прост, то почему его не использовать на постоянной основе? Здесь все довольно просто. За простотой метода кроется большое количество операций:

1. Осуществить прямой проход и сохранить его результат.
2. Немного увеличить один параметр и повторить прямой проход с сохранением результата.
3. Немного уменьшить один параметр и повторить прямой проход с сохранением результата.
4. По найденным точкам построить прямую и определить ее наклон.

И это все для определения только градиента на одном шаге для одного параметра. А представьте, сколько нам потребуется времени и вычислительных ресурсов, если мы воспользуемся этим методом при обучении нейронной сети хотя бы с сотней параметров. И не забывайте, что современные нейронные сети содержат гораздо больше параметров. К примеру, такой гигант как GPT-3 содержит 175 млрд параметров. Конечно, мы с вами не будем строить таких гигантов на домашнем компьютере. Но использование аналитического метода сильно сокращает количество необходимых итераций и времени на их выполнение.

В то же время мы можем собрать небольшую нейронную сеть и сравнить на ней результаты двух методов. Их близость укажет на корректность работы прописанного нами алгоритма аналитического метода. Наличие значительных расхождений в результатах вычислений двух методов укажет на необходимость перепроверить заложенный ранее в методе обратного прохода алгоритм аналитического метода.

Для практической реализации данной идеи создадим скрипт *check_gradient_percp.mq5*. Данный скрипт получит три внешних параметра:

- размер вектора исходных данных,
- флаг использования технологии *OpenCL*,
- функцию активации скрытого слоя.

Обратите внимание, что мы ничего не указали об источнике исходных данных. Дело в том, что для данной работы нам абсолютно неважно, какие данные будут поданы на вход модели. Мы лишь проверяем корректность работы методов обратного прохода. Поэтому в качестве исходных данных мы можем использовать вектор случайных значений.

```
//+------------------------------------------------------------------+
//| Внешние параметры для работы скрипта                             |
//+------------------------------------------------------------------+
// Размер вектора исходных данных
input int      BarsToLine    = 40;
// Использовать OpenCL
input bool     UseOpenCL     =  true;
// Функция активации скрытого слоя
input ENUM_ACTIVATION_FUNCTION HiddenActivation = AF_SWISH;
```

Кроме того, в глобальной области скрипта подключим нашу библиотеку и объявим объект нейронной сети.

```
//+------------------------------------------------------------------+
//| Подключаем библиотеку нейронной сети                             |
//+------------------------------------------------------------------+
#include "..\..\..\Include\NeuroNetworksBook\realization\neuronnet.mqh"
CNet Net;
```

В начале тела самого скрипта зададим архитектуру небольшой нейронной сети. Но так как аналогичную работу нам потребуется провести еще не раз при проверке корректности организации процесса в других архитектурных решениях, непосредственно создание модели мы вынесем в отдельную процедуру *CreateNet*. В параметрах данная процедура получает указатель на объект создаваемой модели нейронной сети.

Напомню, для создания модели нейронной сети ранее мы создали метод *CNet::Create*. В параметрах данный метод принимает динамический массив описания архитектуры нейронной сети. Следовательно, нам необходимо организовать подобное описание новой модели. Описание каждого нейронного слоя соберем в отдельный экземпляр класса *CLayerDescription*. Объединим их в динамический массив *CArrayObj*. При добавлении описания нейронов в динамический массив следим, чтобы их последовательность строго соответствовала расположению нейронных слоев в нейронной сети. В своей практике я просто последовательно создаю описания слоев в порядке их расположения в нейронной сети и добавляю их в массив по мере создания.

```
bool CreateNet(CNet &net)
  {
   CArrayObj *layers = new CArrayObj();
   if(!layers)
     {
      PrintFormat("Error creating CArrayObj: %d", GetLastError());
      return false;
     }
```

Для проверки корректности работы реализованного алгоритма обратного распространения ошибки создадим трехслойную нейронную сеть. Все слои будут построены на базе созданного

нами класса *CNeuronBase*. Размер первого нейронного слоя исходных данных пользователь указал во внешнем параметре *BarsToLine*. Его мы создадим без функции активации и метода обновления весовых коэффициентов. В принципе это базовый подход к созданию слоя исходных данных.

```
//--- слой исходных данных
   CLayerDescription *descr = new CLayerDescription();
   if(!descr)
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      delete layers;
      return false;
     }
   descr.type = defNeuronBase;
   descr.count = BarsToLine;
   descr.window = 0;
   descr.activation = AF_NONE;
   descr.optimization = None;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete layers;
      delete descr;
      return false;
     }
```

Количество нейронов во втором (скрытом) нейронном слое мы зададим в 10 раз больше слоя исходного данного. Однако непосредственно размер нейронного слоя не влияет на процесс анализа работы алгоритма. Этот слой уже получит функцию активации, которую пользователь укажет во внешнем параметре скрипта *HiddenActivation*. Для примера я взял Swish. Вам же я рекомендовал бы поэкспериментировать со всеми используемыми функциями активации, ведь на данном этапе мы хотим проверить корректность работы всех ранее написанных методов. Поэтому чем более разносторонним будет тестирование, тем больше мы сможем закрыть спорных вопросов на данном этапе и не отвлекаться на их поиск и устранение в процессе обучения нейронной сети.

На данном этапе мы не будем проводить обновление весовых коэффициентов. Поэтому указанный метод обновления весовых коэффициентов никак не повлияет на результаты нашего тестирования.

```
//--- скрытый слой
   descr = new CLayerDescription();
   if(!descr)
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      delete layers;
      return false;
     }
   descr.type = defNeuronBase;
   descr.count = 10 * BarsToLine;
   descr.activation = HiddenActivation;
   descr.optimization = Adam;
   descr.activation_params[0] = (TYPE)1;
   descr.activation_params[1] = (TYPE)0;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete layers;
      delete descr;
      return false;
     }
```

Третий нейронный слой будет содержать только один нейрон для вывода результатов и линейную функцию активации.

```
//--- слой результатов
   descr = new CLayerDescription();
   if(!descr)
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      delete layers;
      return false;
     }
   descr.type = defNeuronBase;
   descr.count = 1;
   descr.activation = AF_LINEAR;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
   descr.activation_params[1] = 0;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete layers;
      delete descr;
      return false;
     }
```

Собрав полное описание нейронной сети в едином динамическом массиве, мы генерируем нейронную сеть. Для этого вызываем метод *CNet::Create* нашего базового класса нейронной сети, в котором осуществляется генерация нейронной сети в соответствии с переданным описанием. На каждом шаге проверяем корректность выполнения операций по возвращаемым

результатам. Получение логического значения *true* соответствует корректному выполнению операций метода. При возникновении любой из ошибок метод вернет значение *false*.

Укажем флаг использования технологии *OpenCL*. Для полного тестирования нам предстоит проверить корректность работы метода обратного распространения ошибки в обоих режимах работы нейронной сети.

```
//--- инициализируем нейронную сеть
   if(!net.Create(layers, (TYPE)3.0e-4, (TYPE)0.9, (TYPE)0.999, LOSS_MAE, 0, 0))
     {
      PrintFormat("Error of init Net: %d", GetLastError());
      delete layers;
      return false;
     }
   delete layers;
   net.UseOpenCL(UseOpenCL);
   PrintFormat("Use OpenCL %s", (string)net.UseOpenCL());
//---
   return true;
  }
```

Мы завершаем работу с процедурой создания модели и переходим к основной процедуре нашего скрипта *OnStart*. В ней для создания модели нейронной сети нам достаточно вызвать вышеописанную процедуру.

```
void OnStart()
  {
//--- создание модели
   if(!CreateNet(Net))
      return;
```

На этом этапе объект нейронной сети готов к проведению тестирования. Но нам еще нужны исходные данные для тестирования. Как уже было сказано выше, мы просто заполним их случайными значениями. Создадим буфер данных *CBufferType* для хранения последовательности исходных данных. Целевые результаты нам на данном этапе не интересны. При генерации нейронной сети мы заполнили матрицу весов случайными значениями и не ожидаем попадания в целевые значения. Обучение нейронной сети мы тоже не планируем. Поэтому мы не будем расходовать ресурсы на загрузку лишней информации.

```
//--- создаем буфер для считывания исходных данных
   CBufferType *pattern = new CBufferType();
   if(!pattern)
     {
      PrintFormat("Error creating Pattern data array: %d", GetLastError());
      return;
     }
```

В цикле заполним весь буфер случайными значениями.

```
//--- генерируем случайные исходные данные
   if(!pattern.BufferInit(1, BarsToLine))
      return;
   for(int i = 0; i < BarsToLine; i++)
      pattern.m_mMatrix[0, i] = (TYPE)MathRand() / (TYPE)32767;
```

Теперь информации достаточно для проведения прямого прохода нейронной сети. Его мы и осуществим путем вызова метода *FeedForward* нашей нейронной сети. Результаты прямого прохода сохраним в отдельном буфере данных эталонных значений. Наверно странно звучит название «*эталонный*» для случайно полученного результата. Но в рамках нашего тестирования это будет тот эталон, отклонение от которого мы будем считать при изменении исходных данных или весовых коэффициентов.

```
//--- делаем прямой и обратный проход для получения аналитических градиентов
   const TYPE delta = (TYPE)1.0e-5;
   TYPE dd = 0;
   CBufferType *init_pattern = new CBufferType();
   init_pattern.m_mMatrix.Copy(pattern.m_mMatrix);
   if(!Net.FeedForward(pattern))
     {
      PrintFormat("Error in FeedForward: %d", GetLastError());
      return;
     }
   CBufferType *etalon_result = new CBufferType();
   if(!Net.GetResults(etalon_result))
     {
      PrintFormat("Error in GetResult: %d", GetLastError());
      return;
     }
```

На следующем этапе прибавим к результату прямого прохода небольшую константу и запустим обратный проход нашей нейронной сети для расчета градиентов ошибки аналитическим путем. В приведенном примере в качестве отклонения я использовал константу $1*10^{-5}$.

```
//--- создаем буфер результатов
   CBufferType *target = new CBufferType();
   if(!target)
     {
      PrintFormat("Error creating Pattern Target array: %d", GetLastError());
      return;
     }
//--- сохраняем в отдельные буферы полученные данные
   target.m_mMatrix.Copy(etalon_result.m_mMatrix);
   target.m_mMatrix[0, 0] = etalon_result.m_mMatrix[0, 0] + delta;
   if(!Net.Backpropagation(target))
     {
      PrintFormat("Error in Backpropagation: %d", GetLastError());
      delete target;
      delete etalon_result;
      delete pattern;
      delete init_pattern;
      return;
     }
   CBufferType *input_gradient = Net.GetGradient(0);
   CBufferType *weights = Net.GetWeights(1);
   CBufferType *weights_gradient = Net.GetDeltaWeights(1);
   if(UseOpenCL)
     {
      input_gradient.BufferRead();
      weights.BufferRead();
      weights_gradient.BufferRead();
     }
```

Обратите внимание, что нам необходимо сохранить неизменным эталонный результат. Поэтому нам понадобилось создать еще один объект буфера данных, в который мы скопировали значения из буфера эталонных значений. Уже в нем мы корректируем данные для обратного прохода.

По результатам обратного прохода мы сохраним полученные аналитическим путем градиенты ошибок на уровне исходных данных и весовых коэффициентов. Также сохраним сами весовые коэффициенты, которые нам понадобятся при анализе распределения градиентов ошибки на уровне матрицы весов.

Наверно стоит сказать, что так как в процессе обучения мы корректируем весовые коэффициенты, то наиболее информативным для нас будет получение корректных градиентов на уровне матрицы весов. Корректные градиенты на уровне исходных данных больше служат косвенным свидетельством корректности распределения градиента по всей нейронной сети. Это связано с тем, что прежде, чем определить градиент ошибки на уровне исходных данных, нам предстоит последовательно провести его аналитическим методом через все слои нашей нейронной сети.

Мы получили градиенты ошибок аналитическим методом. Далее нам предстоит определить градиенты эмпирическим путем и сравнить их с результатами аналитического метода.

Сначала посмотрим на уровне исходных данных. Для этого наш паттерн исходных данных скопируем в новый динамический массив, в котором мы сможем изменять нужные нам показатели без страха потерять исходный паттерн.

Организуем цикл по перебору всех показателей нашего паттерна. Внутри цикла мы поочередно к каждому показателю исходного паттерна сначала прибавим нашу константу $1*10^{-5}$ и осуществим прямой проход нейронной сети. После прямого прохода возьмем полученный результат и сравним его с эталонным, который сохранили ранее. Разницу между результатами прямого прохода сохраним в отдельную переменную. Затем от исходного значения того же показателя отнимем константу и повторно осуществим прямой проход. Результат прямого прохода также сравним с эталонным результатом. Найдем среднее арифметическое двух проходов.

```
//--- в цикле поочередно изменяем элементы исходных данных и сравниваем
//--- эмпирический результат со значением аналитического метода
   for(int k = 0; k < BarsToLine; k++)
     {
      pattern.m_mMatrix.Copy(init_pattern.m_mMatrix);
      pattern.m_mMatrix[0, k] = init_pattern.m_mMatrix[0, k] + delta;
      if(!Net.FeedForward(pattern))
        {
         PrintFormat("Error in FeedForward: %d", GetLastError());
         return;
        }
      if(!Net.GetResults(target))
        {
         PrintFormat("Error in GetResult: %d", GetLastError());
         return;
        }
      TYPE d = target.At(0) - etalon_result.At(0);

      pattern.m_mMatrix[0, k] = init_pattern.m_mMatrix[0, k] - delta;
      if(!Net.FeedForward(pattern))
        {
         PrintFormat("Error in FeedForward: %d", GetLastError());
         return;
        }
      if(!Net.GetResults(target))
        {
         PrintFormat("Error in GetResult: %d", GetLastError());
         return;
        }
      d -= target.At(0) - etalon_result.At(0);
      d /= 2;
      dd += input_gradient.At(k) - d;
     }
   delete pattern;
```

В этом моменте следует быть внимательным со знаком операции и отклонений. В первом случае мы прибавляли константу и получили некое отклонение результатов. Отклонение считаем как текущее значение минус эталонное.

$$\Delta = R_{Current} - R_{Reference}$$

Во втором случае мы отнимали константу от первоначального значения. Соответственно при той же формуле расчета отклонения получим значение с противоположным знаком. Следовательно, чтобы объединить полученные результаты по модулю и сохранить правильное направление градиента, нам нужно из первого отклонения отнять второе.

Для получения среднего отклонения разделим полученный результат на 2. Полученный результат сравним с результатом аналитического способа.

Описанные выше операции повторим для всех параметров исходного паттерна.

Есть еще один аспект, который нам следует учесть. Градиент указывает изменение значения функции при изменении показателя на 1. Наша же константа намного меньше. Следовательно, полученный нами эмпирический градиент сильно занижен. Для компенсации этого разделим полученное эмпирическим путем значение на нашу константу и выведем суммарный результат в журнал *MetaTrader 5*.

```
//--- выводим в журнал суммарное значение отклонений на уровне исходных данных
   PrintFormat("Delta at input gradient between methods %.5e", dd / delta);
```

Аналогичным способом определяем эмпирический градиент на уровне матрицы весов. Обратите внимание, что для доступа к матрице весовых коэффициентов мы получаем не копию матрицы, а указатель на объект. Это очень важный момент. Именно благодаря этому мы можем изменять значения весовых коэффициентов непосредственно в нашем скрипте без создания дополнительных методов обновления значений буфера в классах нейронной сети и нейронного слоя. В то же время такой подход может быть применен больше как исключение из правил, нежели в качестве основного подхода. Дело в том, что при таком подходе мы не можем отследить изменение матрицы весов из объекта нейронного слоя.

Суммарный результат сравнения эмпирических и аналитических градиентов на уровне матрицы весов также выведем в журнал *MetaTrader 5*.

```
//--- обнуляем значение суммы и повторяем цикл для градиентов весовых коэффициентов
   dd = 0;
   CBufferType *initial_weights = new CBufferType();
   if(!initial_weights)
     {
      PrintFormat("Error creating reference weights buffer: %d", GetLastError());
      return;
     }
   if(!initial_weights.m_mMatrix.Copy(weights.m_mMatrix))
     {
      PrintFormat("Error copying weights to initial weights buffer: %d",
                                                              GetLastError());
      return;
     }

   for(uint k = 0; k < weights.Total(); k++)
     {
      if(k > 0)
         weights.Update(k - 1, initial_weights.At(k - 1));
      weights.Update(k, initial_weights.At(k) + delta);
      if(UseOpenCL)
         if(!weights.BufferWrite())
            return;
      if(!Net.FeedForward(init_pattern))
        {
         PrintFormat("Error in FeedForward: %d", GetLastError());
         return;
        }
      if(!Net.GetResults(target))
        {
         PrintFormat("Error in GetResult: %d", GetLastError());
         return;
        }
      TYPE d = target.At(0) - etalon_result.At(0);
```

```
      weights.Update(k, initial_weights.At(k) - delta);
      if(UseOpenCL)
         if(!weights.BufferWrite())
            return;
      if(!Net.FeedForward(init_pattern))
        {
         PrintFormat("Error in FeedForward: %d", GetLastError());
         return;
        }
      if(!Net.GetResults(target))
        {
         PrintFormat("Error in GetResult: %d", GetLastError());
         return;
        }
      d -= target.At(0) - etalon_result.At(0);
      d /= 2;
      dd += weights_gradient.At(k) - d;
     }
  //--- выводим в журнал суммарное значение отклонений на уровне весовых коэффициентов
    PrintFormat("Delta at weights gradient between methods %.5e", dd / delta);
```

После выполнения всех итераций очистим память, удалив используемые объекты, и выйдем из программы.

```
 //--- перед выходом из скрипта очищаем память
    delete init_pattern;
    delete etalon_result;
    delete initial_weights;
    delete target;
  }
```

Результаты моего тестирования приведены на скриншоте ниже. По результатам тестирования я получил отклонение в 11–12-м знаке после запятой. Для сравнения в разных источниках считаются приемлемыми отклонения в 8–9-м знаке после запятой. И не стоит обращать внимание, что при использовании OpenCL отклонение получилось на порядок меньше. Это не преимущество использования технологии, а скорее влияние фактора случайности. При каждом запуске заново генерировалась случайная матрица весовых коэффициентов и исходных данных. В результате этого сравнение проводилось на разных участках функции нейронной сети с различной кривизной.

```
OpenCL: CPU device '11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz' selected
Use OpenCL false
Delta at input gradient between methods -2.09701e-11
Delta at weights gradient between methods 5.76474e-11
OpenCL: CPU device '11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz' selected
Use OpenCL true
Delta at input gradient between methods -5.97421e-12
Delta at weights gradient between methods -2.70610e-12
```

Результаты сравнения аналитических и эмпирических градиентов ошибки

В целом можно сказать, что тестирование подтвердило корректность нашей реализации алгоритма обратного распространения ошибки как средствами *MQL5*, так и с использованием технологии многопоточных вычислений *OpenCL*. Следующим нашим шагом будет сборка более сложного перцептрона, и мы попробуем его обучить на обучающей выборке.

3.11 Сравнительное тестирование реализаций

В предыдущем разделе мы проверили корректность работы алгоритма обратного распространения ошибки. Теперь можно смело переходить к обучению нашего перцептрона. Данную работу мы проведем в скрипте *perceptron_test.mq5*. Первый блок скрипта напомнит вам скрипт из предыдущего раздела. Это следствие использования нашей библиотеки для создания нейронных сетей. Создавать нейронную сеть мы будем с ее помощью. Следовательно, алгоритм инициализации и использования нейронной сети будет во всех случаях идентичный.

Для возможности проведения различных вариантов тестирования в скрипт добавим нижеследующие внешние параметры:

- Имя файла, содержащего обучающую выборку.
- Имя файла для записи динамики изменения ошибки. По данным значениям мы сможем построить график изменения ошибки в процессе обучения, что поможет нам визуализировать процесс обучения нейронной сети.
- Количество исторических баров, используемых в описании одного паттерна.
- Количество нейронов входного слоя на один бар.
- Флаг-переключатель использования технологии *OpenCL* в процессе обучения нейронной сети.
- Размер пакета для одной итерации обновления матрицы весов.
- Обучающий коэффициент.
- Количество скрытых слоев.
- Количество нейронов в одном скрытом слое.
- Количество итераций обновления матрицы весов.

Как и в предыдущем разделе, после объявления внешних параметров в глобальной области скрипта подключим нашу библиотеку для создания нейронной сети и объявим объект базового класса *CNet*.

```
//+------------------------------------------------------------------+
//| Внешние параметры для работы скрипта                             |
//+------------------------------------------------------------------+
// Имя файла с обучающей выборкой
input string   StudyFileName  = "study_data.csv";
// Имя файла для записи динамики ошибки
input string   OutputFileName = "loss_study.csv";
// Количество исторических баров в одном паттерне
input int      BarsToLine     = 40;
// Количество нейронов входного слоя на 1 бар
input int      NeuronsToBar   = 4;
// Использовать OpenCL
input bool     UseOpenCL      = false;
// Размер пакета для обновления матрицы весов
input int      BatchSize      = 10000;
// Коэффициент обучения
input double   LearningRate   = 3e-5;
// Количество скрытых слоев
input int      HiddenLayers   = 1;
// Количество нейронов в одном скрытом слое
input int      HiddenLayer    = 40;
// Количество итераций обновления матрицы весов
input int      Epochs         = 1000;
//+------------------------------------------------------------------+
//| Подключаем библиотеку нейронной сети                             |
//+------------------------------------------------------------------+
#include <NeuroNetworksBook\realization\neuronnet.mqh>
CNet *net;
```

Перед тем как перейти к написанию кода скрипта, давайте подумаем, каким функционалом нам нужно его наполнить.

Вначале нам предстоит создать модель. Для этого мы создадим описание архитектуры модели и вызовем метод инициализации модели. Аналогичные операции мы выполняли в скрипте проверки корректности распределения градиента ошибки.

Затем, чтобы обучить нашу модель, нам необходимо загрузить ранее созданную обучающую выборку, которая будет содержать набор исходных данных и целевых значений.

Лишь после успешного выполнения вышеуказанных операций мы можем запустить процесс обучения модели. Это повторяющийся циклический процесс. Он включает в себя прямой проход, обратный проход и обновление матрицы весов. Есть несколько подходов к продолжительности обучения моделей. Наиболее часто используется ограничение по количеству эпох обучения и контроль изменения ошибки модели. Мы будем использовать первый вариант. Анализ динамики ошибки в процессе обучения модели позволит нам выработать критерии для применения второго метода. Поэтому в процессе обучения нам нужно записать изменение ошибки модели и сохранить собранную последовательность после обучения модели.

Ну и конечно, обучение модели не должно быть «пустой» работой. В конце обучения мы сохраним полученную модель.

Таким образом, мы определили необходимый функционал нашего скрипта. С целью создать понятный и читаемый код, мы разделим его на блоки, аналогичные вышеуказанным задачам. В

теле основной функции *OnStart* лишь будем осуществлять последовательный вызов соответствующих функций с контролем выполнения операций.

Вначале мы создадим вектор для записи динамики ошибки модели в процессе обучения. Его размер будет равен количеству эпох обучения.

```
//+------------------------------------------------------------------+
//| Начало программы скрипта                                         |
//+------------------------------------------------------------------+
void OnStart()
  {
//--- подготовим вектор для хранения истории ошибок сети
   VECTOR loss_history = VECTOR::Zeros(Epochs);
```

Затем инициализируем нашу модель для обучения. Здесь мы объявляем экземпляр класса нейронной сети и передадим указатель на объект в функцию инициализации модели. Обязательно проверяем результат выполнения операции.

```
//--- 1. инициализация модели
   CNet net;
   if(!NetworkInitialize(net))
      return;
```

Следующим шагом загружаем обучающую выборку. Для этого нам потребуется два динамических массива: один из них для загрузки паттернов исходных данных, второй — для целевых значений. Оба массива будут синхронизированы.

Непосредственно загрузка данных осуществляется в функции *LoadTrainingData*, в параметрах которой мы передадим файл для загрузки данных и указатели на созданные объекты динамических массивов.

```
//--- 2. загрузка данных обучающей выборки
   CArrayObj data;
   CArrayObj targets;
   if(!LoadTrainingData(StudyFileName, data, targets))
      return;
```

Как и было сказано выше, после создания модели и загрузки обучающей выборки мы можем начать процесс обучения. Данный функционал будет возложен на метод *NetworkFit*, в параметрах которому мы передадим указатели на нашу модель, обучающую выборку с целевыми значениями и вектор записи динамики изменения ошибки модели в процессе обучения.

```
//--- 3. обучение модели
   if(!NetworkFit(net, data, targets, loss_history))
      return;
```

После завершения процесса обучения модели мы сохраняем историю изменения ошибки модели в процессе обучения. Также сохраним обученную модель. Для сохранения обученной модели нам нет необходимости создавать отдельную функцию, так как в данном случае мы можем воспользоваться ране созданным методом нашего базового класса нейронной сети сохранения модели.

```
//--- 4. сохранение истории ошибок модели
   SaveLossHistory(OutputFileName, loss_history);
//--- 5. сохраняем полученную модель
   net.Save("Study.net");
   Print("Done");
  }
```

В подтверждение успешного завершения всех операций выведем информационное сообщение в журнал и завершим работу скрипта.

Как можно заметить, код основной функции скрипта получился довольно сжатым, но четко структурированным. Это выгодно отличает его от скрипта проверки корректности распределения градиента в предыдущем разделе. Выбор стиля программирования остается за программистом и не влияет на функциональность нашей библиотеки. Мы же возвращаемся к нашему скрипту и теперь займемся написанием функций, которые выше мы вызывали из основной функции скрипта.

Первой по списку у нас идет функция инициализация модели *NetworkInitialize*. В параметрах данной функции мы передаем указатель на объект создаваемой модели. В теле функции нам предстоит инициализировать модель перед обучением. Для инициализации модели нам необходимо дать описание создаваемой модели. Напомню, описание модели мы создаем в динамическом массиве, каждый элемент которого содержит указатель на экземпляр объекта *CLayerDescription* с описанием архитектуры конкретного нейронного слоя. Саму операцию создания подобного описания модели мы вынесли в отдельную функцию *CreateLayersDesc*, что является естественным продолжением концепции структурированного кода.

```
//+------------------------------------------------------------------+
//| Инициализация модели                                             |
//+------------------------------------------------------------------+
bool NetworkInitialize(CNet &net)
  {
   CArrayObj layers;
//--- создаем описание слоев сети
   if(!CreateLayersDesc(layers))
      return false;
```

После создания описания архитектуры модели мы вызываем метод инициализации нашей нейронной сети *CNet::Create*. В него передаем описание архитектуры модели, коэффициент обучения, параметры оптимизации, функцию потерь и параметры регуляризации. Не забываем проверить результат выполнения операций создания модели.

```
//--- инициализируем сеть
   if(!net.Create(&layers,(TYPE)LearningRate,(TYPE)0.9,(TYPE)0.999,LOSS_MSE,0,0))
     {
      PrintFormat("Error of init Net: %d", GetLastError());
      return false;
     }
   net.UseOpenCL(UseOpenCL);
   net.LossSmoothFactor(BatchSize);
   return true;
  }
```

После успешной инициализации модели установим флаг использования технологии *OpenCL* и размер пакета для усреднения ошибки модели. В приведенном примере последний устанавливается на уровне пакета обновления матрицы весов.

Чтобы завершить описание процесса инициализации модели, предлагаю сразу посмотреть на функцию создания описания архитектуры модели *CreateLayersDesc*. В параметрах метод получает указатель на динамический массив, в который мы и запишем архитектуру создаваемой модели.

Первым мы создаем описание слоя исходных данных. Количество нейронов во входном слое исходных данных зависит от двух внешних параметров: количества исторических баров в одном паттерне (*BarsToLine*) и количества нейронов входного слоя на один бар (*NeuronsToBar*). Определяется количество их произведением. Входной слой будет без функции активации и не будет обучаться. Это вполне понятно и не должно вызывать вопросов, ведь в массив результатов данного слоя мы закладываем исходные параметры из внешней системы, а внутри слоя не осуществляется никаких операций над данными.

```
bool CreateLayersDesc(CArrayObj &layers)
  {
//--- создаем слой исходных данных
   CLayerDescription *descr;   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   descr.type         = defNeuronBase;
   descr.count        = NeuronsToBar * BarsToLine;
   descr.window       = 0;
   descr.activation   = AF_NONE;
   descr.optimization = None;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      return false;
     }
```

При использовании полносвязных нейронных слоев каждый нейрон скрытого слоя можно рассматривать в качестве определенного паттерна, который нейронная сеть выучит в процессе обучения. В такой логике количество нейронов скрытого слоя представляет собой количество паттернов для запоминания нейронной сетью. Конечно, можно выстроить логическую взаимосвязь между количеством элементов в предыдущем слое и количеством возможных их комбинаций, которые будут представлять паттерны. Но не будем забывать, что наши результаты предыдущего нейронного слоя — небинарные величины, и диапазон их изменения довольно велик. Поэтому полное количество возможных комбинаторных вариантов паттернов получится очень большое. Среднестатистическая вероятность их появления будет сильно отличаться. Поэтому чаще всего количество нейронов в каждом скрытом слое будет определяться архитектором нейронной сети в каком-то диапазоне, а точное количество подбирается по лучшим показателям на валидационной выборке. Именно поэтому мы предоставили пользователю возможность указать количество нейронов в скрытом слое во внешнем параметре *HiddenLayer*. Но сразу скажем, что мы будем создавать все нейронные слои одинаковой архитектуры и одного размера.

Количество скрытых слоев зависит от сложности решаемой задачи и также определяется архитектором нейронной сети. В данном тесте я буду использовать нейронную сеть с одним скрытым слоем. Но в то же время, я предлагаю вам самостоятельно провести несколько экспериментов с разным количеством слоев и оценить влияния изменения данного параметра на результат. Для проведения таких экспериментов мы вывели отдельный внешний параметр — количество скрытых слоев (*HiddenLayers*).

На практике мы создаем описание одного скрытого слоя и затем добавляем его в динамический массив описания архитектуры столько раз, сколько нам необходимо создать скрытых нейронных слоев.

```
//--- скрытый слой
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   descr.type         = defNeuronBase;
   descr.count        = HiddenLayer;
   descr.activation   = AF_SWISH;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
   for(int i = 0; i < HiddenLayers; i++)
      if(!layers.Add(descr))
        {
         PrintFormat("Error adding layer: %d", GetLastError());
         return false;
        }
```

В рамках данного раздела я не ставлю целью полное обучение нейронной сети с получением максимально возможных результатов. Мы лишь сравним работу нашей библиотеки в различных режимах и их влияние на результаты обучения. Также посмотрим на практике влияние некоторых подходов, которые мы обсуждали в теоретической части книги. Поэтому мы не будем сейчас сильно останавливаться на тщательном выборе архитектурных параметров нейронной сети для получения максимальных результатов.

В качестве функции активации для скрытого слоя я указал *Swish*. Это одна из тех функций, диапазон значений которой ограничен снизу и не ограничен сверху. При этом функция дифференцируема на всем протяжении допустимых значений. Но в процессе тестирования мы сможем оценить и другие функции активации.

Выбор функции активации для выходного слоя является компромиссным решением. Дело в том, что у нас есть две цели: направление движения и сила движения. Это не совсем стандартный подход к решению задачи, так как у нас на выходе нейронной сети два нейрона с абсолютно разными значениями. Можно сказать, что у них даже класс значений разный. Если направление движения можно отнести к бинарной классификации (покупка или продажа), то определение силы движения является регрессионной задачей. И, наверное, было бы логично нейронную сеть обучить только на определение силы движения, а направление бы соответствовало знаку результата. Но мы же учимся и экспериментируем. Давайте посмотри на поведение нейронной сети в столь нестандартной ситуации, а активировать нейроны попробуем линейной функцией, как стандартной для решения задач регрессии.

В качестве метода обучения для обоих нейронных слоев я указал Adam.

Алгоритм описания нейронных слоев полностью идентичен рассмотренному в предыдущем разделе. Сначала мы описываем каждый слой в объекте класса *CLayerDescription*. Последовательность описания слоев соответствует их последовательности в нейронной сети от входного слоя исходных данных до выходного слоя результатов. По мере описания слоев добавляем их в коллекцию ранее созданного динамического массива.

```
//---  слой результатов
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   descr.type         = defNeuronBase;
   descr.count        = 2;
   descr.activation   = AF_LINEAR;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      return false;
     }
   return true;
  }
```

Следующим этапом работы нашего скрипта стала загрузка обучающей выборки в функции *LoadTrainingData*. Загружать мы ее будем из файла, указанного в параметрах функции. В теле функции мы сразу открываем указанный файл для чтения и проверяем результат операции по значению полученного хендла.

```
//+------------------------------------------------------------------+
//| Загрузка данных обучения                                         |
//+------------------------------------------------------------------+
bool LoadTrainingData(string path, CArrayObj &data, CArrayObj &targets)
  {
   CBufferType *pattern;
   CBufferType *target;
//--- открываем файл с обучающей выборкой
   int handle = FileOpen(path, FILE_READ | FILE_CSV | FILE_ANSI | FILE_SHARE_READ,
                                                                  ",", CP_UTF8);
   if(handle == INVALID_HANDLE)
     {
      PrintFormat("Error opening study data file: %d", GetLastError());
      return false;
     }
```

Операцию загрузки обучающей выборки мы будем осуществлять в два этапа. Сначала поэлементно будем загружать паттерны и целевые значения в два буфера *CBufferType*. В буфере *pattern* будем собирать элементы исходных данных одного паттерна, а в буфере *target* — соответствующие ему целевые результаты.

```
//--- выводим прогресс загрузки данных обучения в комментарий чарта
   uint next_comment_time = 0;
   enum
     {
      OutputTimeout = 250 // не чаще 1 раза в 250 миллисекунд
     };
//--- организовываем цикл загрузки обучающей выборки
   while(!FileIsEnding(handle) && !IsStopped())
     {
      if(!(pattern = new CBufferType()))
        {
         PrintFormat("Error creating Pattern data array: %d", GetLastError());
         return false;
        }
      if(!pattern.BufferInit(1, NeuronsToBar * BarsToLine))
         return false;
      if(!(target = new CBufferType()))
        {
         PrintFormat("Error creating Pattern Target array: %d", GetLastError());
         return false;
        }
      if(!target.BufferInit(1, 2))
         return false;
      for(int i = 0; i < NeuronsToBar * BarsToLine; i++)
         pattern.m_mMatrix[0, i] = (TYPE)FileReadNumber(handle);
      for(int i = 0; i < 2; i++)
         target.m_mMatrix[0, i] = (TYPE)FileReadNumber(handle);
```

После загрузки из файла информации одного паттерна сохраним указатели на объекты с данными в два динамических массива *CArrayObj*. Указатели на них мы также получили в параметрах функции. Один массив используется для паттернов исходных данных (*data*), а второй — для целевых значений (*targets*). Операции повторим в цикле до достижения конца файла. Для мониторинга процесса пользователем выведем информацию о количестве загруженных паттернов на график в поле комментариев.

Обратите внимание, что поскольку мы передаем в динамические массивы указатели на объекты с данными, то после записи указателей в массив нам необходимо создать новые экземпляры объектов *CBufferType*. В противном случае мы заполним весь динамический массив указателем на один и тот же экземпляр объекта, а буфер будет содержать безликую информацию обо всех паттернах, работа с которыми потребует другого алгоритма. Следовательно, вся нейронная сеть будет работать некорректно.

```
      if(!data.Add(pattern))
        {
         PrintFormat("Error adding study data to array: %d", GetLastError());
         return false;
        }
      if(!targets.Add(target))
        {
         PrintFormat("Error adding study data to array: %d", GetLastError());
         return false;
        }
      //--- выводим прогресс загрузки в комментарий чарта
      //--- (не чаще 1 раза в 250 миллисекунд)
      if(next_comment_time < GetTickCount())
        {
         Comment(StringFormat("Patterns loaded: %d", data.Total()));
         next_comment_time = GetTickCount() + OutputTimeout;
        }
     }
   FileClose(handle);
   return true;
  }
```

После завершения цикла считывания данных мы получим два массива объектов с одинаковым количеством элементов. В них элементы с одним индексом будут составлять пару «исходные — целевые» данные паттерна. Здесь закрываем файл обучающей выборки.

Теперь, когда у нас уже создана нейронная сеть и загружена обучающая выборка, мы можем приступить к обучению в функции *NetworkFit*. В параметрах данный метод получает указатели на объекты нейронной сети и обучающей выборки. Также он получает указатель на вектор записи динамики изменения ошибки модели в процессе обучения. Для обучения нейронной сети мы создадим два вложенных цикла. Первый цикл запустим с числом итераций равным внешнему параметру количества обновлений матрицы весов *Epochs*. Во вложенном цикле создадим число итераций равное размеру пакета для обновления весовых коэффициентов *BatchSize*.

```
bool NetworkFit(CNet &net, const CArrayObj &data,
                const CArrayObj &target, VECTOR &loss_history)
  {
//--- обучение
   int patterns = data.Total();
//--- цикл по эпохам
   for(int epoch = 0; epoch < Epochs; epoch++)
     {
      ulong ticks = GetTickCount64();
      //--- обучаем батчами
      for(int i = 0; i < BatchSize; i++)
        {
         //--- проверим на остановку обучения
         if(IsStopped())
           {
            Print("Network training stopped by user");
            return true;
           }
```

В теле вложенного цикла будем случайным образом выбирать из обучающей выборки по одному паттерну. Для каждого выбранного паттерна сначала сделаем прямой проход на соответствующих исходных данных. Затем откроем целевые значения и сделаем обратный проход.

```
         //--- выбор случайного паттерна
         int k = (int)((double)(MathRand() * MathRand()) / MathPow(32767.0, 2) *
                                                                  patterns);
         if(!net.FeedForward(data.At(k)))
           {
            PrintFormat("Error in FeedForward: %d", GetLastError());
            return false;
           }
         if(!net.Backpropagation(target.At(k)))
           {
            PrintFormat("Error in Backpropagation: %d", GetLastError());
            return false;
           }
        }
```

Повторяя итерации прямого и обратного проходов, мы накапливаем градиент ошибки на каждом элементе матрицы весов. После набора числа итераций прямого и обратного проходов до размера пакета для обновления матрицы весов выходим из внутреннего цикла. Обновляем весовые коэффициенты в направлении среднего градиента ошибки. Очищаем буфер накопленных градиентов ошибки. Сохраняем текущее значение функции потерь в вектор для отслеживания процесса обучения. После этого заходим на новый цикл итераций обучения.

```
      //--- перенастраиваем веса сети
      net.UpdateWeights(BatchSize);
      printf("Use OpenCL %s, epoch %d, time %.5f sec", (string)UseOpenCL,
                                epoch, (GetTickCount64() - ticks) / 1000.0);
      //--- сообщим о прошедшей эпохе
      TYPE loss = net.GetRecentAverageLoss();
      Comment(StringFormat("Epoch %d, error %.5f", epoch, loss));
      //--- запомним ошибку эпохи для сохранения в файл
      loss_history[epoch] = loss;
     }
   return true;
  }
```

В предложенном примере процесс обучения ограничивается внешним параметром числа итераций обновления матрицы весов. На практике часто применяется подход, когда процесс обучения прекращается при достижении заданных результатов метрик качества обучения. Это может быть значение функции потерь, доля попаданий в ожидаемые результаты и т.д. А можно применять и гибридный подход, при котором отслеживаются метрики и в тоже время задается максимальное число итераций обучения.

После завершения процесса обучения мы сохраняем в файл динамику изменения функции потерь. Этот функционал выполняется функцией *SaveLossHistory*, в параметрах которой мы передадим имя файла для записи данных и вектор динамики изменений ошибки модели в процессе обучения.

В теле функции мы открываем или создаем новый *CSV*-файл для записи данных и в цикле сохраняем все значения ошибки модели в процессе обучения.

```
void SaveLossHistory(string path, const VECTOR &loss_history)
  {
   int handle = FileOpen(OutputFileName, FILE_WRITE | FILE_CSV | FILE_ANSI,
                                                                ",", CP_UTF8);
   if(handle == INVALID_HANDLE)
     {
      PrintFormat("Error creating loss file: %d", GetLastError());
      return;
     }
   for(ulong i = 0; i < loss_history.Size(); i++)
      FileWrite(handle, loss_history[i]);
   FileClose(handle);
   printf("The dynamics of the error change is saved to a file %s\\MQL5\\Files\\%s",
                            TerminalInfoString(TERMINAL_DATA_PATH), OutputFileName);
  }
```

После записи данных в файл закрываем файл и выводим в журнал информационное сообщение с указанием полного пути сохраненного файла.

В рассмотренном примере реализации скрипта показана полная загрузка в память обучающей выборки. Конечно, работа с оперативной памятью всегда быстрее обращения к постоянной памяти. Но не всегда размеры обучающей выборки позволяют ее полностью загрузить в оперативную память. В таких случаях обучающую выборку загружают и обрабатывают частями.

Нормализация данных на входе нейронной сети

После создания такого скрипта мы можем провести несколько поучительных экспериментов. К примеру, ранее мы обсуждали важность нормализации исходных данных перед подачей их на вход нейронной сети. А насколько это важно? Почему в процессе обучения нейронной сети нельзя подобрать подходящие весовые коэффициенты, которые бы учитывали масштаб данных? Да, мы говорили о влиянии больших величин. Но сейчас мы можем провести практический эксперимент и увидеть влияние нормализации исходных данных на результат обучения модели.

Возьмем исторические данные по инструменту EURUSD, пятиминутный таймфрейм за период с 01.01.2015 по 31.12.2020 и создадим две обучающие выборки с нормированными и ненормированными данными. Запустим вышеописанный скрипт обучения нейронной сети на обоих выборках. Скрипт создания обучающей выборки мы создали в разделе 3.9.

График динамики функции потерь более чем красноречив. Ошибка на нормированных данных значительно ниже, даже если мы начинаем со случайными весовыми коэффициентами. Если на ненормированных данных стартовое значение функции потерь около 120, то на нормированных данных оно составляет только 0,6. Конечно, в процессе обучения значение функции потерь на ненормированных данных стремительно падает и после 200 итераций обновления весовых коэффициентов снижается до 6, а после 1000 итераций достигает 4,5. Но несмотря на столь стремительную динамику снижения показателя функции потерь оно все же значительно превосходит показатели для нормированных данных. На последних функция потерь после 1000 итераций обновления матрицы весов приближается к 0.44.

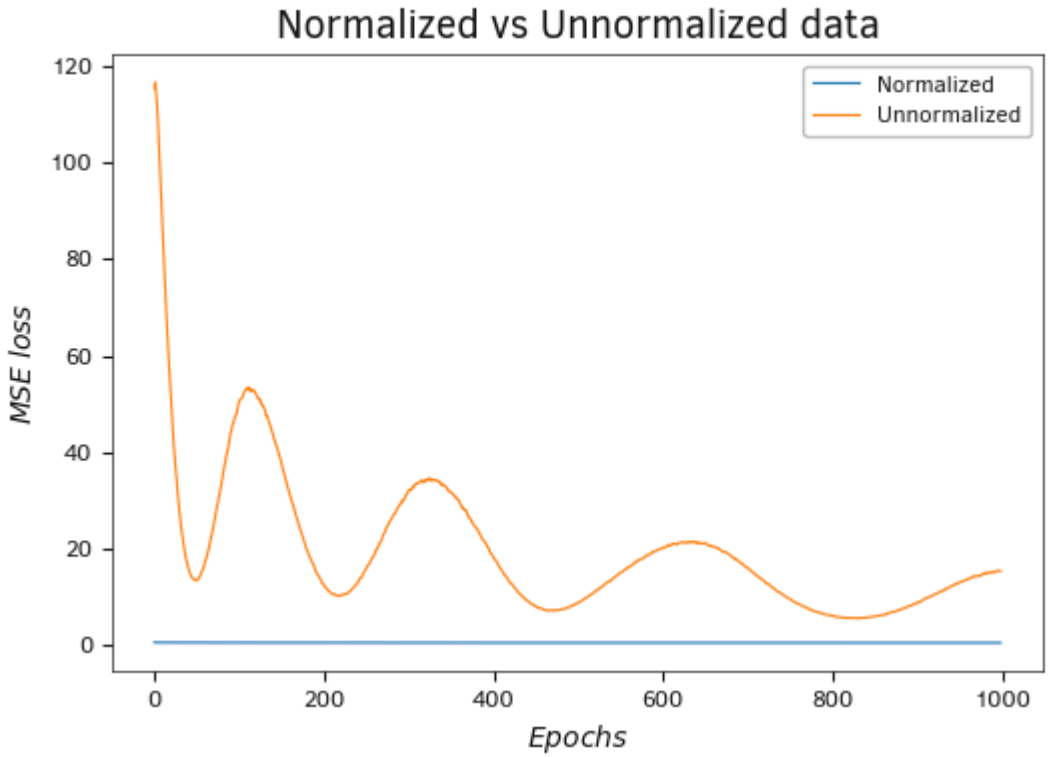


График динамики функции потерь MSE при обучении нейронной сети на нормированных и ненормированных данных

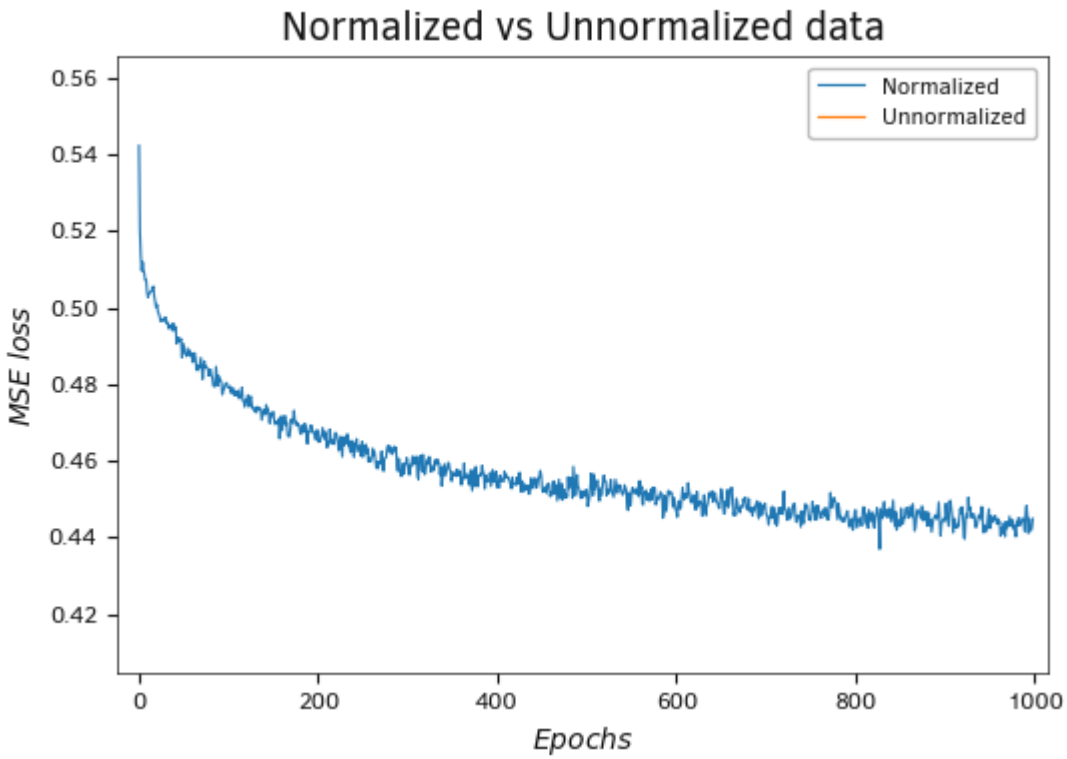


График динамики функции потерь MSE при обучении нейронной сети на нормированных и ненормированных данных (масштаб)

Я провел подобный эксперимент как с использованием технологии OpenCL, так и без. Результаты работы нейронной сети оказались сопоставимыми. А вот по производительности на столь малой нейронной сети выиграла CPU. Очевидно, что накладные расходы на передачу данных оказались значительно выше прироста производительности за счет использования многопоточной технологии. Такие результаты были ожидаемы. Мы уже говорили ранее, что использование подобной технологии оправдано для больших нейронных сетей, когда затраты на передачу данных между устройствами покрываются приростом производительности за счет разделения вычислительных итераций на параллельные потоки.

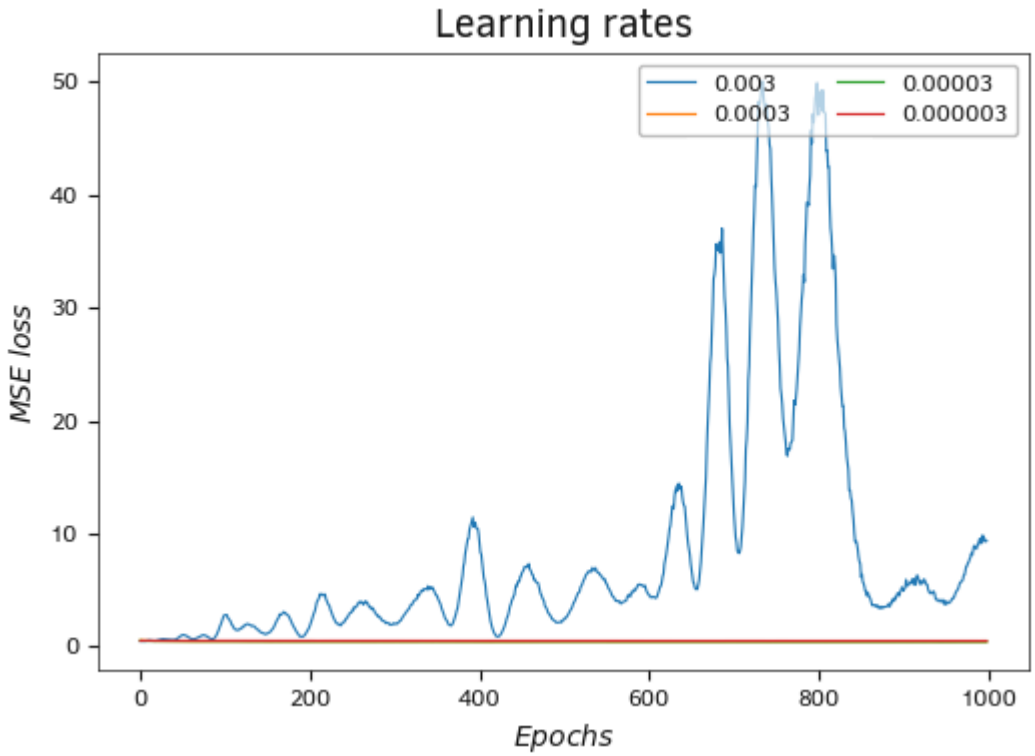
Я предлагаю вам повторить подобный опыт со своими данными — тогда у вас не останется вопроса о необходимости нормирования исходных данных.

Думаю, после проведенного эксперимента становится очевидным, что дальнейшее тестирование будет проводиться на нормированных данных.

Выбор коэффициента обучения

Следующий вопрос, который всегда становится перед создателями нейронных сетей, — это выбор коэффициента обучения. При решении этой задачи необходимо соблюсти баланс между производительностью и качеством обучения. Выбор заведомо большого коэффициента обучения позволяет быстрее снижать ошибку в начале обучения. Но потом скорость обучения быстро снижается и в лучшем случае останавливается далеко от заведомой цели. В худшем случае ошибка начинает расти. Выбор заведомо малого коэффициента обучения снижает скорость обучения. Процесс занимает больше времени, и при этом повышается риск «застрять» в локальном минимуме, так и не достигнув желаемой цели.

Для экспериментального тестирования влияния скорости обучения на процесс обучения нейронной сети проведем обучение созданной выше нейронной сети при четырех различных коэффициентах обучения: 0,003, 0,0003, 0,00003 и 0,000003. Результаты тестирования представлены на графике ниже.

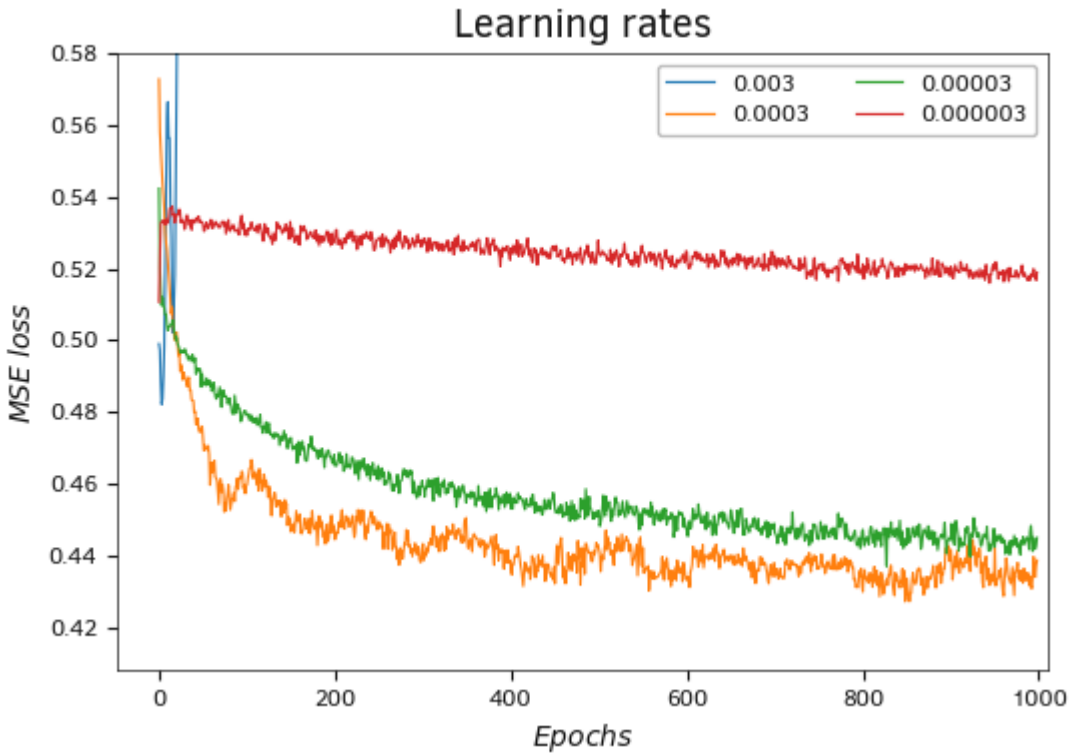


Сравнение динамики функции потерь при использовании разной скорости обучения

При обучении нейронной сети с использованием коэффициента обучения 0,003 наблюдаются колебания функции потерь. В процессе обучения амплитуда колебаний растет. В целом наблюдается тенденция к росту ошибки модели. Такое поведение характерно для завышенного коэффициента обучения.

Снижение коэффициента обучения делает график обучения более плавным. Но при этом снижается и скорость падения значения функции потерь с каждым обновлением матрицы весов. Наиболее ровное снижение значения функции потерь демонстрирует процесс обучения с коэффициентом обучения 0,000003. Но за плавность графика пришлось заплатить ростом количества итераций обновления матрицы весов для достижения оптимального результата. За весь процесс обучения в 1000 итераций обновления матрицы весов коэффициент обучения 0,000003 показал наихудший результат из всех.

Обучение нейронной сети с коэффициентами 0,0003 и 0,00003 показали близкие результаты. График функции потерь при коэффициенте обучения 0,00003 получился более рваный. Но в то же время наилучший результат по значению ошибки показало обучение с коэффициентом 0,0003.



Сравнение динамики функции потерь при использовании разной скорости обучения (масштаб)

Подбор количества нейронов в скрытом слое

Следующий момент, который бы хотелось продемонстрировать на практике — это влияние количества нейронов в скрытом слое на процесс обучения и его результат. Когда мы говорим о полносвязных нейронных слоях, в которых каждый нейрон последующего слоя имеет взаимосвязи со всеми нейронами предыдущего слоя и при этом каждая связь индивидуальна и независима, логично предположить, что каждый нейрон будет активирован своей комбинацией из состояний нейронов предыдущего слоя. Таким образом, каждый нейрон реагирует на свой паттерн состояния предыдущего слоя. Следовательно, наличие большего числа нейронов в скрытом слое потенциально способно запомнить больше таких паттернов и сделать их более детальными. При этом мы не программируем вариации паттернов, а позволяем нейронной сети их выучить самостоятельно из представленной обучающей выборки. Казалось бы, в такой логике увеличение количества нейронов в скрытом слое позволяет только увеличить качество обучения нейронной сети.

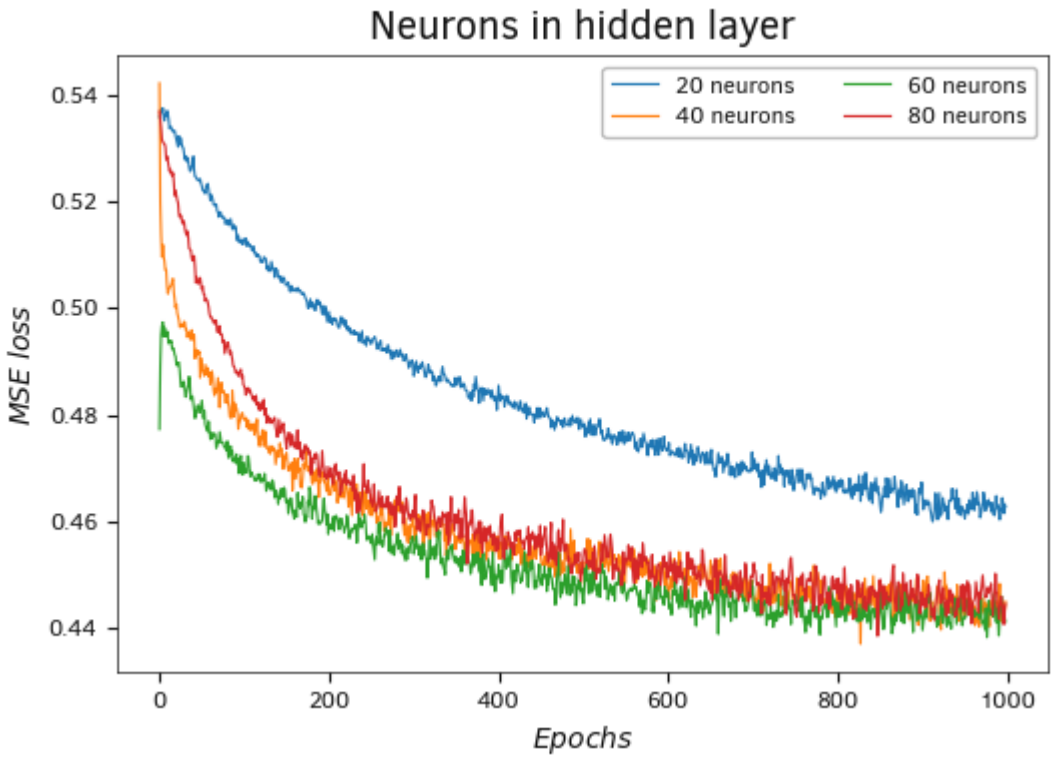
Но на практике не все паттерны обладают равной долей вероятности к появлению. Цель обучения нейронной сети не в том, чтобы заучить каждое отдельное состояние до мельчайших подробностей. Их цель — используя обучающую выборку, обобщить представленные в ней данные, найти и выделить зависимости и закономерности. Полученные данные должны позволить построить функцию зависимости целевых значений от исходных данных с требуемой точностью. Поэтому безмерное увеличение нейронов в скрытом слое снижает способность нейронной сети к обобщению и ведет к переобучению нейронной сети.

Вторая сторона вопроса увеличения числа нейронов в скрытом слое — это увеличение затрат временных и вычислительных ресурсов. Дело в том, что добавление одного нейрона в скрытом слое добавляет к матрице весов столько элементов, сколько содержит предыдущий слой плюс один элемент для *bias*. Следовательно, выбирая количество нейронов в скрытом слое, следует

учитывать баланс между получаемым качеством обучения и затратами на обучение такой нейронной сети. В то же время нужно помнить о риске переобучения.

Конечно, существуют выработанные методы борьбы с переобучением нейронных сетей. Это в первую очередь увеличение обучающей выборки и регуляризация. Теоретические аспекты регуляризации мы уже обсуждали ранее, а о практическом использовании поговорим чуть позже.

Сейчас же предлагаю посмотреть на графики значения функции ошибки при обучении нейронной сети с одним скрытым слоем, в котором меняется число нейронов при прочих равных условиях. При тестировании я сравнил процесс обучения 4 нейронных сетей с 20, 40, 60 и 80 нейронами в скрытом слое. Конечно, подобное количество нейронов слишком мало для получения каких-либо достойных результатов обучения на выборке в 350 тыс. паттернов. И тем более здесь нет никакого риска переобучения. Но их достаточно, чтобы посмотреть на влияние данного фактора на процесс обучения.



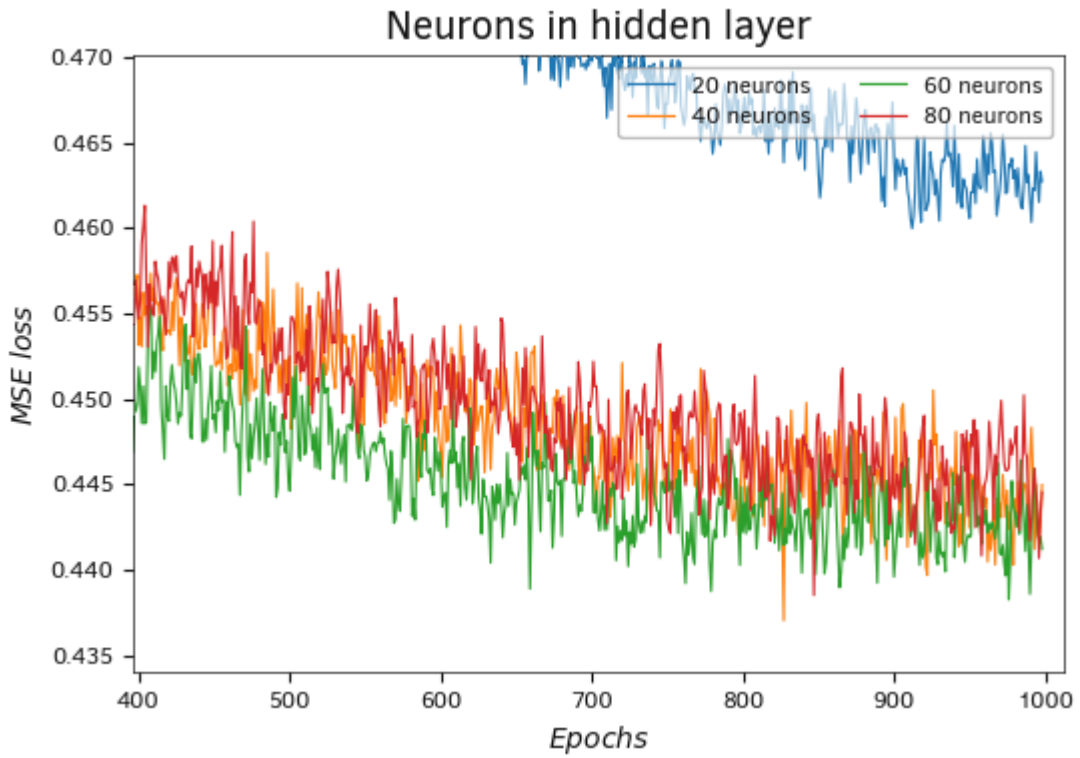
Сравнение динамики функции потерь при использовании разного количества нейронов в скрытом слое

Как можно заметить на графике, наихудший результат показала модель с 20 нейронами в скрытом слое. Их явно недостаточно для решения подобной задачи.

Касательно остальных трех моделей можно сказать, что разброс графиков в первые 100 итераций обновлений весовых коэффициентов можно списать на фактор случайности вследствие инициализации моделей случайными весами. Примерно после 250–300 итераций обновлений матрицы весов графики переплетаются в единый пучок и идут дальше вместе.

Увеличение масштаба графика позволяет выявить основную тенденцию: с увеличением числа нейронов в слое увеличивается количество итераций для достижения локальных минимумов и в целом для обучения нейронной сети. При этом локальные минимумы нейронных сетей с большим количеством нейронов опускаются ниже, и их графики обладают меньшей частотой колебаний.

В целом же на протяжении всего обучения модель с 60 нейронами демонстрирует наилучшие показатели. С небольшим отставанием почти параллельно идет график модели с 40 нейронами в скрытом слое. Для модели же с 80 нейронами в скрытом слое 1000 итераций обновления матрицы весов оказалось недостаточно. Эта модель демонстрирует более медленное снижение значения функции потерь. В то же время график динамики значений функции потерь демонстрирует потенциал снижения значения функции потерь при дальнейшем обучении модели. И есть все основания ожидать снижение показателей, достигнутых моделью с 60 нейронами в скрытом слое.



Сравнение динамики функции потерь при использовании разного количества нейронов в скрытом слое (масштаб)

Однако следует обратить внимание на то, что снижение ошибки на обучающей выборки может быть связано и с переобучением модели. Поэтому перед практическим использованием обученной модели всегда следует провести ее тестирование на «незнакомых» данных.

Обучение, валидация, тестирование.

В процессе обучения мы подбираем такие параметры матрицы весов, чтобы добиться минимальной ошибки на обучающей выборке. Но как поведет себя модель на новых данных за пределами обучающей выборки? Следует еще учесть, что мы имеем дело с нестатичными данными, которые постоянно меняются, и на их изменение оказывает влияние большое количество факторов. Некоторые из таких факторов нам известны, а о некоторых мы даже не догадываемся. И даже об известных нам факторах мы не можем с точностью сказать, как они изменятся в будущем. И тем более мы не знаем, как это повлияет на изменение изучаемых данных. Наиболее вероятным будет ухудшение показателей работы нейронной сети на новых данных. Но каким будет это ухудшение? Готовы ли мы нести такие риски?

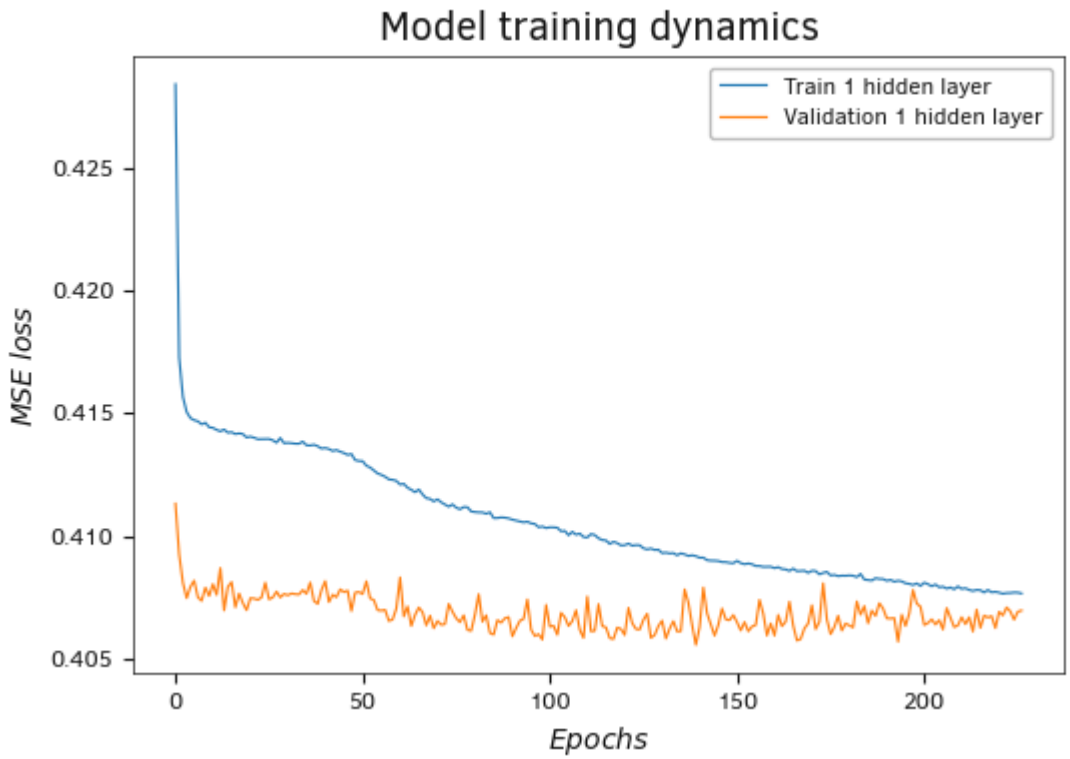
Первым шагом на пути решения этого вопроса является валидация параметров обучения модели. Для валидации модели используется набор данных, не входящих в обучающую выборку. Чаще всего весь набор исходных данных делится на три блока:

- обучающая выборка (~60%),
- валидационная выборка (~20%),
- тестовая выборка (~20%).

Процентное соотношение для каждой выборки приведено условно и может сильно отличаться в зависимости от поставленной задачи.

Суть процесса валидации заключается в проверке параметров обучаемой модели на данных, не входящих в обучающую выборку. В процессе валидации подбираются гиперпараметры обучаемой модели для достижения максимального результата.

При написании скрипта полносвязного перцептрона на языке Python мы выделили 20% от обучающей выборки для валидации. Обучение первой модели продемонстрировало результаты схожие с полученными при обучении модели, созданной в MQL5. Это положительный для нас сигнал. Получение схожих результатов при обучении моделей созданных в трех разных языках программирования может свидетельствовать о правильности реализации выстроенного нами алгоритма.



Изменение показателей модели с одним скрытым слоем на валидации в темпе с её обучением

Оценивая графики результатов тестирования, можно заметить тенденцию к снижению ошибки в процессе обучения. Это положительный фактор, который свидетельствует о способности модели к обучению и выстраиванию зависимостей между исходными данными и целевыми метками. В то же время можно заметить рост ошибки на валидационных данных, что может говорить как о переобучаемости модели, так и о нерепрезентативности данных валидационной выборки.

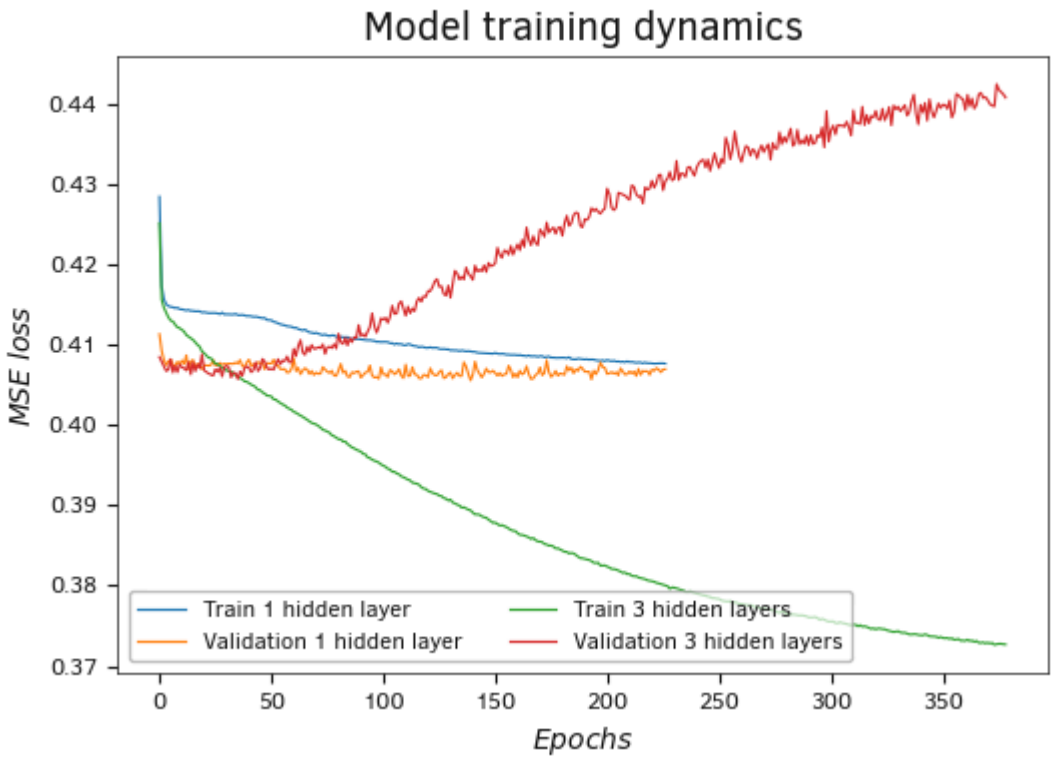
Дело в том, что мы указали долю данных для валидации в обучающей выборке. В этом случае библиотекой TenzorFlow берутся последние данные в наборе обучающей выборки. Но такой подход не всегда дает верный результат, ведь на результаты отдельно взятого периода могут большое влияние оказывать локальные тенденции.

В представленном ниже графике я вижу влияние обоих факторов. Общая динамика к росту ошибки на валидационных данных свидетельствует о склонности модели к переобучению, а начальное положение ошибки валидации ниже ошибки обучения может говорить о влиянии локальных тенденций.

Изменение показателей модели с одним скрытым слоем на валидации в темпе с её обучением

График метрики *accuracy* демонстрирует схожие тенденции. Сам показатель отображает долю правильных ответов модели в общем объеме результатов. Здесь мы видим рост показателя в процессе обучения при практически неизменном показателе на валидации. Это может свидетельствовать о том, что модель учит паттерны, не встречающиеся в валидационной выборке.

Теоретически добавление скрытых слоев должно повысить способность модели к обучению и распознаванию более сложных паттернов и фигур. Вторую модель в Python мы создали с тремя скрытыми слоями. Действительно, в этой модели на обучении ошибка значительно снизилась. Но в то же время она еще больше возросла в процессе валидации. Это явный признак переобученности модели. Когда модель в силу своих возможностей не обобщает зависимости, а просто «заучивает» пары «исходные данные — целевые значения», на новых данных, не входящих в обучающую выборку, результат проявляется случайным образом.



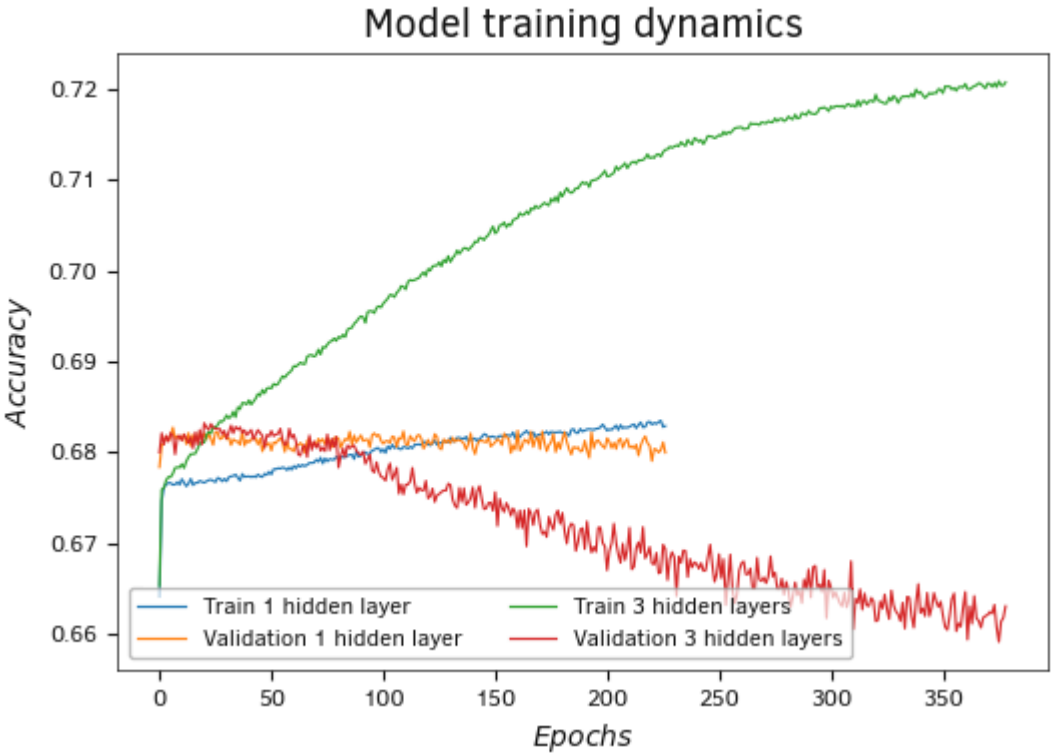
Изменение показателей модели с тремя скрытыми слоями на валидации в темпе с её обучением

Динамика метрики *accuracy* имеет тенденции схожие с функцией потерь. С той лишь разницей, что функция потерь снижается в процессе обучения, а *accuracy* растет.

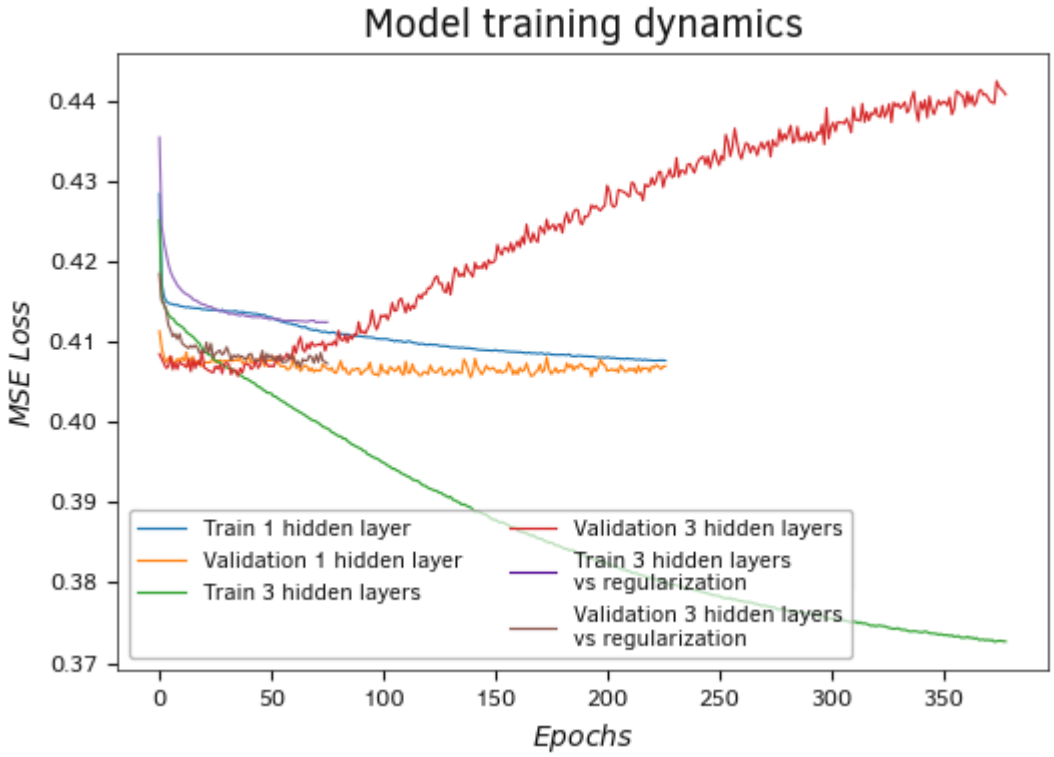
Один из способов борьбы с переобучением является регуляризация модели. В третью модель мы добавили регуляризацию *ElasticNet*, и она сделала свое дело. При обучении модели с регуляризацией ошибка снижалась с меньшей скоростью. В то же время и на валидации рост ошибки замедлился.

И вновь на графике метрики accuracy мы наблюдаем те же тенденции, что и на графике функции потерь.

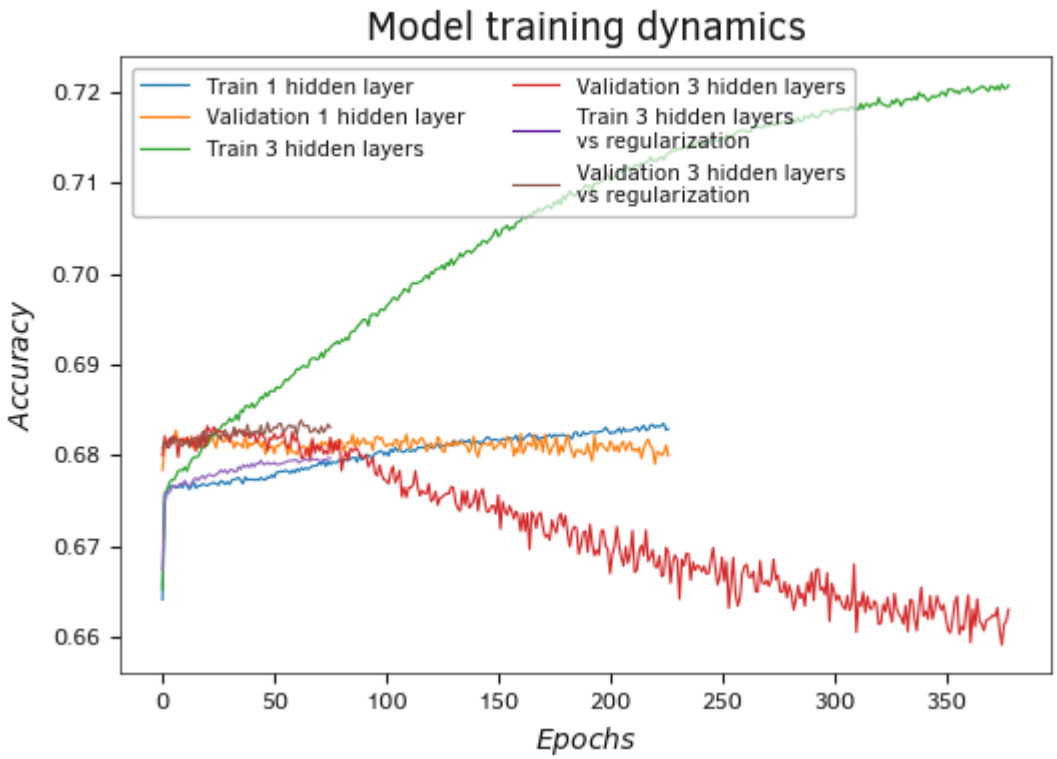
Надо сказать, что параметры ни обучения, ни регуляризации тщательно не подбирались. Поэтому, результаты обучения нельзя считать окончательными.



Изменение показателей модели с тремя скрытыми слоями на валидации в темпе с её обучением



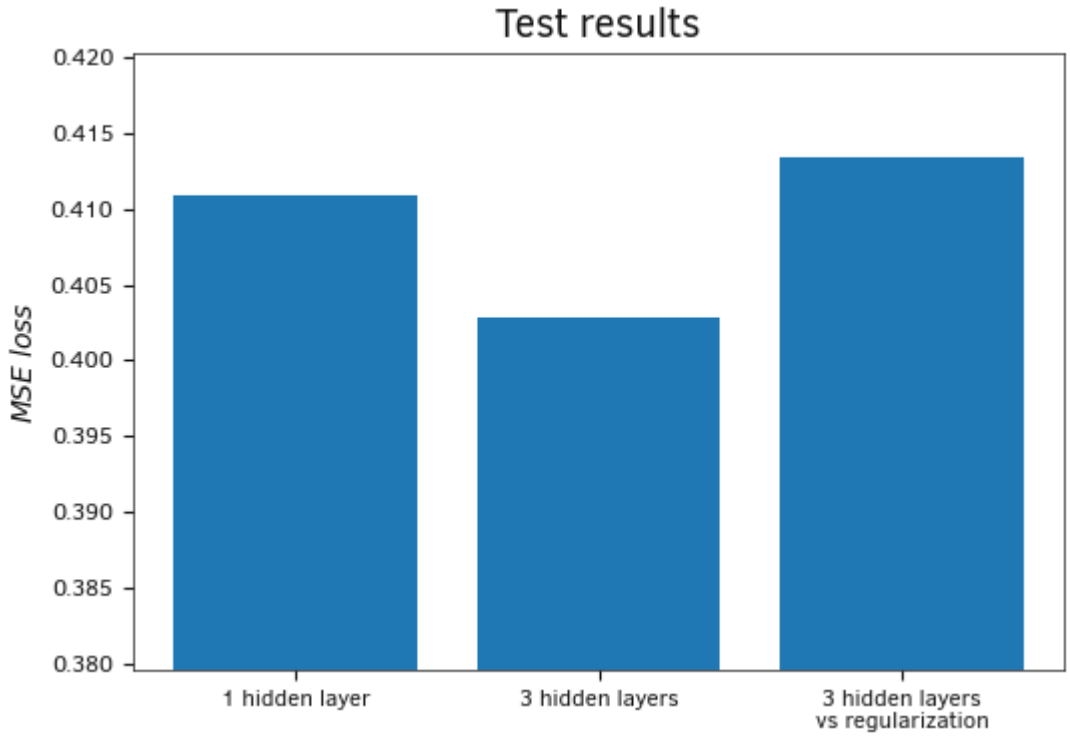
Изменение показателей модели с тремя скрытыми слоями и регуляризацией на валидации в темпе с её обучением



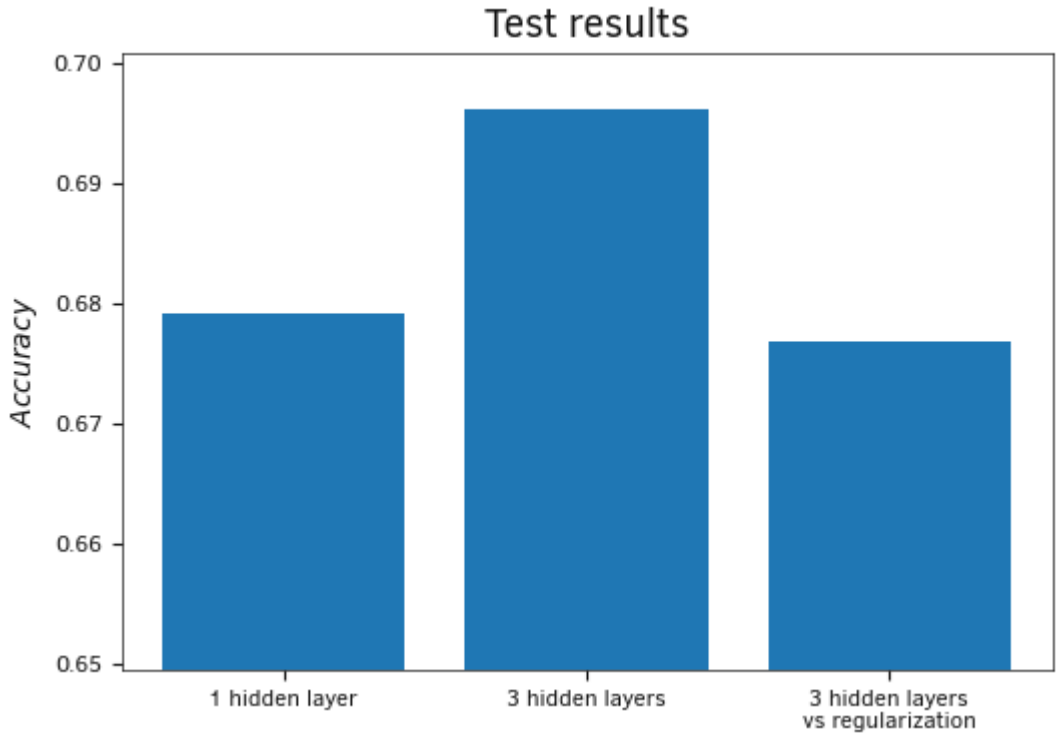
Изменение показателей модели с тремя скрытыми слоями и регуляризацией на валидации в темпе с её обучением

После обучения моделей проверим их на тестовой выборке. В отличие от валидации, минимальную ошибку на тестовой выборке показала именно модель с тремя скрытыми слоями без регуляризации. Модель с регуляризацией показала максимальную ошибку. Такое различие результатов между тестовой и валидационной выборками можно постараться объяснить способом создания выборок. Если в валидационную выборку попали только последние данные из обучающей выборки, то в тестовую выборку собирались случайные наборы данных из всей совокупности. Таким образом, тестовую выборку можно считать более репрезентативной, так как она лишена влияния локальных тенденций.

Измерение метрики *accuracy* на тестовой выборке показало аналогичные результаты. Наилучший результат показала модель с тремя скрытыми слоями.



Сравнение результатов моделей на тестовой выборке



Сравнение результатов моделей на тестовой выборке

Можно подвести итоги нашей небольшой практической работы.

1. Нормализация исходных данных перед подачей на вход нейронной сети сильно повышает шансы на сходимость нейронной сети и сокращает время на ее обучение.

2. Коэффициент обучения должен тщательно подбираться экспериментальным путем. Слишком большой коэффициент обучения ведет к разбалансированию нейронной сети и росту ошибки. Слишком маленький коэффициент обучения ведет к увеличению времени и расходованию вычислительных ресурсов на обучение нейронной сети. При этом повышается риск остановки процесса обучения в локальном минимуме без достижения желаемого результата.
3. Увеличение числа нейронов в скрытом слое дает улучшение результатов обучения. Но при этом растут и затраты на обучение. При выборе размера скрытого слоя необходимо найти баланс между ошибкой обучения и затратами ресурсов на проведение обучения нейронной сети. При этом следует помнить, что чрезмерное увеличение числа нейронов в скрытом слое повышает риск переобучения нейронной сети.
4. Увеличение числа скрытых слоев также повышает способность модели к обучению и распознаванию более сложных фигур и конструкций. При этом сильно возрастает склонность модели к переобучению.
5. Использование набора последних значений обучающей выборки для валидации не всегда способно показать истинные тенденции, так как такая валидационная выборка сильно подвержена влиянию локальных тенденций и не может быть репрезентативной.

Однако мы создаем модель для работы на финансовых рынках. Нам важно получение прибыли как в перспективе, так и в текущий момент. Конечно, какие-то локальные потери могут быть, но они не должны быть большими и частыми. Поэтому нам важно получить приемлемые результаты как на отдельном локальном участке данных, так и на более репрезентативной выборке. И наверное, получение лучших результатов на локальном участке имеет больший вес — после получения локальной прибыли мы можем переобучить модель для адаптации к новым тенденциям и получения прибыли на новом локальном участке. В то же время, если затраты на обучения будут превышать возможные локальные убытки, то больший вес приобретает доходность на длинном временном периоде с использованием репрезентативной выборки.

4. Базовые типы нейронных слоев

В предыдущих разделах мы познакомились с архитектурой полносвязного перцептрона и даже построили свою первую модель нейронной сети. Мы провели ее тестирование в различных режимах, получили первые результаты и первый опыт. Но используемые в перцептроне полносвязные нейронные слои на ряду со своими достоинствами обладают и некоторыми недостатками. К примеру, полносвязный слой анализирует только текущие данные без какой-либо связи с ранее обработанными данными. По существу, каждый пакет информации анализируется в информационном вакууме. Чтобы расширить объем анализируемых данных, нужно постоянно увеличивать размер модели. При этом расходы на обучение и эксплуатацию растут в геометрической прогрессии. Полносвязный слой анализирует всю совокупность как нечто целое и не выявляет зависимостей между отдельными элементами.

В этой главе мы рассмотрим различные архитектурные решения для построения нейронных слоев, предназначенных для преодоления недостатков полносвязных слоев, которые мы изучили ранее. Полносвязные нейронные сети анализируют данные без учета их контекста и связей между ними, что может приводить к недостаточной эффективности и увеличению объема модели. Мы рассмотрим следующие архитектурные подходы:

- Сверточные нейронные сети (CNN) - мы углубимся в их архитектуру и принципы реализации, а также рассмотрим способы построения их с использованием инструментов MQL5 и OpenCL. Далее мы изучим практическое тестирование сверточных моделей для оценки их производительности и эффективности.

- Рекуррентные нейронные сети (RNN) - мы также рассмотрим их архитектуру и принципы реализации, а затем изучим способы построения LSTM-блоков с помощью MQL5 и организации параллельных вычислений средствами OpenCL. Завершим главу реализацией рекуррентных моделей на Python и их сравнительным тестированием.

Таким образом, в этой главе мы углубимся в изучение сверточных и рекуррентных нейронных сетей, их принципов работы и применения в практических задачах, а также рассмотрим различные способы их построения и оптимизации.

4.1 Сверточные нейронные сети

Продолжаем наше погружение в архитектуры нейронных сетей. Сейчас я предлагаю рассмотреть принципы работы и построения сверточных нейронных сетей (Convolutional Neural Network). Данный вид нейронных сетей широко применяется в задачах распознавания объектов на фото- и видеоизображениях. Считается, что сверточные нейронные сети устойчивы к изменению масштаба, смене ракурса и прочим пространственным искажениям изображения. Их архитектура позволяет одинаково успешно находить объекты в любом месте сцены.

Помимо архитектурных отличий, о которых мы поговорим в следующей главе, между сверточными и полносвязными нейронными сетями есть кардинальное отличие в логической обработке входящего потока данных. Ранее мы обсуждали, что в полносвязных нейронных сетях каждый нейрон имеет связи со всеми нейронами предыдущего слоя и реагирует на наличие своего целостного паттерна в исходных данных. Давайте постараемся переложить это осознание на распознавание образов на изображении.

Представьте, вы обучаете нейронную сеть на распознавание напечатанных на листке бумаге цифр. Каждая цифра напечатана на идеально чистом листке бумаги, и ваша нейронная сеть научилась идеально их распознавать. Но стоит подать на вход сигнал с небольшим шумом, и ее результат будет непредсказуем, так как шум дополняет образ, после чего он уже не соответствует идеальным образам из обучающей выборки.

Возможна и обратная ситуация. Когда вы обучаете нейронную сеть на зашумленных изображениях, где изучаемый предмет находится на каком-то фоне, полносвязная нейронная сеть при достаточном количестве нейронов и достаточной обучающей выборке способна решить такую задачу. Но при этом она воспринимает картинку как единое целое. Стоит изменить или убрать фон, как это выведет полносвязную нейронную сеть из состояния равновесия, и результат ее работы будет не предсказуем. Все дело опять же в целостности восприятия мира. Когда при обучении с учителем мы подавали на вход нейронной сети изображение с шумом и давали ей правильный ответ, нейронная сеть сопоставила изображение с ответом и запомнила. Но она не выделила из картины нужный образ, а запомнила всю картинку целиком. Поэтому отсутствие фона нейронная сеть воспринимает как отсутствие некой составляющей образа. В такой случае выдать правильный результат будет затруднительно.

Помимо фона такая же ситуация возникает и при попытке вращения или масштабирования изображения. По существу, любое, даже незначительное, изменение исходных данных воспринимается полносвязной нейронной сетью как что-то новое, ранее невиданное. Это требует дополнительных ресурсов в виде нейронов для обработки и запоминания.

Помимо указанных выше проблем распознавания есть и проблема производительности. С ростом размера обрабатываемых изображений растет и размер входящего потока данных. Как следствие, растет и матрицы весов. Поэтому требуется больше памяти для хранения матрицы весов и

больше времени для обучения нейронной сети. При этом чаще всего большая часть изображения не несет полезной информации. Следовательно, ресурсы расходуются неэффективно.

Предложенные сверточные нейронные сети призваны были решить эти проблемы. В них использовалась технология, благодаря которой изображение не рассматривается целиком. Наоборот, изображение делится на мелкие составляющие и изучается как под микроскопом в поиске отдельных составляющих.

Каждый сверточный слой содержит несколько небольших фильтров-образов. Получая на входе изображение, сверточный слой раскладывает его на составные части. Каждый из составных элементов изображения проверяется на соответствие искомым образам. Такой подход призван выделить из общего изображения только составляющие искомого изображения, частично или полностью исключив влияние шума и фона на результат. Также это помогает решить проблему перемещения или масштабирования искомого образа на изображении.

Использование малых матриц весов для каждого фильтра позволяет значительно уменьшить требование к размеру памяти для их хранения. Более того, при таком подходе размер матрицы весов зависит не от размера исходного изображения, а от размера фильтра-образа. Следовательно, с ростом размера обрабатываемых изображений размер матрицы весов остается неизменным.

В дополнение к вышесказанному использование сверточных нейронных слоев позволяет с каждым слоем уменьшить размер обрабатываемых данных. Это связано с тем, что для каждой малой составной части изображения возвращается только одно значение, указывающее на меру соответствия изображения искомому образу.

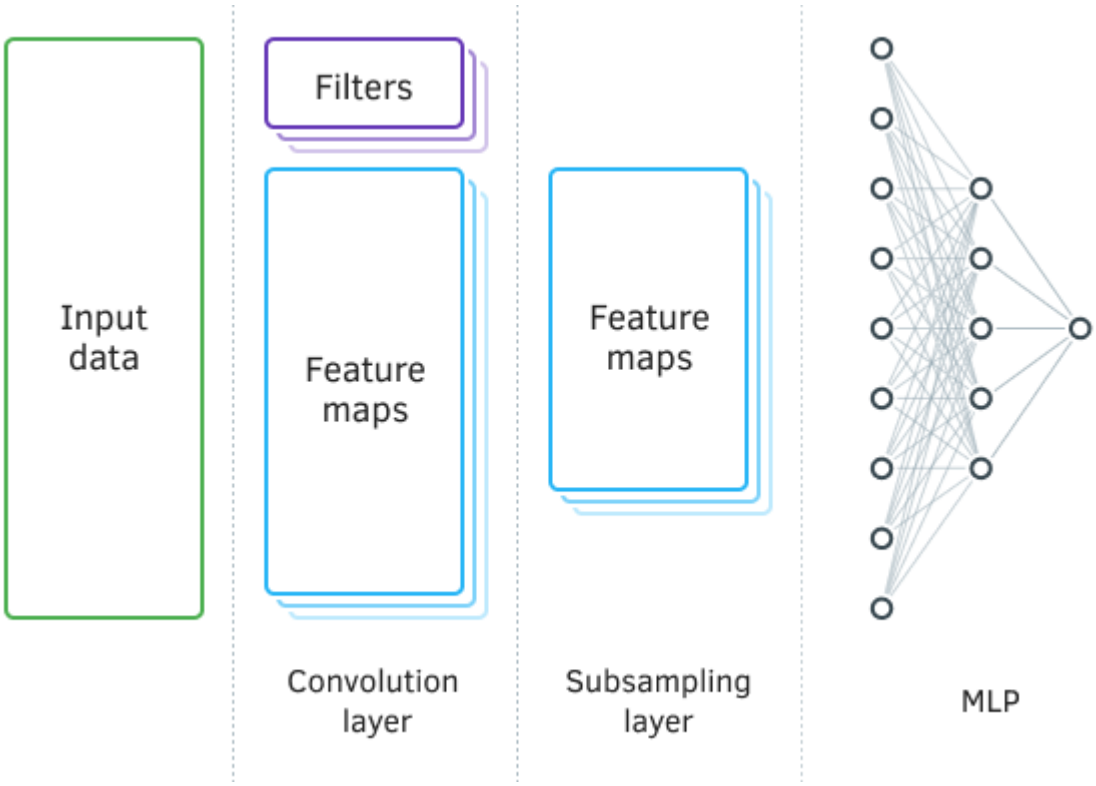
Если говорить применительно к трейдингу, я попытался перевести преимущества сверточных сетей в новую плоскость. Вместо поиска малых составных искомого образа на изображении мы будем искать в потоке исходных данных малые составные части паттернов из ценовых свечей и значений индикаторов. Тем самым, мы постараемся исключить влияние шумовых колебаний на общий результат.

И конечно, большим преимуществом сверточных нейронных сетей является тот факт, что фильтры подбираются нейронной сетью в процессе обучения.

Давайте погрузимся в архитектуры этого решения и познакомимся с ним по ближе.

4.1.1 Описание архитектуры и принципов реализации

В сверточных сетях, по сравнению с полносвязным перцептроном, добавляются два новых вида слоев: сверточный (фильтр) и подвыборочный (субдискретизирующий). Чередуясь, указанные слои призваны выделить основные компоненты и отсеять шумы в исходных данных с параллельным понижением размерности (объема) данных, которые в последующем передаются на вход полносвязного перцептрона для принятия решения. В зависимости от решаемых задач допускается последовательное использование нескольких групп из чередующихся сверточного и подвыборочного слоев.



Свёрточная нейронная сеть

За распознавание объектов в массиве исходных данных отвечает сверточный слой (Convolution layer). В данном слое осуществляются последовательные операции математической свертки исходных данных с небольшим шаблоном (фильтром), выступающими в качестве ядра свертки.

> Свертка — операция в функциональном анализе, которая при применении к двум функциям f и g возвращает третью функцию, соответствующую взаимокорреляционной функции f(x) и g(-x). Операцию свертки можно интерпретировать как «схожесть» одной функции с отраженной и сдвинутой копией другой.

Иными словами, сверточный слой осуществляет поиск шаблонного элемента во всей исходной выборке. При этом на каждой итерации шаблон сдвигается по массиву исходных данных с заданным шагом, который может быть от 1 до размера шаблона. Если величина шага смещения меньше размера шаблона, то такая свертка называется с перекрытием.

В результате операции свертки получаем массив признаков, показывающих схожесть исходных данных с искомым шаблоном на каждой итерации. Для нормализации данных используются функции активации. Размер полученного массива будет меньше массива исходных данных, количество таких массивов равно количеству шаблонов (фильтров).

$$Size_{Conv} = \frac{Size_{Input} - Size_{Filter}}{Step} + 1$$

Немаловажен для нас и тот факт, что сами шаблоны не задаются при проектировании нейронной сети, а подбираются в процессе обучения.

Следующий далее подвыборочный слой используется для снижения размерности массива признаков и фильтрации шумов. Применение данной итерации обусловлено предположением о том, что наличие сходства исходных данных с шаблоном первично, а точные координаты признака в массиве исходных данных не столь важны. Это позволяет решать проблему

масштабирования, так как допускает некую вариативность расстояния между искомыми объектами.

На данном этапе происходит уплотнение данных путем сохранения максимального или среднего значения в пределах заданного окна. Таким образом, сохраняется только одно значение для каждого окна данных. Операции осуществляются итерационно со смещением окна на заданный шаг при каждой новой итерации. Уплотнение данных выполняется отдельно для каждого массива признаков.

Довольно часто применяются подвыборочные слои с окном и шагом равным двум, что позволяет вдвое снизить размерность массива признаков. Но в практике допускается и использование большего размера окна. Кроме того, итерации уплотнения могут осуществляться как с перекрытием (величина шага меньше размера окна), так и без.

На выходе подвыборочного слоя получаем массивы признаков меньшей размерности.

В зависимости от сложности решаемых задач, после подвыборочного слоя возможно использование еще одной или нескольких групп из сверточного и подвыборочного слоев. Принципы их построения и функциональность соответствуют описанным выше принципам.

В общем случае после одной или нескольких групп «свертка + уплотнения» массивы полученных признаков по всем фильтрам собираются в единый вектор и подаются на вход многослойного перцептрона для принятия решения нейронной сетью.

Обучение сверточных нейронных сетей осуществляется уже известным методом обратного распространения ошибки. Данный метод относится к методам обучения с учителем и заключается в спуске градиента ошибки от выходного слоя нейронов через скрытые слои к входному слою нейронов с корректировкой весовых коэффициентов в сторону антиградиента.

Рассмотрим организацию обучение нейронов подвыборочного и сверточного слоев.

В подвыборочном слое градиент ошибки считается для каждого элемента в массиве признаков по аналогии с градиентами нейронов полносвязного перцептрона. Алгоритм передачи градиента на предыдущий слой зависит от применяемой операции уплотнения. Если берется только максимальное значение, то и весь градиент передается на нейрон с максимальным значением. Для остальных элементов в пределах окна уплотнения устанавливается нулевой градиент, так как при прямом проходе они не оказывали влияния на конечный результат. Если же используется операция усреднения в пределах окна, то и градиент равномерно распределяется на все элементы в пределах окна.

В операции уплотнения не используются весовые коэффициенты, следовательно, и в процессе обучения ничто не корректируется.

Немного сложнее операции при обучении нейронов сверточного слоя. Градиент ошибки рассчитывается для каждого элемента массива признаков и спускается к соответствующим нейронам предыдущего слоя. В основе процесса обучения сверточного слоя лежат операции свертки и обратной свертки.

Для передачи градиента ошибки от подвыборочного слоя к сверточному, сначала края массива градиентов ошибок, полученных от подвыборочного слоя, дополняются нулевыми элементами, а затем производится свертка полученного массива с ядром свертки, развернутым на 180°. На выходе получаем массив градиентов ошибок размером равным массиву входных данных, в котором индексы градиентов будут соответствовать индексу корреспондирующего нейрона предшествующего слоя.

Для получения дельт весовых коэффициентов осуществляется свертка матрицы входных значений с матрицей градиентов ошибок данного слоя, развернутой на 180°. На выходе получим массив дельт с размером равным ядру свертки. Полученные дельты нужно скорректировать на производную функции активации сверточного слоя и коэффициент обучения. После этого весовые коэффициенты ядра свертки изменяются на величину скорректированных дельт.

Наверное, звучит довольно сложно для понимания. Попробуем прояснить данные моменты при подробном рассмотрении кода.

4.1.2 Построение средствами MQL5

Как мы уже увидели в описании архитектуры сверточной сети, для ее построения нам предстоит создать два новых типа нейронных слоев: сверточный и подвыборочный. Первый отвечает за фильтрацию данных и выделение искомых данных, а второй — за выделение точек максимального соответствия фильтру и снижение размерности массива данных. Сверточный слой имеет матрицу весов, но она на много меньше матрицы весов полносвязного слоя из-за того, что осуществляется поиск небольшого паттерна. Подвыборочный слой и вовсе не имеет весовых коэффициентов. Такое снижение размерности матрицы весов позволяет снизить количество математических вычислений и тем самым повысить скорость обработки информации. При этом количество операций снижается как при прямом, так и обратном проходах. Поэтому значительно снижается и время на обучение нейронной сети. А способность алгоритма отсеивать шумы позволяет повысить качество работы нейронной сети.

Подвыборочный слой

Реализацию алгоритма начнем с построения подвыборочного слоя. Для этого создадим класс *CNeuronProof*. Мы уже раньше озвучивали идею, что для преемственности нейронных слоев все они будут наследоваться от одного базового класса. Придерживаясь данной концепции, новый нейронный слой будем наследовать от ранее созданного класса *CNeuronBase*. Наследование будет публичным. Поэтому все методы, не переопределенные внутри класса *CNeuronProof*, будут доступны из родительского класса.

Для покрытия дополнительных требований, вызванных особенностями алгоритма сверточных сетей, в новом классе добавим переменные для хранения дополнительной информации:

- *m_iWindow* — размер окна на входе нейронного слоя;
- *m_iStep* — размер шага входного окна;
- *m_iNeurons* — размер выхода одного фильтра;
- *m_iWindowOut* — количество фильтров;
- *m_eActivation* — функция активации.

Обратите внимание, что в отличие от базового класса *CNeuronBase* мы не использовали отдельный класс функции активации *CActivation*, а ввели новую переменную *m_eActivation*. Дело в том, что подвыборочный слой не использует функцию активации в ранее рассмотренном виде. Здесь ее функционал немного другой. Обычно результатом работы подвыборочного слоя является максимальное или среднеарифметическое значение анализируемого окна. Поэтому мы реализуем новый функционал внутри методов данного класса и создадим новое перечисление из двух элементов:

- *AF_AVERAGE_POOLING* — среднеарифметическое входного окна данных,
- *Af_MAX_POOLING* — максимальное значение входного окна данных.

При этом мы намеренно не будем вносить изменения в код базового класса относительно новых функций активации, так как они не будут использоваться в других архитектурах нейронных слоев.

```
//--- функции активации подвыборочного слоя
enum ENUM_PROOF
  {
   AF_MAX_POOLING,
   AF_AVERAGE_POOLING
  };
```

Еще одна особенность подвыборочного слоя заключается в отсутствии матрицы весов. Следовательно, он не будет участвовать в процессе обучения и обновления весовых коэффициентов. При этом мы даже можем удалить некоторые объекты для высвобождения памяти. В то же время подвыборочный слой нельзя полностью исключить из обратного прохода, так как он будет участвовать в распространении градиента ошибки. Чтобы не нагромождать методы диспетчерских классов излишними проверками и при этом исключить вызов ненужных методов родительского класса, мы ряд методов заменим «заглушками», которые будут возвращать значение, требуемое для нормальной работы целостного алгоритма нейронной сети.

- *CalcOutputGradient* — всегда возвращает *false*, т.к. не предполагается использования слоя в качестве выходного для нейронной сети.
- *CalcDeltaWeights* и *UpdateWeights* — всегда возвращают *true*. Отсутствие матрицы весов делает данные методы излишними, но для корректной работы всей модели необходим возврат положительного результата от методов.
- *GetWeights* и *GetDeltaWeights* — всегда возвращают *NULL*. Методы переопределены для предотвращения ошибки доступа к несуществующему объекту.

Добавим еще один метод для возврата числа элементов на выходе одного фильтра и получим нижеследующую структуру класса.

```
class CNeuronProof    :  public CNeuronBase
  {
protected:
   uint              m_iWindow;              //Размер окна на входе нейронного слоя
   uint              m_iStep;                //Размер шага входного окна
   uint              m_iNeurons;             //Размер выхода одного фильтра
   uint              m_iWindowOut;           //Количество фильтров
   ENUM_PROOF        m_eActivation;          //Функция активации
public:
                     CNeuronProof(void);
                    ~CNeuronProof(void) {};
   //---
   virtual bool      Init(const CLayerDescription *desc) override;
   virtual bool      FeedForward(CNeuronBase *prevLayer) override;
   virtual bool      CalcOutputGradient(CBufferType *target) override;
                                                         { return false;}
   virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer)  { return true; }
   virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                   VECTOR &Beta, VECTOR &Lambda) override
                                                         { return true; }
   //---
   virtual CBufferType     *GetWeights(void)       const {  return(NULL);     }
   virtual CBufferType     *GetDeltaWeights(void)  const {  return(NULL);     }
   virtual uint      GetNeurons(void)              const {  return m_iNeurons;}
   //--- Методы работы с файлами
   virtual bool      Save(const int file_handle) override;
   virtual bool      Load(const int file_handle) override;
   //--- Метод идентификации объекта
   virtual int       Type(void) override      const { return(defNeuronProof); }
  };
```

В конструкторе класса мы лишь инициализируем начальными значениями добавленные переменные.

```
CNeuronProof::CNeuronProof(void) :  m_eActivation(AF_MAX_POOLING),
                                    m_iWindow(2),
                                    m_iStep(1),
                                    m_iWindowOut(1),
                                    m_iNeurons(0)
  {
  }
```

Мы не добавляли новых объектов, а за удаление созданных в базовом классе отвечает деструктор базового класса. Следовательно, деструктор нашего класса останется пустым.

Рассматриваем дальше методы нового класса подвыборочного слоя *CNeuronProof*. Разберем метод инициализации нейронного слоя *Init*. В параметрах метод, как и метод родительского класса, получает объект описания слоя. В начале метода мы проверяем действительность полученного объекта, а также соответствие требуемого слоя и текущего класса нейронной сети.

```
bool CNeuronProof::Init(const CLayerDescription *description)
  {
//--- блок контролей
   if(!description || description.type != Type() ||
      description.count <= 0)
      return false;
```

После успешного прохождения первого контроля сохраним и проверим параметры создаваемого слоя:

- размер входного окна,
- шаг входного окна,
- количество фильтров,
- количество элементов на выходе одного фильтра.

Все указанные параметры должны быть ненулевыми положительными значениями.

```
//--- Сохраняем константы
   m_iWindow = description.window;
   m_iStep = description.step;
   m_iWindowOut = description.window_out;
   m_iNeurons = description.count;
   if(m_iWindow <= 0 || m_iStep <= 0 || m_iWindowOut <= 0 || m_iNeurons <= 0)
      return false;
```

Также проверим указанную функцию активации. Для подвыборочного слоя мы можем использовать только два варианта функции активации *AF_AVERAGE_POOLING* и *AF_MAX_POOLING*. В остальных случаях будем выходить из метода с результатом *false*.

```
//--- Проверка функции активации
   switch((ENUM_PROOF)description.activation)
     {
      case AF_AVERAGE_POOLING:
      case AF_MAX_POOLING:
         m_eActivation = (ENUM_PROOF)description.activation;
         break;
      default:
         return false;
         break;
     }
```

После успешного прохождения всех контрольных блоков приступаем непосредственно к инициализации нейронного слоя. Сначала инициализируем нулевыми значениями вектор результатов *m_cOutputs*. Данный буфер мы создадим в виде прямоугольной матрицы, строки которой будут представлять отдельные фильтры.

```
//--- Инициализируем буфер результатов
   if(!m_cOutputs)
      if(!(m_cOutputs = new CBufferType()))
         return false;
   if(!m_cOutputs.BufferInit(m_iWindowOut, m_iNeurons, 0))
      return false;
```

Использование матриц позволяет нам распределить данные по фильтрам в рамках одного объекта. Это дает нам возможность использовать понятную структуру данных и в одном буфере осуществлять обмен данными между *CPU* и контекстом *OpenCL*. Это позволит нам немного выиграть время при передаче данных и организовать параллельную обработку данных сразу всеми фильтрами.

Аналогичный подход используется и для буфера градиентов ошибки *m_cGradients*.

```
//--- Инициализируем буфер градиентов ошибки
   if(!m_cGradients)
      if(!(m_cGradients = new CBufferType()))
         return false;
   if(!m_cGradients.BufferInit(m_iWindowOut, m_iNeurons, 0))
      return false;
```

После завершения инициализации буферов результатов и градиентов удалим неиспользуемые объекты и выходим из метода с положительным результатом.

```
//---
   m_eOptimization = None;
//--- Удаляем неиспользуемые объекты
   if(!!m_cActivation)
      delete m_cActivation;
   if(!!m_cWeights)
      delete m_cWeights;
   if(!!m_cDeltaWeights)
      delete m_cDeltaWeights;
   for(int i = 0; i < 2; i++)
      if(!!m_cMomenum[i])
         delete m_cMomenum[i];
//---
   return true;
  }
```

Теперь, когда мы завершили инициализацию нейронного слоя, переходим к организации прямого прохода в методе *FeedForward*. Как и предыдущий метод, метод прямого прохода построен с соблюдением концепции наследования и переопределения виртуальных методов базового класса с добавлением нового функционала. В параметрах метод получает указатель на объект предыдущего нейронного слоя. Как всегда, в начале метода организуем контрольный блок проверки исходных данных. Здесь мы проверяем действительность указателей на предыдущий нейронный слой, на буферы результатов предыдущего и текущего нейронных слоев.

```
bool CNeuronProof::FeedForward(CNeuronBase *prevLayer)
  {
//--- Блок контролей
   if(!prevLayer || !m_cOutputs ||
      !prevLayer.GetOutputs())
      return false;
   CBufferType *input_data = prevLayer.GetOutputs();
```

После успешного прохождения контрольного блока сохраним указатель на буфер результатом предыдущего слоя и создадим разветвление алгоритма метода по используемому вычислительному устройству: *CPU* или контекст *OpenCL*. К алгоритму многопоточных вычислений мы вернемся немного позже, а сейчас рассмотри реализацию средствами *MQL5*.

Еще раз акцентируем внимание, что подвыборочный слой не имеет матрицы весов. И так же, как и все другие нейронные слои, использует одну функцию активации для всех нейронов и фильтров. А значит, различие между результатами работы фильтров возможны только при использовании различных исходных данных. Иными словами, количество фильтров подвыборочного слоя должно соответствовать количеству фильтров предшествующего сверточного слоя. Поэтому, мы сначала скопируем матрицу исходных данных и при необходимости переформатируем ее.

```
//---  Разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      MATRIX inputs = input_data.m_mMatrix;
      if(inputs.Rows() != m_iWindowOut)
        {
         ulong cols = (input_data.Total() + m_iWindowOut - 1) / m_iWindowOut;
         if(!inputs.Reshape(m_iWindowOut, cols))
            return false;
        }
```

Следует отметить, что несмотря на то, что подразумевается использование подвыборочного слоя после сверточного, алгоритмом нашего метода допускается его использование после базового класса полносвязного нейронного слоя. Именно поэтому мы копируем матрицу исходных данных. Это позволяет безболезненно переформатировать нам в нужный формат без боязни нарушить структуру предшествующего слоя.

Надо сказать, что MQL5 не поддерживает трехмерные матрицы. Поэтому дальше нам придется работать отдельно по каждому фильтру. Вначале мы создадим локальную матрицу с числом строк и столбцов равными размерам результатов одного фильтра и входного окна соответственно. Организуем два вложенных цикла: внешний цикл с числом итераций равным числу фильтров и внутренний с числом повторений равным числу элементов в одном фильтре текущего слоя.

```
//--- Создаем локальную матрицу для сбора данных одного фильтра
      MATRIX array = MATRIX::Zeros(m_iNeurons, m_iWindow);
      m_cOutputs.m_mMatrix.Fill(0);
//--- Цикл перебора фильтров
      for(uint f = 0; f < m_iWindowOut; f++)
        {
//--- Цикл перебора элементов буфера результатов
         for(uint o = 0; o < m_iNeurons; o++)
           {
            uint shift = o * m_iStep;
            for(uint i = 0; i < m_iWindow; i++)
               array[o, i] = ((shift + i) >= inputs.Cols() ? 0 :
                              inputs[f, shift + i]);
           }
```

Во внутреннем цикле мы организуем еще один вложенный цикл. В его теле мы распределим исходные данные одного фильтра в выше созданную матрицу в соответствии с размерам окна данных и его шага. Использование цикла вызвано унификацией подхода на случай, когда размер и шага не равны.

После распределения исходных данных мы воспользуемся матричными операциями в соответствии с заданной функцией активации. Полученный вектор сохраняем в матрицу результатов. Строка матрицы результатов соответствует номеру анализируемого фильтра.

```
//--- Сохраняем текущий результат в соответствии с функцией активации
         switch(m_eActivation)
           {
            case AF_MAX_POOLING:
               if(!m_cOutputs.Row(array.Max(1), f))
                  return false;;
               break;
            case AF_AVERAGE_POOLING:
               if(!m_cOutputs.Row(array.Mean(1), f))
                  return false;
               break;
            default:
               return false;
           }
        }
     }
```

Я использую термин фильтр, чтобы в вашем понимании прослеживалась цепочка: фильтр сверточного слоя переходит в фильтр подвыборочного слоя. Итерации работы подвыборочного слоя сложно назвать фильтром. В то же время я хочу, чтобы в вашем понимании четко отложилось: сверточный и подвыборочный слои хоть и организованы в два объекта нейронных слоев, но составляют единую целую конструкцию. Поэтому используется та же терминология.

После успешной отработки всех итераций системы циклов выходим из метода с результатом *true*.

```
   else
     {
//--- Блок многопоточных вычислений будет добавлен в следующей главе
      return false;
     }
//--- Успешное завершение метода
   return true;
  }
```

За прямым проходом идет обратный проход. Отсутствие матрицы весов в подвыборочном слое позволяет организовать обратный проход в одном методе, в отличие от базового класса нейронной сети *CNeuronBase*, в котором обратный проход разделен на несколько функциональных методов.

По существу, для подвыборочного слоя обратный проход вырождается в метод передачи градиента ошибки в скрытом слое *CalcHiddenGradient*. Остальные методы мы заменили заглушками, о чем уже было сказано выше.

Сам метод *CalcHiddenGradient* построен в рамках нашей концепции использования единого формата виртуальных методов для всех классов нейронных сетей с общим наследование от единого базового класса нейронного слоя. Поэтому, как и метод базового класса нейронного слоя *CNeuronBase::CalcHiddenGradient*, в параметрах метод получает указатель на объект предыдущего нейронного слоя. В начале метода организован контрольный блок проверки входящих данных. Здесь мы проверяем корректность полученного в параметрах указателя на объект предыдущего нейронного слоя и наличие в предыдущем слое действующих буферов результатов и градиентов ошибки. Также проверяем корректность буферов результатов и градиентов ошибки текущего слоя.

```
bool CNeuronProof::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
//--- Блок контролей
   if(!prevLayer || !m_cOutputs ||
      !m_cGradients || !prevLayer.GetOutputs() ||
      !prevLayer.GetGradients())
      return false;
   CBufferType *input_data = prevLayer.GetOutputs();
   CBufferType *input_gradient = prevLayer.GetGradients();
   if(!input_gradient.BufferInit(input_data.Rows(), input_data.Cols(), 0))
      return false;
```

После успешного прохождения блока контролей мы, по аналогии с методом прямого прохода, скопируем и переформатируем матрицу исходных данных. А также создадим аналогичного размера нулевую локальную матрицу для накопления градиентов ошибки.

Обратите внимание, что в базовом классе нейронного слоя мы не проводили предварительное обнуление буфера градиентов. Дело в разнице подходов к передаче градиентов ошибки на предыдущий слой. В алгоритме базового класса заложен пересчет и сохранение значения градиента для каждого элемента. При таком подходе предварительное обнуление буфера не имеет смысла, т.к. любое значение будет перезаписано новым значением. В алгоритме же подвыборочного слоя запись градиента ошибки в каждый элемент буфера предыдущего слоя предусматривается только при использовании *Average Pooling* (среднеарифметического значения). В случае же *Max Pooling* (максимального значения) градиент ошибки передается только элементу с максимальным значением, т.к. только он влияет на последующий результат

работы нейронной сети. Остальные элементы получают нулевой градиент ошибки. Поэтому, мы сразу обнуляем весь буфер и вписываем значение градиента только для элементов, оказывающих влияние на результат.

Далее следует разделение алгоритма в зависимости от вычислительного устройства. Мы сейчас не будем обсуждать реализацию многопоточных вычислений в OpenCL, а сосредоточимся на реализации средствами MQL5.

Здесь мы, как и при прямом проходе, организуем систему вложенных циклов с перебором фильтров и их элементов. Внутри циклов осуществляется распределение градиента ошибки на элементы предыдущего слоя в зависимости от функции активации.

```
//---  Разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      MATRIX inputs = input_data.m_mMatrix;
      ulong cols = (input_data.Total() + m_iWindowOut - 1) / m_iWindowOut;
      if(inputs.Rows() != m_iWindowOut)
        {
         if(!inputs.Reshape(m_iWindowOut, cols))
            return false;
        }
//--- Создаем локальную матрицу для сбора данных одного фильтра
      MATRIX inputs_grad = MATRIX::Zeros(m_iWindowOut, cols);
```

```
//--- Цикл перебора фильтров
      for(uint f = 0; f < m_iWindowOut; f++)
        {
//--- Цикл перебора элементов буфера результатов
         for(uint o = 0; o < m_iNeurons; o++)
           {
            uint shift = o * m_iStep;
            TYPE out = m_cOutputs.m_mMatrix[f, o];
            TYPE gradient = m_cGradients.m_mMatrix[f, o];
 //--- Передача градиента в соответствии с функцией активации
            switch(m_eActivation)
              {
               case AF_MAX_POOLING:
                  for(uint i = 0; i < m_iWindow; i++)
                    {
                     if((shift + i) >= cols)
                        break;
                     if(inputs[f, shift + i] == out)
                       {
                        inputs_grad[f, shift + i] += gradient;
                        break;
                       }
                    }
                  break;
               case AF_AVERAGE_POOLING:
                  gradient /= (TYPE)m_iWindow;
                  for(uint i = 0; i < m_iWindow; i++)
                    {
                     if((shift + i) >= cols)
                        break;
                     inputs_grad[f, shift + i] += gradient;
                    }
                  break;
               default:
                  return false;
              }
           }
        }
//--- копирование матрицы градиентов в буфер предыдущего нейронного слоя
      if(!inputs_grad.Reshape(input_gradient.Rows(), input_gradient.Cols()))
         return false;
      input_gradient.m_mMatrix = inputs_grad;
     }
```

При использовании среднеарифметического значения (*AF_AVERAGE_POOLING*) градиент ошибки в равной мере передается на все элементы входного окна данных для соответствующего элемента результатов.

При использовании максимального значения (*AF_MAX_POOLING*) весь градиент ошибки передается на элемент с максимальным значением. Причем при наличии нескольких элементов с одинаковым значением, ставшим максимальным, градиент ошибки передается на элемент с минимальным индексом в буфере результатов предыдущего слоя. Такой вариант выбран

осознанно с целью повысить общую эффективность нейронной сети. Причина тому заключается в том, что при передаче одинакового градиента на элементы с одинаковым значением мы рискуем попасть в ситуацию, при которой два и более нейронов будут работать, синхронно выдавая одинаковые результаты. Дублирование сигнала разными нейронами не повышает значимость сигнала, а только снижает эффективность работы нейронной сети. Ведь работая синхронно эффективность таких нейронов становится равной работе одного нейрона. Поэтому при передаче градиента ошибки только одному нейрону мы надеемся, что в следующий раз другой элемент получит другое значение градиента и нарушит синхронность работы нейронов.

После заполнения локальной матрицы градиентов мы перенесем полученный результат в буфер градиентов предыдущего слоя и выходим из метода с результатом проведения операций.

```
    else
      {
//--- Блок многопоточных вычислений будет добавлен в следующей главе
       return false;
      }
//--- Успешное завершение метода
   return true;
  }
```

Рассмотренные выше методы описывают основной функционал подвыборочного слоя. Для полноты функционала класса необходимо добавить методы работы с файлами для записи информации об обученной нейронной сети в файл. Основная особенность подвыборочного слоя заключается в отсутствии матрицы весов. Следовательно, в нем нет обучаемых элементов и нет необходимости сохранять какие-либо буферы данных. Для полного восстановления функционала слоя нам достаточно сохранить значения его переменных, определяющих параметры работы класса.

- *m_iWindow* — размер окна на входе нейронного слоя;
- *m_iStep* — размер шага входного окна;
- *m_iNeurons* — размер выхода одного фильтра;
- *m_iWindowOut* — количество фильтров;
- *m_eActivation* — функция активации.

```
bool CNeuronProof::Save(const int file_handle)
  {
//--- Блок контролей
   if(file_handle == INVALID_HANDLE)
      return false;
//--- Сохраняем константы
   if(FileWriteInteger(file_handle, Type()) <= 0)
      return false;
   if(FileWriteInteger(file_handle, (int)m_iWindow) <= 0)
      return false;
   if(FileWriteInteger(file_handle, (int)m_iStep) <= 0)
      return false;
   if(FileWriteInteger(file_handle, (int)m_iWindowOut) <= 0)
      return false;
   if(FileWriteInteger(file_handle, (int)m_iNeurons) <= 0)
      return false;
   if(FileWriteInteger(file_handle, (int)m_eActivation) <= 0)
      return false;
//--- Успешное завершение метода
   return true;
  }
```

Метод восстановления слоя из файла немного сложнее метода сохранения. В данном случае мне кажется более удачным именно определение «восстановление», а не «загрузка». Это связано с тем, что мы не будем считывать из файла какую-либо информацию об обучении и наработках метода. Из файла мы сначала считаем параметры слоя, которые содержат примерно столько информации, сколько передаем в метод инициализации в объекте описания слоя. Затем инициализируем буферы результатов и градиентов ошибки.

```
bool CNeuronProof::Load(const int file_handle)
  {
//--- Блок контролей
   if(file_handle == INVALID_HANDLE)
      return false;
//--- Загружаем константы
   m_iWindow = (uint)FileReadInteger(file_handle);
   m_iStep = (uint)FileReadInteger(file_handle);
   m_iWindowOut = (uint)FileReadInteger(file_handle);
   m_iNeurons = (uint)FileReadInteger(file_handle);
   m_eActivation = (ENUM_PROOF)FileReadInteger(file_handle);
//--- Инициализируем буфер результатов
   if(!m_cOutputs)
     {
      m_cOutputs = new CBufferType();
      if(!m_cOutputs)
         return false;
     }
   if(!m_cOutputs.BufferInit(m_iWindowOut, m_iNeurons, 0))
      return false;
//--- Инициализируем буфер градиентов ошибки
   if(!m_cGradients)
     {
      m_cGradients = new CBufferType();
      if(!m_cGradients)
         return false;
     }
   if(!m_cGradients.BufferInit(m_iWindowOut, m_iNeurons, 0))
      return false;
//---
   return true;
  }
```

На этом этапе можно сказать, что мы завершили первую часть работы по построению объектов сверточной нейронной сети. Переходим ко второму этапу — построению класса сверточного слоя.

Сверточный слой

Построение сверточного слоя осуществляется в классе *CNeuronConv*, который мы будем наследовать от созданного выше класса подвыборочного слоя *CNeuronProof*. Наследование от класса подвыборочного слоя никак не нарушает нашу концепцию общего наследования всех классов нашей нейронной сети от общего базового класса. Класс подвыборочного слоя является прямым наследником базового класса, и все его наследники так же будут наследниками базового класса.

В то же время, наследуясь от класса подвыборочного слоя, мы сразу получаем весь добавленный в него и переопределенный функционал, включая переменные для работы с окнами данных. При этом наследование объектов и переменных подтверждает и подчеркивает связь классов и единство подходов в обработке данных.

Таким образом, благодаря наследованию в классе сверточного слоя *CNeuronConv* мы будем использовать объекты и переменные, объявленные в родительских классах. Нам нет необходимости объявлять какие-либо новые объекты и переменные. Как следствие, конструктор и деструктор нашего класса остаются пустыми методами. В то же время в классе сверточного слоя используется матрица весов. При этом нам нужно будет переопределить некоторые установленные ранее заглушки.

- *UpdateWeights* — полностью удовлетворяет алгоритм метода базового класса *CNeuronBase*, поэтому вызовем его выполнение.
- *GetWeights* и *GetDeltaWeights* — вернем указатели на соответствующие буферы данных.

В результате структура класса примет нижеследующий вид.

```
class CNeuronConv    :  public CNeuronProof
  {
public:
                     CNeuronConv(void) {};
                    ~CNeuronConv(void) {};
   //---
   virtual bool      Init(const CLayerDescription *desc) override;
   virtual bool      FeedForward(CNeuronBase *prevLayer);
   virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer);
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer);
   virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                   VECTOR &Beta, VECTOR &Lambda)
     {
      return CNeuronBase::UpdateWeights(batch_size, learningRate,
                                        Beta, Lambda);
     }
   //---
   virtual CBufferType*  GetWeights(void)      const { return(m_cWeights);     }
   virtual CBufferType*  GetDeltaWeights(void) const { return(m_cDeltaWeights);}
   bool              SetTransposedOutput(const bool value);
   //--- методы работы с файлами
   virtual bool      Save(const int file_handle);
   virtual bool      Load(const int file_handle);
   //--- метод идентификации объекта
   virtual int       Type(void)       const { return(defNeuronConv); }
  };
```

Посмотрим на организацию метода инициализации сверточного слоя *Init*. Он отчасти объединил в себе методы инициализации обоих родительских классов. Но к сожалению, мы не можем использовать ни один из них: в методе инициализации базового класса будут созданы буферы неверного размера и их все равно придется переопределять, а в методе инициализации подвыборочного слоя удаляются объекты, которые потом нужно будет повторно создавать. Поэтому запишем в метод весь алгоритм целиком.

Как и аналогичные методы родительских классов, в параметрах метод инициализации получает указатель на объект описания создаваемого нейронного слоя. Как и ранее, метод начинается с контрольного блока, в котором проверяем действительность полученного указателя, указанный тип создаваемого слоя и параметры слоя.

```
bool CNeuronConv::Init(const CLayerDescription *desc)
  {
//--- блок контролей
   if(!desc || desc.type != Type() || desc.count <= 0 || desc.window <= 0)
      return false;
```

После выполнения блока контролей сохраняем параметры слоя в специальные переменные и инициализируем необходимые буферы.

```
//--- сохраняем константы
   m_iWindow = desc.window;
   m_iStep = desc.step;
   m_iWindowOut = desc.window_out;
   m_iNeurons = desc.count;
//--- сохраняем метод оптимизации параметров
   m_eOptimization = desc.optimization;
```

Сначала инициализируем буфер результатов *m_cOutputs*. Количество строк и столбцов матрицы буфера по аналогии с подвыборочным слоем устанавливаем равным количеству фильтров и количеству элементов в одном фильтре соответственно. Буфер инициализируется нулевыми значениями.

Следом инициализируем нулевыми значениями буфер градиентов ошибки *m_cGradients*. Его размер устанавливаем равным размеру буфера результатов *m_cOutputs*.

```
//--- инициализируем буфер результатов
   if(!m_cOutputs)
      if(!(m_cOutputs = new CBufferType()))
         return false;
//--- инициализируем буфер градиентов ошибки
   if(!m_cGradients)
      if(!(m_cGradients = new CBufferType()))
         return false;
   if(!m_cOutputs.BufferInit(m_iWindowOut, m_iNeurons, 0))
      return false;
   if(!m_cGradients.BufferInit(m_iWindowOut, m_iNeurons, 0))
      return false;
```

Далее нам предстоит инициализировать экземпляр объекта функции активации. Как вы помните, при разработке базового класса нейронного слоя мы решили вынести всю работу по инициализации экземпляра объекта функции активации в отдельный метод *SetActivation*. Здесь мы лишь вызываем данный метод родительского класса и проверяем результат выполнения операций.

```
//--- инициализируем класс функции активации
   VECTOR params=desc.activation_params;
   if(!SetActivation(desc.activation, params))
      return false;
```

Затем инициализируем матрицу весов случайными значениями. Количество строк в матрице весов равно количеству используемых фильтров, а количество столбцов в матрице на единицу больше размера анализируемого окна. Добавленный элемент используется для *bias*-смещения. Инициализация матрицы осуществляется случайными значениями.

```
//--- инициализируем буфер матрицы весов
   if(!m_cWeights)
      if(!(m_cWeights = new CBufferType()))
         return false;
   if(!m_cWeights.BufferInit(desc.window_out, desc.window + 1))
      return false;
   double weights[];
   double sigma = desc.activation == AF_LRELU ?
                  2.0 / (double)(MathPow(1 + desc.activation_params[0], 2) *
                                                                  desc.window) :
                  1.0 / (double)desc.window;
   if(!MathRandomNormal(0, MathSqrt(sigma), m_cWeights.Total(), weights))
      return false;
   for(uint i = 0; i < m_cWeights.Total(); i++)
      if(!m_cWeights.m_mMatrix.Flat(i, (TYPE)weights[i]))
         return false;
```

И в заключение метода инициализируем буферы, участвующие в процессе обучения. Это буфер дельт весовых коэффициентов (он же буфер накопленных градиентов) и буферы моментов. Напомню, что количество используемых буферов моментов зависит от указанного пользователем метода оптимизации параметров модели. Размеры указанных буферов будут соответствовать размеру матрицы весов.

```
//--- инициализируем буфер градиентов на уровне матрицы весов
   if(!m_cDeltaWeights)
      if(!(m_cDeltaWeights = new CBufferType()))
         return false;
   if(!m_cDeltaWeights.BufferInit(desc.window_out, desc.window + 1, 0))
      return false;
//--- инициализируем буферы моментов
   switch(desc.optimization)
     {
      case None:
      case SGD:
         for(int i = 0; i < 2; i++)
            if(m_cMomenum[i])
               delete m_cMomenum[i];
         break;
      case MOMENTUM:
      case AdaGrad:
      case RMSProp:
         if(!m_cMomenum[0])
            if(!(m_cMomenum[0] = new CBufferType()))
               return false;
         if(!m_cMomenum[0].BufferInit(desc.window_out, desc.window + 1, 0))
            return false;
         if(m_cMomenum[1])
            delete m_cMomenum[1];
         break;
      case AdaDelta:
      case Adam:
         for(int i = 0; i < 2; i++)
           {
            if(!m_cMomenum[i])
               if(!(m_cMomenum[i] = new CBufferType()))
                  return false;
            if(!m_cMomenum[i].BufferInit(desc.window_out, desc.window + 1, 0))
               return false;
           }
         break;
      default:
         return false;
         break;
     }
   return true;
  }
```

После инициализации класса перейдем к методу прямого прохода, который создадим в переопределенном виртуальном методе *FeedForward*. Тем самым мы продолжаем эксплуатировать концепции наследования и виртуализации методов классов. В параметрах метод прямого прохода получает указатель на объект предыдущего слоя, впрочем, как и все аналогичные методы родительских классов.

В начале метода, как обычно, вставим контрольный блок проверки исходных данных. В нем мы проверяем действительность полученного указателя на объект предшествовавшего нейронного

слоя и наличие в нем «живого» буфера результатов. Также проверим, созданы ли буфер результатов и матрица весов текущего слоя. Для упрощения процедуры доступа к данным буфера результатов предшествовавшего слоя сохраним в локальную переменную указатель на данный объект.

```
bool CNeuronConv::FeedForward(CNeuronBase *prevLayer)
  {
//--- блок контролей
   if(!prevLayer || !m_cOutputs || !m_cWeights || !prevLayer.GetOutputs())
      return false;
   CBufferType *input_data = prevLayer.GetOutputs();
   ulong total = input_data.Total();
```

Дальше идет разделение алгоритма на два потока в зависимости от устройства выполнения операций. Алгоритм построения многопоточных вычислений с использованием технологии *OpenCL* мы разберем в следующей главе. Сейчас же рассмотрим алгоритм выстраивания операций средствами *MQL5*.

Алгоритм прямого прохода сверточного слоя отчасти будет напоминать аналогичный метод подвыборочного слоя. Это вполне понятно: оба слоя работают с окном данных, которое с заданным шагом перемещается по массиву исходных данных. Отличия существуют в методах обработки набора значений, попадающих в окно.

Еще одно отличие заключается в подходе к восприятию массива исходных данных. Подвыборочный слой в алгоритме сверточной нейронной сети стоит после сверточного слоя, который может содержать несколько фильтров. Соответственно, в буфере результатов будут результаты обработки данных несколькими фильтрами. Подвыборочный слой должен отделить результаты одного фильтра от другого. В сверточном слое я решил упростить этот момент, поэтому рассматриваю весь входящий массив как единый вектор исходных данных. Такой подход позволяет упростить алгоритм метода без потерь качества работы нейронной сети в целом.

Вернемся к алгоритму. Перед тем как воспользоваться матричными операциями, нам предстоит преобразовать вектор исходных данных в матрицу с количеством строк, равным количеству элементов в одном фильтре. Количество столбцов должно соответствовать размеру анализируемого окна исходных данных. Здесь возможно два варианта: размер анализируемого окна равен его шагу и нет.

В первом случае мы можем просто переформатировать вектор в матрицу. Во втором нам потребуется создать систему циклов для копирования данных.

```
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
      MATRIX m;
      if(m_iWindow == m_iStep && total == (m_iNeurons * m_iWindow))
        {
         m = input_data.m_mMatrix;
         if(!m.Reshape(m_iNeurons, m_iWindow))
            return false;
        }
      else
        {
         if(!m.Init(m_iNeurons, m_iWindow))
            return false;
         for(ulong r = 0; r < m_iNeurons; r++)
           {
            ulong shift = r * m_iStep;
            for(ulong c = 0; c < m_iWindow; c++)
              {
               ulong k = shift + c;
               m[r, c] = (k < total ? input_data.At((uint)k) : 0);
              }
           }
        }
```

Затем к полученной матрице добавим единичный столбец *bias*-смешения. Результирующую матрицу умножим на транспонированную матрицу весов.

```
//--- добавляем bias-столбец
      if(!m.Resize(m.Rows(), m_iWindow + 1) ||
         !m.Col(VECTOR::Ones(m_iNeurons), m_iWindow))
         return false;
//--- Вычисление взвешенной суммы элементов входного окна
      m_cOutputs.m_mMatrix = m_cWeights.m_mMatrix.MatMul(m.Transpose());
     }
```

В завершение вызовем метод *Activation* класса функции активации и завершим работу метода.

```
   else
     {
//--- Блок многопоточных вычислений будет добавлен в следующей главе
      return false;
     }
   if(!m_cActivation.Activation(m_cOutputs))
      return false;
//--- Успешное завершение метода
   return true;
  }
```

После завершения работы над прямым проходом перейдем к работе над обратным проходом. В отличие от подвыборочного слоя, сверточный слой содержит матрицу весов. Поэтому для организации обратного прохода нам потребуется полный набор методов.

Немного забегая вперед, скажу, что метод обновления матрицы весов нам полностью подходит от базового класса. Но так как мы наследовались не напрямую от класса *CNeuronBase*, а от подвыборочного слоя *CNeuronProof*, в котором метод заменен заглушкой, то нам придется принудительно обратиться к методу базового класса.

```
bool CNeuronConv::UpdateWeights(int batch_size, TYPE learningRate,
                                VECTOR &Beta, VECTOR &Lambda)
      {
       return CNeuronBase::UpdateWeights(batch_size, learningRate, Beta, Lambda);
      }
```

Но вернемся к логической цепочке алгоритма обратного распространения и посмотрим на метод распределения градиента через скрытый слой *CNeuronConv::CalcHiddenGradient*.

Если посмотреть влияние элементов исходных данных на элементы результатов, то можно заметить зависимость. Каждый элемент вектора результатов анализирует блок данных из вектора исходных данных в размере заданного окна. Так же и каждый элемент исходных данных оказывает влияние на значение элементов вектора результатов в пределах некоего окна влияния. Размер этого окна зависит от шага, с которым перемещается входное окно по массиву исходных данных. При шаге равном единице оба окна равны. Но с ростом шага уменьшается размер окна влияния. Следовательно, для распространения градиента ошибки нам необходимо собрать градиенты ошибки с элементов последующего слоя в пределах окна влияния.

Предлагаю посмотреть на практическую реализацию данного метода. Мы продолжаем работать с виртуальными методами родительских классов. В параметрах метод получает указатель на объект предыдущего слоя. По аналогии с другими методами в начале метода делаем блок контроля входящих данных. Здесь мы проверяем действительность полученного в параметрах указателя и наличие действительных объектов буферов выходных значений и градиентов ошибки предыдущего слоя. Также проверим наличие буфера градиентов ошибки и матрицы весов текущего слоя.

```
bool CNeuronConv::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
//--- блок контролей
   if(!prevLayer || !prevLayer.GetOutputs() || !prevLayer.GetGradients() ||
      !m_cGradients || !m_cWeights)
      return false;
```

После успешного прохождения блока контролей скорректируем градиент ошибки на производную функции активации текущего слоя.

```
//--- корректировка градиентов ошибки на производную функции активации
   if(m_cActivation)
     {
      if(!m_cActivation.Derivative(m_cGradients))
         return false;
     }
```

Далее идет разветвление алгоритма в зависимости от используемого вычислительного устройства. Сейчас мы рассматриваем ветку MQL5.

Метод обратного прохода зеркален методу прямого прохода. Если при прямом проходе мы сначала переносили исходные данные в локальную матрицу и затем умножали на матрицу весов, то при

выполнении обратного прохода мы сначала переформатируем матрицу градиента ошибки, полученного от предыдущего слоя, в необходимый формат и умножим на матрицу весов.

```
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   CBufferType* input_gradient = prevLayer.GetGradients();
   if(!m_cOpenCL)
     {
      MATRIX g = m_cGradients.m_mMatrix;
      if(!g.Reshape(m_iWindowOut, m_iNeurons))
         return false;
      g = g.Transpose();
      g = g.MatMul(m_cWeights.m_mMatrix);
      if(!g.Resize(m_iNeurons, m_iWindow))
         return false;
```

В результате умножения матриц мы получим матрицу градиентов на предыдущий слой. Но процесс усложняется наличием анализируемого окна и его шага. При их равенстве нам достаточно просто переформатировать матрицу и скопировать ее значение в буфер предыдущего слоя. Если же размер анализируемого окна исходных данных не равен его шагу, то нам потребуется организовать систему циклов для копирования и суммирования градиентов. Ведь в таком случае один нейрон исходных данных оказывает влияние на несколько нейронов результатов каждого фильтра.

```
      if(m_iWindow == m_iStep && input_gradient.Total() == (m_iNeurons * m_iWindow))
        {
         if(!g.Reshape(input_gradient.Rows(), input_gradient.Cols()))
            return false;
         input_gradient.m_mMatrix = g;
        }
      else
        {
         input_gradient.m_mMatrix.Fill(0);
         ulong total = input_gradient.Total();
         for(ulong r = 0; r < m_iNeurons; r++)
           {
            ulong shift = r * m_iStep;
            for(ulong c = 0; c < m_iWindow; c++)
              {
               ulong k = shift + c;
               if(k >= total)
                  break;
               if(!input_gradient.m_mMatrix.Flat(k,
                                 input_gradient.m_mMatrix.Flat(k) + g[r, c]))
                  return false;
              }
           }
        }
   }
```

После завершения итераций циклов выходим из метода с положительным результатом.

```
    else
      {
//--- Блок многопоточных вычислений будет добавлен в следующей главе
       return false;
      }
//--- Успешное завершение метода
   return true;
  }
```

После распределения градиента через скрытый слой пришло время посчитать градиент ошибки на элементах матрицы весов. Ведь именно весовые коэффициенты мы будем подбирать для оптимальной работы нейронной сети. Вся работа по распространению градиента ошибки нужна лишь для того, чтобы получить направление и силу корректировки весовых коэффициентов. Такой подход делает работу по подбору оптимальной матрицы весов направленной и управляемой.

Работа по распределению градиента ошибки по элементам матрицы весов осуществляется в методе *CalcDeltaWeights*. Данный метод у нас также является виртуальным и переопределяется в каждом классе. В параметрах метод получает указатель на объект предыдущего слоя. И сразу в начале метода проверяем корректность полученного указателя и наличие рабочих буферов данных в текущем и предыдущем нейронных слоях. Для расчета градиента на матрице весов нам потребуется буфер входящего градиента, буфер исходных данных (результатов предыдущего слоя) и буфер для записи полученных результатов (*m_cDeltaWeights*). Напомню, что нашим алгоритмом предусмотрено распределение градиентов на каждой итерации обратного прохода, а обновление матрицы — по запросу из внешней программы. Поэтому в буфере *m_cDeltaWeights* мы будем накапливать значение градиента ошибки. А при обновлении разделим накопленное значение на количество совершенных итераций. Тем самым получим среднюю ошибку по каждому весовому коэффициенту.

```
bool CNeuronConv::CalcDeltaWeights(CNeuronBase *prevLayer)
  {
//--- блок контролей
   if(!prevLayer || !prevLayer.GetOutputs() || !m_cGradients || !m_cDeltaWeights)
      return false;
```

Для упрощения доступа к буферу данных предыдущего слоя сохраним указатель на объект в локальную переменную.

И далее следует разделение алгоритма на два логических потока операций в зависимости от используемого вычислительного устройства.

```
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   CBufferType *input_data = prevLayer.GetOutputs();
   if(!m_cOpenCL)
     {
```

Реализацию алгоритма в OpenCL мы рассмотрим в следующей главе, а сейчас остановимся на реализации средствами MQL5. Мы имеем двумерную матрицу весов, в которой одно измерение олицетворяет фильтры нашего слоя. Каждая строка в матрице весов — это отдельный фильтр. Следовательно, количество строк в матрице весов равно количеству используемых фильтров. Второе измерение (столбцы) матрицы представляют собой элементы нашего фильтра, и их число равно размеру входного окна плюс *bias*.

В то же время, так как окно фильтра перемещается по массиву исходных данных, то и каждый элемент фильтра оказывает влияние на результат всех элементов вектора результатов текущего слоя. Следовательно, для каждого элемента фильтра нам нужно собрать градиенты ошибки со всех элементов вектора результатов, которые хранятся в буфере *m_cGradients*. И в этом нам помогут векторные операции. Но вначале напомню, что в прямом проходе мы преобразовывали вектор исходных данных. Повторим этот процесс.

```
   MATRIX inp;
   uint input_total = input_data.Total();
   if(m_iWindow == m_iStep && input_total == (m_iNeurons * m_iWindow))
     {
      inp = input_data.m_mMatrix;
      if(!inp.Reshape(m_iNeurons, m_iWindow))
         return false;
     }
   else
     {
      if(!inp.Init(m_iNeurons, m_iWindow))
         return false;
      for(ulong r = 0; r < m_iNeurons; r++)
        {
         ulong shift = r * m_iStep;
         for(ulong c = 0; c < m_iWindow; c++)
           {
            ulong k = shift + c;
            inp[r, c] = (k < input_total ? input_data.At((uint)k) : 0);
           }
        }
     }
   //--- добавляем bias-столбец
   if(!inp.Resize(inp.Rows(), m_iWindow + 1) ||
      !inp.Col(VECTOR::Ones(m_iNeurons), m_iWindow))
      return false;
```

Далее мы будем непосредственно собирать градиенты ошибок для элементов фильтра. Как и в полносвязном слое градиент ошибки весового коэффициента равен произведению градиента ошибки нейрона на значение соответствующего элемента исходных данных. В разрезе матричных операций нам достаточно умножить матрицу градиентов перед функцией активации на переформатированную выше матрицу исходных данных.

```
   MATRIX g = m_cGradients.m_mMatrix;
   if(!g.Reshape(m_iWindowOut, m_iNeurons))
      return false;
   m_cDeltaWeights.m_mMatrix += g.MatMul(inp);
  }
```

Полученный результат прибавим к накопленным ранее градиентам ошибки в матрице *m_cDeltaWeights*.

```
   else
     {
//--- Блок многопоточных вычислений будет добавлен в следующей главе
      return false;
     }
//--- Успешное завершение метода
   return true;
  }
```

С алгоритмом реализации многопоточных вычислений мы познакомимся в следующей главе, а на данном этапе мы выходим из метода с положительным результатом.

О методе обновления весовых коэффициентов мы уже поговорили выше. Нам остается создать методы работы с файлами, ведь у нас должна быть возможность загрузить и использовать однажды обученную нейронную сеть. И здесь мы тоже воспользуемся ранее созданными наработками. Мы уже создали аналогичные методы для двух родительских классов: базового класса нейронного слоя *CNeuronBase* и подвыборочного слоя *CNeuronProof*. Методы подвыборочного слоя сильно упрощены, так как он не содержит матрицу весов и объектов для ее обучения. Поэтому воспользуемся методом базового класса и вызовем его принудительно из метода *CNeuronConv::Save*. Такой подход поможет нам исключить излишние контроли, так как они уже реализованы в методе родительского класса. Нам лишь остается проверить результат работы метода. Но нам его недостаточно, так как в подвыборочном слое введены новые переменные. Поэтому после выполнения метода родительского класса добавим в файл недостающие параметры.

```
bool CNeuronConv::Save(const int file_handle)
  {
//--- вызов метода родительского класса
   if(!CNeuronBase::Save(file_handle))
      return false;
//--- сохранение значений констант
   if(FileWriteInteger(file_handle, (int)m_iWindow) <= 0)
      return false;
   if(FileWriteInteger(file_handle, (int)m_iStep) <= 0)
      return false;
   if(FileWriteInteger(file_handle, (int)m_iWindowOut) <= 0)
      return false;
   if(FileWriteInteger(file_handle, (int)m_iNeurons) <= 0)
      return false;
   if(FileWriteInteger(file_handle, (int)m_bTransposedOutput) <= 0)
      return false;
//---
   return true;
  }
```

Организация загрузки данных построена по тому же принципу. Прежде всего, нам нужно считать данные из файла в том же порядке, в котором они были туда записаны. Следовательно, сначала вызовем метод родительского класса. В нем уже реализованы все контроли и соблюдена последовательность загрузки данных. Нам лишь остается проверить результат, возвращенный методом родительского класса, и после успешного его выполнения считать из файла дополнительные параметры в той же последовательности, в которой они были записаны.

```
bool CNeuronConv::Load(const int file_handle)
  {
//--- вызов метода родительского класса
   if(!CNeuronBase::Load(file_handle))
      return false;
//--- считывание значений констант
   m_iWindow = (uint)FileReadInteger(file_handle);
   m_iStep = (uint)FileReadInteger(file_handle);
   m_iWindowOut = (uint)FileReadInteger(file_handle);
   m_iNeurons = (uint)FileReadInteger(file_handle);
   m_eActivation = -1;
//---
   if(!m_cOutputs.Reshape(m_iWindowOut, m_iNeurons))
      return false;
   if(!m_cGradients.Reshape(m_iWindowOut, m_iNeurons))
      return false;
//---
   return true;
  }
```

В данном разделе мы создали два новых типа нейронных слоев: подвыборочный и сверточный. В следующем разделе мы еще дополним их функционал возможностью использования технологии *OpenCL* для организации параллельных вычислений с использованием многопоточных технологий. Затем в блоке сравнительного тестирования соберем небольшую нейронную сеть и сравним работу нового архитектурного решения с полученными ранее результатами тестирования полносвязных нейронных сетей.

4.1.3 Организация параллельных вычислений в сверточных сетях средствами OpenCL

В предыдущем разделе мы уже создали классы двух новых типов нейронных слоев. Это сверточный и подвыборочный слои. Данные типы слоев являются ключевыми в архитектуре сверточных нейронных сетей. Чередуя сверточный и подвыборочный слои, мы можем создать модель, которая будет искать в массиве исходных данных ключевые составляющие искомого объекта и в то же время позволит понизить размер обрабатываемой информации без потери качества работы модели в целом с параллельным отсеиванием шумовой составляющей в исходных данных.

Конечно, понижение объема информации само по себе ведет к сокращению расходов на ее обработку. Но мы также можем распараллелить вычисления в сверточном и подвыборочном слоях с использованием технологии многопоточных вычислений в *OpenCL*. Это позволит сократить время на выполнения расчетов с сохранением общего объема операций и тем самым сделает обучение и эксплуатацию нейронной сети намного быстрее.

Для организации многопоточных операций с использованием технологии *OpenCL* нам предстоит выполнить два блока операций:

- написание дополнительных кернелов в ранее созданной программе *OpenCL (opencl_program.cl);*
- организация процесса взаимодействия с контекстом *OpenCL* на стороне основной программы.

Разумеется, прежде чем организовывать передачу данных из основной программы в контекст *OpenCL*, необходимо понимать, когда и какие данные будет нужны. Поэтому свою работу мы начнем с внесения изменений в программе *OpenCL*.

Подвыборочный слой

Создание кернелов в программе OpenCL, как и построение классов в основной программе, начнем с методов подвыборочного слоя. Операции прямого прохода реализуем в кернеле *ProofFeedForward*. Из основной программы в кернел будем передавать два буфера данных:

- *inputs* — вектор исходных данных;
- *outputs* — вектор для записи результатов.

Для предотвращения ошибки выхода за пределы массива в параметрах кернелу передадим размер вектора исходных данных *inputs_total*.

Напомню, что в алгоритме сверточных нейронных сетей подвыборочный слой идет за сверточным слоем нейронов. В свою очередь, сверточный слой включает несколько фильтров. Следовательно, получая на вход результаты работы нескольких фильтров сверточного слоя в едином буфере, подвыборочный слой должен отработать каждый фильтр отдельно. Поэтому для логического разделения общего буфера результатов сверточного слоя по фильтрам дадим кернелу размер выходного вектора одного фильтра *input_neurons*.

Ну и конечно, в параметрах кернелу укажем размер окна для анализа исходных данных (*window*), шаг перемещения окна (*step*), количество фильтров (*window_out*) и функцию активации (*activation*).

```
__kernel void ProofFeedForward(__global double *inputs,
                               __global double *outputs,
                               int inputs_total,
                               int input_neurons,
                               int window,
                               int step,
                               int activation)
```

Данный кернел мы будем запускать в двухмерном пространстве задач. А значит, в каждом кернеле мы будем обрабатывать один элемент массива результатов в одном фильтре. Номер обрабатываемого элемента будем определять по идентификатору потока в измерении с индексом 0. Следовательно, общее количество потоков покажет нам количество элементов на выходе одного фильтра (*neurons*). По этим данным определим смещения до начала окна анализируемых данных в разрезе фильтра массива исходных данных (*shift*).

```
{
 const int n = get_global_id(0);
 const int w = get_global_id(1);
 const int neurons = get_global_size(0);
 const int window_out = get_global_size(1);
 int shift = n * step;
```

Второе измерение с индексом 1 укажет нам на индекс анализируемого фильтра. Соответственно, определим смещение в массивах исходных данных (*shift_inp*) и результатов (*out*) до начала обрабатываемого фильтра. Не забываем проверить на предмет выхода за пределы диапазона массива результатов.

Подготовим переменную для хранения промежуточных значений текущего элемента вектора результатов (*s*).

```
int out = w * neurons + n;
int shift_inp = w * input_neurons;
TYPE s = 0;
TYPE k = (TYPE)1 / (TYPE)window;
TYPE4 k4 = (TYPE4)(k);
```

Непосредственно вычисление значений результатов подвыборочного слоя будет осуществляться во вложенном массиве. В нем мы будем перебирать элементы исходных данных, попадающих в анализируемое окно, и собирать результирующее значение в соответствии с формулы активации.

Напомню, в нашей реализации подвыборочный слой может получить одну из двух функций активации:

- Average pooling — среднее арифметической элементов окна исходных данных;
- Max pooling — максимальный элемент окна исходных данных.

При вычислении среднего арифметического мы не будем собирать сумму всех элементов и затем делить на размер анализируемого окна. Наоборот, каждый элемент сначала разделим на размер окна, а потом суммируем полученные частные. Это позволит нам получить итоговый результат в теле цикла, исключив операцию деления за циклом. Само по себе проведение операции деления за циклом не критично, но только если это касается любых вариантов операций в цикле. В нашем же случае деление необходимо только в случае среднеарифметического. При использовании *Max pooling* деление излишне, и для корректной работы нам потребовалась бы дополнительная проверка функции активации. Переместив же деление во внутрь цикла, мы исключаем дополнительную проверку используемой функции активации, оставив ее только при непосредственном вычислении значения.

Обратите внимание, что для ускорения процесса мы используем векторные операции с типом данных *TYPE4*. Следовательно, и шаг цикла перебора элементов окна равен четырем.

```
for(int i = 0; i < window; i += 4)
   switch(activation)
     {
      case 0:
         s += dot(ToVect4(inputs, i, 1, min(shift_inp+input_neurons,inputs_total),
                          shift_inp + shift), k4);
         break;
      case 1:
         s = Max4(ToVect4(inputs, i, 1, min(shift_inp+input_neurons,inputs_total),
                          shift_inp + shift), s);
         break;
      default:
         break;
     }
 outputs[out] = s;
}
```

После выхода из цикла перебора элементов анализируемого окна сохраним полученное значение в соответствующий элемент вектора результатов и выйдем из кернела.

Мы рассмотрели кернел прямого прохода и можем перейти к построению алгоритма обратного распространения ошибки в процессе обратного прохода. Как уже обсуждалось ранее при построении алгоритма средствами MQL5, в подвыборочном слое алгоритм обратного прохода заключается только в передаче градиента ошибки через скрытый слой. Поэтому процесс построения обратного прохода будет заключаться в написании алгоритма кернела передачи градиента *ProofCalcHiddenGradient*.

С внешней программой новый кернел будет «общаться» через четыре буфера данных:

- *inputs* — буфер результатов предшествующего слоя;
- *gradient_inputs* — буфер градиентов предшествующего слоя (в данном случае служит для записи результатов работы кернела);
- *outputs* — буфер результатов прямого прохода текущего слоя;
- *gradients* — буфер градиентов на уровне результатов текущего слоя.

Контроль за соблюдением размеров буферов будет организован через параметры *inputs_total* и *outputs_total*. Названия параметров соответствуют буферам, размеры которых они хранят.

Не будем забывать, что в отличие от полносвязного слоя, нейроны подвыборочного слоя имеют ограниченные связи с нейронами предыдущего слоя. Определять зоны связей будем с помощью параметров *window*, *step.* Можно заметить, что одноименные параметры были объявлены в кернеле прямого прохода. Мы сохранили и их функциональное значение.

Добавим параметры количества элементов на выходе одного фильтра и используемой функции активации.

```
__kernel void ProofCalcHiddenGradient(__global TYPE *inputs,
                                      __global TYPE *gradient_inputs,
                                      __global TYPE *outputs,
                                      __global TYPE *gradients,
                                      int inputs_total,
                                      int outputs_total,
                                      int window,
                                      int step,
                                      int neurons,
                                      int activation)
```

При организации многопоточных вычислений следует учитывать проблему попыток одновременной записи в одни и те же элементы буфера из разных потоков. Поэтому наиболее приемлемыми являются алгоритмы, в которых каждому потоку предоставляются свои объекты для записи данных, и при этом они не пересекаются с объектами записи в других потоках.

Следуя вышеуказанной логике, мы будем создавать алгоритм, в котором каждый поток будет осуществлять сбор градиентов и запись в отдельный элемент буфера градиентов предыдущего слоя. Надо отметить одно отличие такого подхода от принятого нами в реализации средствами MQL5. При использовании функции активации *Max pooling* и наличии двух и более элементов со значениями равными максимальному, градиент будет в полном объеме передаваться на все такие элементы. В отличие от этого, в реализации основной программы мы передавали градиент только одному элементу. Принимая во внимание использование переменных и их точность, мы оцениваем риск появления такой ситуации минимальным и принимаем его.

В начале тела кернела определим порядковый номер искомого элемента и фильтра по идентификаторам потока. Общее количество потоков даст нам количество элементов одного

фильтра в буфере исходных данных (*input_neurons*) и количество фильтров (*window_out)*. По этим данным определим первый (*start*) и последний (*stop*) элементы вектора результатов, на которые влияет обрабатываемый элемент. При определении зоны влияния помним об ограничениях размерности буфера данных каждого фильтра. Следовательно, первый элемент не может быть меньше 0, а последний не может быть больше числа элементов в одном фильтре neurons.

```
    {
     const int n = get_global_id(0);
     const int w = get_global_id(1);
     const int input_neurons = get_global_size(0);
     const int window_out = get_global_size(1);
  //---
     int start = n - window + step;
     start = max((start - start % step) / step, 0);
     int stop = min((n - n % step) / step + 1, neurons);
```

Далее определим смещение анализируемого элемента в общем буфере исходных данных. При этом не забываем проверить на предмет выхода за пределы массива исходных данных.

После этого подготовим необходимые внутренние переменные. В первую очередь, это переменная для сбора промежуточных значений градиента (*grad*) и значение текущего элемента в буфере исходных данных (*inp*).

Создание последней обусловлено тем фактом, что при использовании *Max pooling* нам нужно будет постоянно сравнивать значение элемента исходных данных со значением из буфера результатов. По техническим причинам обращение к внутренним переменным значительно быстрее обращения к элементам буфера глобального массива. Это связано с местом хранения данных. Внутренние переменные хранятся в частной памяти, а буферы — в глобальной. Размер частной памяти мал, и мы не может туда скопировать весь массив, но обращение к ней занимает минимум времени. Размер глобальной памяти намного больше, но и время обращения к ней значительно дольше. Чтобы сократить общее время работы программы, перенесем используемое часто значение из глобальной в частную память контекста *OpenCL*.

```
     TYPE grad = 0;
     int shift_inp = w * input_neurons + n;
     if(shift_inp >= inputs_total)
        return;
     TYPE inp = inputs[shift_inp];
```

Затем организуем вложенный цикл, в котором будем перебирать элементы, попадающие в зону влияния анализируемого элемента исходных данных. В теле цикла мы сначала определим смещение обрабатываемого элемента в буфере градиентов ошибки результатов. Сразу проверим попадание в границы массива градиентов ошибки. Затем осуществим передачу градиента в соответствии с используемой функцией активации.

Для *Average pooling* мы просто разделим значение градиента ошибки на размер окна исходных данных и полученное значение прибавим к накопленному градиенту ошибки анализируемого элемента исходных данных. Обратите внимание, что делить градиент ошибки будем на размер окна исходных данных, а не зоны влияния. Ведь ошибка получена в результате прямого прохода, и на ее значение повлияли все элементы исходных данных, оказывающие влияние на конкретное значение.

В случае *Max pooling* мы сначала сравним значение соответствующих элементов на выходе и входе нейронного слоя. Только в случае их соответствия передадим градиент ошибки в полном объеме.

После выхода из цикла сохраним полученное значение градиента в буфер градиентов ошибки предыдущего слоя и завершим работу кернела.

```
   for(int o = start; o < stop; o ++)
     {
      int shift_g = w * neurons + o;
      if(shift_g >= outputs_total)
         break;
      switch(activation)
        {
         case 0:
            grad += gradients[shift_g] / (TYPE)window;
            break;
         case 1:
            grad += (outputs[shift_g] == inp ? gradients[shift_g] : 0);
            break;
         default:
            break;
        }
     }
   gradient_inputs[shift_inp] = grad;
  }
```

Два вышеуказанных кернела покрывают процессы прямого и обратного проходов в подвыборочном слое. Мы можем перейти к работе со сверточным слоем.

Сверточный слой

Для сверточного слоя нам также предстоит реализовать алгоритмы прямого и обратного проходов. По аналогии с рассмотренными ранее кернелами алгоритм прямого прохода будет описан в кернеле *ConvolutionFeedForward*. Сверточный слой, как и полносвязный, имеет матрицу весов и функцию активации. Поэтому для связи с основной программой нам потребуются четыре буфера данных:

- *inputs* — буфер исходных данных;
- *weights* — матрица весов;
- *sums* — вектор взвешенных сумм исходных данных перед функцией активации;
- *outputs* — вектор результатов.

Помимо буферов, для нормального функционирования новому кернелу потребуются параметры:

- *inputs_total* — размер массива исходных данных;
- *window* — размер анализируемого окна исходных данных;
- *step* — шаг окна исходных данных;
- *window_out* — количество фильтров в слое.

```
__kernel void ConvolutionFeedForward(__global TYPE *inputs,
                                     __global TYPE *weights,
                                     __global TYPE *outputs,
                                     int inputs_total,
                                     int window,
                                     int step,
                                     int window_out)
```

Построение самого алгоритма кернела подобно построению аналогичного кернела полносвязного нейрона. Как и в полносвязном слое, число потоков будет привязано к количеству элементов выходного буфера. Но учитывая специфику работы сверточного слоя, мы будем ориентироваться не на общее число элементов в буфере, а на число элементов в буфере результатов одного фильтра. При этом в одном потоке будут рассчитаны результаты *n*-го элемента всех фильтров.

В начале кернела проведем подготовительную работу. Определим порядковый номер обрабатываемого элемента в буфере результатов фильтра по номеру потока. Общее количество потоков даст нам количество элементов на выходе каждого фильтра. Из полученных данных и информации из параметров кернела рассчитаем смещение к началу анализируемого окна в буфере исходных данных и размер используемой матрицы весов.

```
{
 const int n = get_global_id(0);
 const int neurons = get_global_size(0);
 const int weights_total = (window + 1) * window_out;
 int shift = n * step;
```

Так как мы решили в одном потоке обрабатывать последовательно все фильтры, далее организуем цикл перебора фильтров. Внутри цикла определим смещение до обрабатываемого элемента в общем буфере результатов и смещение в матрице весов. Тут же проверим выход за пределы матрицы весов и подготовим внутреннюю переменную для сбора результирующего значения. Инициализировать переменную будем элементом *bias*.

```
 for(int w = 0; w < window_out; w++)
   {
    int out = (transposed_out == 1 ? w + n * window_out : w * neurons + n);
    int shift_weights = w * (window + 1) ;
    if((shift_weights + window) >= weights_total)
       break;
    TYPE s = weights[shift_weights + window];
```

Непосредственно вычисление взвешенной суммы анализируемого окна исходных данных будем осуществлять во вложенном цикле. В нем будем перебирать элементы анализируемого окна исходных данных и умножать на соответствующий весовой коэффициент. Для снижения затрат времени на выполнение воспользуемся векторными операциями. При этом не забываем увеличить размер шага цикла до размера используемых векторных переменных.

```
      for(int i = 0; i < window; i += 4)
         s += dot(ToVect4(inputs, i, 1, inputs_total, shift),
                  ToVect4(weights, i, 1, shift_weights + window, shift_weights));
      outputs[out] = s;
     }
  }
```

После сбора взвешенной суммы полученное значение запишем в буфер результатов.

Далее переходим к созданию кернелов процесса обратного прохода. В отличие от подвыборочного слоя сверточный слой содержит матрицу весов. Поэтому нам нужно будет создать более одного кернела, как и в аналогичном процессе полносвязного слоя.

Начнем мы построение процесса, как и раньше, в порядке алгоритма обратного распространения ошибки. Корректировку градиента на производную функции активации полностью будем осуществлять с использованием наработок сделанных для полносвзного слоя. Непосредственно работу над сверточным слоем начнем с создания кернела распространения градиента через слой *ConvolutionCalcHiddenGradient*.

В данном случае передача градиента на нижний слой не зависит от исходных данных и результатов прямого прохода. Поэтому для работы нашего кернела мы передадим ему три буфера данных:

- *gradient_inputs* — буфер градиентов ошибки предшествующего слоя (в данном случае буфер результатов);
- *weights* — матрица весов;
- *gradients* — буфер градиентов ошибки на входе текущего слоя.

Помимо буферов данных, для правильной работы кернела потребуются ряд параметров:

- *outputs_total* — общее количество элементов в буфере результатов (градиентов на выходе текущего нейронного слоя);
- *window* — размер окна исходных данных (количество элементов исходных данных анализируемых одним нейроном текущего слоя);
- *step* — шаг перемещения окна по массиву исходных данных;
- *window_out* — количество фильтров в текущем сверточном слое;
- *neurons* — количество элементов на выходе одного фильтра.

```
__kernel void ConvolutionCalcHiddenGradient(__global TYPE *gradient_inputs,
                                            __global TYPE *weights,
                                            __global TYPE *gradients,
                                            int window,
                                            int step,
                                            int window_out,
                                            int neurons)
```

Запускаться кернел будет в многопоточном режиме с числом потоков равным числу элементов в буфере градиентов ошибок предыдущего слоя, которое равно числу элементов в буфере исходных данных.

В начале кернела, как обычно, определим порядковый номер обрабатываемого элемента по номеру текущего потока и число элементов в буфере градиентов предыдущего слоя по общему

числу запущенных потоков. Кроме того, посчитаем размер матрицы весов исходя из размера окна исходных данных и числа фильтров в текущем сверточном слое.

```
{
 const int n = get_global_id(0);
 const int inputs_total = get_global_size(0);
 int weights_total = (window + 1) * window_out;
```

Продолжая подготовительную работу, определим зону влияния текущего элемента в буфере результатов одного фильтра и подготовим внутреннюю переменную для записи промежуточных результатов накопления градиента ошибки для обрабатываемого элемента.

```
 TYPE grad = 0;
 int w_start = n % step;
 int r_start = max((n - window + step) / step, 0);
 int total = (window - w_start + step - 1) / step;
 total = min((n + step) / step, total);
```

Напомню, что при создании класса сверточного слоя в основной программе мы решили воспринимать массив исходных данных единым целом и применять все фильтры к общему объему данных. Следовательно, каждый элемент исходных данных оказывает влияние на результаты работы всех фильтров. Значит, и градиент ошибки на каждом элементе исходных данных нам предстоит собрать со всех фильтров. Поэтому для сбора градиентов ошибок нам потребуется система вложенных циклов с перебором фильтров и элементов каждого фильтра.

Во внешнем цикле происходит перебора элементов вектора градиентов ошибки на выходе текущего нейронного слоя. В нем мы определим смещение до конкретного элемента в векторе градиента и сразу проверим на выход за пределы размера фильтра.

```
 for(int i = 0; i < total; i ++)
   {
    int row = r_start + i;
    if(row >= neurons)
       break;
```

В теле вложенного цикла мы сначала определим смещение в буфере градиентов ошибки на выходе текущего слоя и матрице весов, а затем прибавим произведение значений этих элементов к ранее накопленному градиенту ошибки для анализируемого элемента исходных данных.

```
    for(int wo = 0; wo < window_out; wo++)
      {
       int shift_g = (transposed_out == 1 ? row * window_out + wo :
                                                    row + wo * neurons);
       int shift_w = w_start + (total - i - 1) * step + wo * (window + 1);
       grad += gradients[shift_g] * weights[shift_w];
      }
   }
 gradient_inputs[n] = grad;
}
```

После завершения всех итераций и выхода из блока двух вложенных циклов значение накопленного градиента сохраняется в буфере градиентов ошибки предыдущего слоя.

За распределением градиента ошибки через скрытые слои нейронной сети, в соответствии с алгоритмом метода обратного распространения ошибки следует перенос градиента ошибки на весовые коэффициенты. Для выполнения этого функционала создадим кернел *ConcolutionCalcDeltaWeights*.

Для корректной работы кернела потребуется использование 3-х буферов данных:

- *inputs* — буфер исходных данных;
- *delta_weights* — буфер накопленных градиентов ошибок для матрицы весов (в данном случае — буфер результатов);
- *gradients* — буфер градиентов ошибки текущего слоя (на уровне результатов).

Напомню, что в буфере градиентов содержатся значения градиентов ошибки уже скорректированные на производную функции активации. Так как эта процедура выполнена перед передачей градиента ошибки на предыдущий слой. Поэтому, корректировка на производную функции активации на данном этапе будет излишней.

В дополнение к буферам данных нам понадобится ввести несколько параметров, чтобы корректно построить алгоритм:

- *inputs_total* — общее число элементов в буфере результатов и, соответственно, буфере градиентов ошибки;
- *step* — шаг перемещения окна анализируемых данных по массиву исходных данных;
- *neurons* — количество элементов на выходе одного фильтра.

```
__kernel void ConvolutionCalcDeltaWeights(__global TYPE *inputs,
                                          __global TYPE *delta_weights,
                                          __global TYPE *gradients,
                                          int inputs_total,
                                          int step,
                                          int neurons)
```

Можно заметить, что в числе параметров отсутствуют переменные для указания размера окна анализируемых данных и количества фильтров в текущем сверточном слое. Это связано с изменением подхода к созданию потоков для операций. В данном кернеле будут собираться градиенты ошибок на уровне матрицы весов, поэтому вполне логично запускать кернел для каждого весового коэффициента. Кроме того, матрица весов представлена в виде двумерной таблицы, в которой каждая строка является отдельным фильтром, а элементы этой строки являются весовыми коэффициентами соответствующего фильтра.

Технология *OpenCL* позволяет осуществлять запуск потоков в двухмерном пространстве с указанием каждому потоку двух индексов. Воспользуемся этим свойством и создадим потоки для данного кернела в двух измерениях. В первом измерении количество потоков будет равно количеству весовых коэффициентов в одном фильтре. Во втором измерении количество потоков будет соответствовать количеству используемых фильтров.

В теле кернела определим положение анализируемого элемента в матрице весов и ее размеры. Здесь следует напомнить, что каждый фильтр имеет весовой коэффициент смещения *bias*, поэтому размер анализируемого окна данных будет на один элемент меньше числа потоков в первом измерении (измерение с индексом 0).

Тут же мы определим положение анализируемого элемента в одномерном буфере матрицы весов и смещение до начала соответствующего фильтра в буфере градиентов ошибки. И конечно,

подготовим переменную для хранения промежуточных значений накопленного градиента ошибки.

```
  {
   const int inp_w = get_global_id(0);
   const int w = get_global_id(1);
   const int window = get_global_size(0) - 1;
   const int window_out = get_global_size(1);
//---
   int shift_delt = w * (window + 1) + inp_w;
   TYPE value = 0;
```

Далее идет процесс непосредственно подсчета градиента ошибки. Тут надо вспомнить, что для элемента смещения *bias* нет соответствующих элементов в буфере исходных данных. Следовательно, на данный элемент градиент будет передаваться в полном объеме. Чтобы не осуществлять проверку на каждой итерации цикла, мы ее сделаем один раз перед запуском цикла. В цикле мы будем перебирать элементы буферов градиента ошибки и исходных данных, а элемент матрицы весов остается неизменным.

Таким образом, сначала мы проверяем, является ли текущий элемент матрицы весов смещением *bias*, и затем организовываем цикл с перебором всех элементов градиентов ошибки соответствующего фильтра. Внутри цикла будем суммировать градиент ошибки, скорректированный на соответствующее значение буфера исходных данных.

После выхода из цикла прибавим полученное значение к ранее накопленному градиенту ошибки для анализируемого элемента матрицы весов. Напомню, что мы не будем обновлять матрицу весов на каждой итерации обратного прохода. Мы только накапливаем градиент ошибки. Обновление матрицы весов осуществляется по команде из основной программы после обработки установленного пользователем пакета данных.

```
   if(inp_w == window)
     {
      for(int n = 0; n < neurons; n ++)
         value += gradients[w * neurons + n];
     }
   else
      for(int n = 0; n < neurons; n ++)
        {
         int shift_inp = n * step + inp_w;
         if(shift_inp >= inputs_total)
            break;
         value += inputs[shift_inp] * gradients[w * neurons + n];
        }
   delta_weights[shift_delt] += value;
  }
```

После распределения градиента ошибки до матрицы весов алгоритмом обратного распространения ошибки предусмотрено ее обновление. Корректировка весовых коэффициентов осуществляется в сторону антиградиента. Как уже говорилось при создании сверточного слоя средствами MQL5, созданный ранее процесс для полносвязного слоя полностью отвечает требованиям работы и в сверточных слоях. Поэтому мы не будет создавать отдельные кернел и блоки основной программы, а воспользуемся ранее созданным решением.

Реализация функционала на стороне основной программы

После дополнения программы *OpenCL* новыми кернелами нам предстоит встроить в основную программу блоки кода для организации процесса обмена данными и запуска кернелов на выполнение в нужное время и с нужным объемом информации. Давайте подробно разберем каким, образом это можно реализовать.

Как вы помните, при построении полносвязного нейронного слоя аналогичную работу мы начинали с объявления констант. Сейчас мы сделаем то же самое: объявим константы для вызова каждого кернела.

```
#define def_k_ProofFeedForward             21
#define def_k_ProofHiddenGradients         22
#define def_k_ConvolutionFeedForward       23
#define def_k_ConvolutionHiddenGradients   24
#define def_k_ConvolutionDeltaWeights      25
```

Также объявим константы параметров каждого кернела. Константы параметров должны строго соответствовать порядковому номеру параметра в кернеле программы *OpenCL*. Нумерация параметров начинается с нуля.

```
//--- прямой проход подвыборочного слоя
#define def_prff_inputs                 0
#define def_prff_outputs                1
#define def_prff_inputs_total           2
#define def_prff_input_neurons          3
#define def_prff_window                 4
#define def_prff_step                   5
#define def_prff_activation             6

//--- распределение градиента через подвыборочный слой
#define def_prhgr_inputs                0
#define def_prhgr_gradient_inputs       1
#define def_prhgr_outputs               2
#define def_prhgr_gradients             3
#define def_prhgr_inputs_total          4
#define def_prhgr_outputs_total         5
#define def_prhgr_window                6
#define def_prhgr_step                  7
#define def_prhgr_neurons               8
#define def_prff_activation             9

//--- прямой проход сверточного слоя
#define def_cff_inputs                  0
#define def_cff_weights                 1
#define def_cff_outputs                 2
#define def_cff_inputs_total            3
#define def_cff_window                  4
#define def_cff_step                    5
#define def_cff_window_out              6

//--- распределение градиента через сверточный слой
#define def_convhgr_gradient_inputs     0
#define def_convhgr_weights             1
#define def_convhgr_gradients           2
#define def_convhgr_window              3
#define def_convhgr_step                4
#define def_convhgr_window_out          5
#define def_convhgr_neurons             6

//--- распределение градиента на матрицу весов сверточного слоя
#define def_convdelt_inputs             0
#define def_convdelt_delta_weights      1
#define def_convdelt_gradients          2
#define def_convdelt_inputs_total       3
#define def_convdelt_step               4
#define def_convdelt_neurons            5
```

После объявления констант нам нужно обновить список используемых кернелов из программы *OpenCL*. Напомню, что эта работа осуществляется в методе *CNet::InitOpenCL*. Здесь нам нужно изменить количество используемых кернелов на 26.

```
   if(!m_cOpenCL.SetKernelsCount(26))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
```

Создадим точки входа для новых кернелов.

```
   if(!m_cOpenCL.KernelCreate(def_k_ProofFeedForward, "ProofFeedForward"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
   if(!m_cOpenCL.KernelCreate(def_k_ProofHiddenGradients,
                                                 "ProofCalcHiddenGradient"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
   if(!m_cOpenCL.KernelCreate(def_k_ConvolutionFeedForward,
                                                  "ConvolutionFeedForward"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
   if(!m_cOpenCL.KernelCreate(def_k_ConvolutionHiddenGradients,
                                           "ConvolutionCalcHiddenGradient"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
   if(!m_cOpenCL.KernelCreate(def_k_ConvolutionDeltaWeights,
                                             "ConcolutionCalcDeltaWeights"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
```

Дальнейшая работа продолжится непосредственно в соответствующих методах. Помните, при построении классов во многих методах мы делали разветвление алгоритма в зависимости от устройства выполнения операций. Часть *MQL5* мы написали сразу. Сейчас нам предстоит описать алгоритм работы с контекстом *OpenCL*.

Дополнять методы будем в той же последовательности, в которой мы их создавали ранее. Начнем эту работу с метода прямого прохода подвыборочного слоя *CNeuronProof::FeedForward.* Для корректной работы этот метод использует два буфера данных: исходных данных и результатов. В начале блока проверим наличие указанных буферов в контекст *OpenCL*. Наличие хендла буфера будет свидетельствовать о ранее переданном буфере в контекст *OpenCL*.

```
bool CNeuronProof::FeedForward(CNeuronBase *prevLayer)
  {
//--- Блок контролей
   if(!prevLayer || !m_cOutputs ||
      !prevLayer.GetOutputs())
      return false;
   CBufferType *input_data = prevLayer.GetOutputs();
//---  Разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
     // Здесь пропущен Блок MQL5
     }
   else // Блок работы с OpenCL
     {
      //--- проверим наличие буферов в контекст OpenCL
      if(input_data.GetIndex() < 0)
         return false;
      if(m_cOutputs.GetIndex() < 0)
         return false;
```

При наличии данных в контексте *OpenCL* передадим в кернел указатели на буферы данных и параметры, необходимые для его работы. На каждом шаге также проверяем результат выполнения операций. Это очень важно, так как запуск кернела с недостающей информацией может привести к критической ошибке и остановке всей программы.

```
      //--- Передача параметров в кернел
      if(!m_cOpenCL.SetArgumentBuffer(def_k_ProofFeedForward, def_prff_inputs,
                                                          input_data.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_ProofFeedForward, def_prff_outputs,
                                                          m_cOutputs.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ProofFeedForward, def_prff_inputs_total,
                                                            input_data.Total()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ProofFeedForward, def_prff_window,
                                                                  m_iWindow))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ProofFeedForward, def_prff_step, m_iStep))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ProofFeedForward, def_prff_activation,
                                                            (int)m_eActivation))
         return false;
      uint input_neurons = (input_data.Total()+m_iWindowOut-1) / m_iWindowOut;
      if(!m_cOpenCL.SetArgument(def_k_ProofFeedForward, def_prff_input_neurons,
                                                               input_neurons))
         return false;
```

Когда вся необходимая информация будет передана в кернел, нам остается указать количество потоков для запуска кернела и начальное смещение в пространстве задач. После этого мы инициируем выполнение кернела и завершаем работу метода.

```
      //--- Постановка кернела в очередь на выполнение
      uint off_set[] = {0, 0};
      uint NDRange[] = {m_iNeurons, m_iWindowOut};
      if(!m_cOpenCL.Execute(def_k_ProofFeedForward, 2, off_set, NDRange))
         return false;
    }
//---
   return true;
  }
```

После добавления кода метода прямого прохода подвыборочного слоя *CNeuronProof::FeedForward* проведем аналогичную работу в методе обратного прохода *CNeuronProof::CalcHiddenGradient*. В отличие от прямого прохода, кернел распределения градиента ошибки через подвыборочный слой использует четыре буфера данных:

- исходных данных,
- результатов прямого прохода,
- градиентов ошибок на выходе нейронного слоя,
- градиентов ошибок на уровне исходных данных (буфер результатов в данном случае).

Первые два буфера используются для определение нужных элементов при использовании *Max pooling*.

Следовательно, нам предстоит загрузить все четыре буфера в память контекста *OpenCL*.

```
bool CNeuronProof::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
//--- Блок контролей
   if(!prevLayer || !m_cOutputs ||
      !m_cGradients || !prevLayer.GetOutputs() ||
      !prevLayer.GetGradients())
      return false;
   CBufferType *input_data = prevLayer.GetOutputs();
   CBufferType *input_gradient = prevLayer.GetGradients();
   if(!input_gradient.BufferInit(input_data.Rows(), input_data.Cols(), 0))
      return false;
//---  Разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
     // Здесь пропущен Блок MQL5
     }
   else     // Блок работы с OpenCL
     {
      //--- проверяем наличие буферов в контекст OpenCL
      if(input_data.GetIndex() < 0)
         return false;
      if(m_cOutputs.GetIndex() < 0)
         return false;
      if(input_gradient.GetIndex() < 0)
         return false;
      if(m_cGradients.GetIndex() < 0)
         return false;
```

При наличии данных в память контекста *OpenCL* передадим указатели на буферы и необходимые константы в параметры кернела. При этом не забываем контролировать результаты выполнения операций.

```
      //--- Передача параметров в кернел
      if(!m_cOpenCL.SetArgumentBuffer(def_k_ProofHiddenGradients,
                                      def_prhgr_inputs, input_data.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_ProofHiddenGradients,
                                     def_prhgr_outputs, m_cOutputs.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_ProofHiddenGradients,
                                   def_prhgr_gradients, m_cGradients.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_ProofHiddenGradients,
                           def_prhgr_gradient_inputs, input_gradient.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ProofHiddenGradients,
                                    def_prhgr_inputs_total, input_data.Total()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ProofHiddenGradients,
                                                 def_prhgr_window, m_iWindow))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ProofHiddenGradients,
                                                     def_prhgr_step, m_iStep))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ProofHiddenGradients,
                                     def_prhgr_activation, (int)m_eActivation))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ProofHiddenGradients,
                                               def_prhgr_neurons, m_iNeurons))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ProofHiddenGradients,
                                    def_prhgr_outputs_total, m_cOutputs.Total()))
         return false;
```

Затем укажем количество потоков для запуска кернела и смещение в области задач. После этого поставим кернел в очередь выполнения.

Обратите внимание, что при запуске кернела прямого прохода количество потоков равно количеству элементов на выходе одного фильтра подвыборочного слоя. При запуске кернела обратного прохода количество потоков равно количеству элементов в одном фильтре предыдущего нейронного слоя.

```
      //--- Постановка кернела в очередь на выполнение
      uint input_neurons = (input_data.Total() + m_iWindowOut - 1) / m_iWindowOut;
      uint off_set[] = {0, 0};
      uint NDRange[] = {input_neurons, m_iWindowOut};
      if(!m_cOpenCL.Execute(def_k_ProofHiddenGradients, 2, off_set, NDRange))
         return false;
     }
//---
   return true;
  }
```

На этом работа с классом подвыборочного слоя завершена. Мы переходим к выполнению аналогичной работы с классом сверточного слоя *CNeuronConv*.

Сверточный нейронный слой, в отличие от подвыборочного, имеет матрицу весов и функцию активации. Следовательно, для его работы потребуется использование большего количества буферов. Метод прямого прохода сверточного слоя *CNeuronConv::FeedForward* требует передачи в память контекста *OpenCL* 4-х буферов:

- исходные данные,
- матрица весов,
- дополнительный буфер функции активации (используется для функции активации *Swish*),
- буфер результатов.

Работу в методе прямого прохода *CNeuronConv::FeedForward* начнем с проверки наличия используемых буферов в контексте *OpenCL*.

```
bool CNeuronConv::FeedForward(CNeuronBase *prevLayer)
  {
//--- блок контролей
   if(!prevLayer || !m_cOutputs || !m_cWeights || !prevLayer.GetOutputs())
      return false;
   CBufferType *input_data = prevLayer.GetOutputs();
   ulong total = input_data.Total();
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
     // Здесь пропущен Блок MQL5
     }
   else
     {
      //--- проверка буферов данных
      if(input_data.GetIndex() < 0)
         return false;
      if(m_cWeights.GetIndex() < 0)
         return false;
      if(m_cOutputs.GetIndex() < 0)
         return false;
```

Затем мы должны передать указатели на буферы в соответствующий кернел. Кроме того, в параметрах кернела мы будем передавать и некоторые константы, необходимые для корректной работы алгоритма. Среди передаваемых параметров будут размеры анализируемого окна, шаг окна и количество фильтров. На каждом шаге контролируем процесс выполнения операций.

```
         //--- передача аргументов кернелу
         if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionFeedForward,
                                         def_cff_inputs, input_data.GetIndex()))
            return false;
         if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionFeedForward,
                                        def_cff_weights, m_cWeights.GetIndex()))
            return false;
         if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionFeedForward,
                                        def_cff_outputs, m_cOutputs.GetIndex()))
            return false;
         if(!m_cOpenCL.SetArgument(def_k_ConvolutionFeedForward,
                                      def_cff_inputs_total, input_data.Total()))
            return false;
         if(!m_cOpenCL.SetArgument(def_k_ConvolutionFeedForward,
                                                    def_cff_window, m_iWindow))
            return false;
         if(!m_cOpenCL.SetArgument(def_k_ConvolutionFeedForward,
                                                          def_cff_step, m_iStep))
            return false;
         if(!m_cOpenCL.SetArgument(def_k_ConvolutionFeedForward,
                                                 def_cff_window_out, m_iWindowOut))
            return false;
```

После передачи всех необходимых данных кернелу укажем количество запускаемых потоков и инициируем его постановку в очередь выполнения.

```
         //--- постановка кернела в очередь выполнения
         int off_set[] = {0};
         int NDRange[] = {(int)m_iNeurons};
         if(!m_cOpenCL.Execute(def_k_ConvolutionFeedForward, 1, off_set, NDRange))
            return false;
        }
      if(!m_cActivation.Activation(m_cOutputs))
         return false;
   //---
      return true;
     }
```

В завершение вызываем функцию активации и завершаем работу метода.

Прямой проход реализован. Начнем реализацию обратного прохода в сверточном нейронном слое. Как вы помните, обратный проход включает в себя три подпроцесса:

- распределение градиента ошибки по нейронной сети от результата к исходным данным;
- распределение градиент ошибки до матрицы весов каждого нейронного слоя;
- корректировка матрицы весов в сторону антиградиента.

Из уже реализованных методов средствами *MQL5* мы знаем, что для последнего подпроцесса новый метод не создавался. Вместо этого предложено использовать уже готовый метод полносвязного нейронного слоя, в котором мы уже реализовали и многопоточные вычисления средствами *OpenCL*. Следовательно, на данном этапе нам предстоит доработать только методы двух первых подпроцессов.

За распределение градиента ошибки через сверточный слой отвечает метод *CNeuronConv::CalcHiddenGradient*. Корректное выполнение алгоритма данного метода требует наличия трех буферов данных:

- буфер градиентов ошибки на выходе нейронного слоя (получаем от последующего слоя в процессе выполнения аналогичного метода);
- матрица весов;
- буфер градиентов ошибки на уровне исходных данных (в данном случае выполняет роль буфера результатов работы метода).

Поэтому в начале блока работы с технологией *OpenCL* мы проверяем наличие необходимых буферов в памяти контекста.

```
bool CNeuronConv::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
//--- блок контролей
   if(!prevLayer || !prevLayer.GetOutputs() || !prevLayer.GetGradients() ||
      !m_cGradients || !m_cWeights)
      return false;
//--- корректировка градиентов ошибки на производную функции активации
   if(m_cActivation)
     {
      if(!m_cActivation.Derivative(m_cGradients))
         return false;
     }
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   CBufferType* input_gradient = prevLayer.GetGradients();
   if(!m_cOpenCL)
     {
     // Здесь пропущен Блок MQL5
     }

   else // Блок работы с OpenCL
     {
      //--- проверка буферов данных
      if(m_cWeights.GetIndex() < 0)
         return false;
      if(input_gradient.GetIndex() < 0)
         return false;
      if(m_cGradients.GetIndex() < 0)
         return false;
```

Следующим шагом передадим необходимые данные в параметры кернела. Среди них — указатели на используемые буфера данных.

```
      //--- передача аргументов кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionHiddenGradients,
                        def_convhgr_gradient_inputs, input_gradient.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionHiddenGradients,
                                  def_convhgr_weights, m_cWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionHiddenGradients,
                               def_convhgr_gradients, m_cGradients.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ConvolutionHiddenGradients,
                                            def_convhgr_neurons, m_iNeurons))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ConvolutionHiddenGradients,
                                              def_convhgr_window, m_iWindow))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ConvolutionHiddenGradients,
                                                  def_convhgr_step, m_iStep))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ConvolutionHiddenGradients,
                                        def_convhgr_window_out, m_iWindowOut))
         return false;
```

Далее укажем количество потоков равное количеству элементов в буфере исходных данных и отправим кернел в очередь на выполнение.

```
      //--- постановка кернела в очередь выполнения
      int NDRange[] = {(int)input_gradient.Total()};
      int off_set[] = {0};
      if(!m_cOpenCL.Execute(def_k_ConvolutionHiddenGradients, 1, off_set, NDRange))
         return false;
     }
//---
   return true;
  }
```

Для завершения работы над методами обратного прохода в сверточной сети нам осталось внести аналогичные изменения в метод распределения градиента ошибки до матрицы весов *CNeuronConv::CalcDeltaWeights* с учетом специфики данного метода.

Алгоритм работы метода распределения градиента ошибки до матрицы весов требует наличие трех буферов:

- градиент ошибки на уровне выхода нейронного слоя;
- буфер исходных данных;
- буфер для накопления градиентов ошибки на уровне матрицы весов.

Проверим наличие указанных буферов в памяти контекста *OpenCL*. Напомню, что мы исходим из предположения о достаточности видеопамяти для хранения всей модели. Если же модель полностью на помещается в памяти вашего *OpenCL* устройства, то потребуется перед запускам каждого кернела загружать необходимые данные в память контекста. И после завершения работы кернела освобождать память для загрузки следующей партии данных.

```
bool CNeuronConv::CalcDeltaWeights(CNeuronBase *prevLayer)
  {
//--- блок контролей
   if(!prevLayer || !prevLayer.GetOutputs() || !m_cGradients || !m_cDeltaWeights)
      return false;
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   CBufferType *input_data = prevLayer.GetOutputs();
   if(!m_cOpenCL)
     {
     // Здесь пропущен Блок MQL5
     }
   else // Блок работы с OpenCL
     {
      //--- проверка буферов данных
      if(m_cGradients.GetIndex() < 0)
         return false;
      if(m_cDeltaWeights.GetIndex() < 0)
         return false;
      if(input_data.GetIndex() < 0)
         return false;
```

Затем передаем необходимые параметры кернелу, соответствующему нашему подпроцессу. Напомню, очень важно соблюдать соответствие указанных ID кернела, ID параметра и указанного значения, а также на каждом шаге контролируем процесс выполнения операций.

```
      //--- передача аргументов кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionDeltaWeights,
                        def_convdelt_delta_weights, m_cDeltaWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionDeltaWeights,
                                  def_convdelt_inputs, input_data.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_ConvolutionDeltaWeights,
                              def_convdelt_gradients, m_cGradients.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ConvolutionDeltaWeights,
                                def_convdelt_inputs_total, input_data.Total()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ConvolutionDeltaWeights,
                                             def_convdelt_neurons, m_iNeurons))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_ConvolutionDeltaWeights,
                                                   def_convdelt_step, m_iStep))
         return false;
```

Когда вся необходимая информация передана в кернел, указываем количество потоков. В данном случае мы решили использовать двумерное распределение потоков:

- по количеству фильтров,
- по количеству весовых коэффициентов в одном фильтре.

Для этого мы указываем два параметра в массиве *NDRange*. Каждый параметр задает размерность соответствующей области задач. Отправляем кернел в очередь выполнения.

```
         //--- постановка кернела в очередь выполнения
         uint NDRange[] = {m_iWindow + 1, m_iWindowOut};
         uint off_set[] = {0, 0};
         if(!m_cOpenCL.Execute(def_k_ConvolutionDeltaWeights, 2, off_set, NDRange))
            return false;
        }
   //---
      return true;
     }
```

Теперь мы создали уже три типа полностью функциональных нейронных слоев для нашего конструктора нейронных сетей и можем сравнить их эффективность на решении практической задачи. Предлагаю провести несколько экспериментов в следующей главе. Но прежде чем приступить к «полевым испытаниям», нам предстоит еще проверить корректность работы методов передачи градиентов.

4.1.4 Реализация сверточной модели в Python

Реализацию сверточных моделей на языке Python мы будем осуществлять с помощью средств, предлагаемых библиотекой *Keras* из *TensorFlow*. Указанная библиотека предлагает несколько вариантов сверточных слоев. В первую очередь, это базовые варианты сверточных слоев:

- *Conv1D*
- *Conv2D*
- *Conv3D*

Как можно определить из названия объектов сверточных слоев, они предназначены для обработки исходных данных различной размерности.

Объекты класса *Conv1D* создают ядро свертки, которое сворачивается с исходными данными в одном измерении для создания тензора выходных данных. Здесь важно понять и не путать. Исходные данные сворачиваются в одном измерении, но исходные данные, подаваемые на вход нейронного слоя, должны иметь вид трехмерного тензора. Первое измерение определяет размер пакета обрабатываемых данных (*batch size*). Второе — измерение свертки. А в третьем измерении содержатся исходные данные для свертки.

В результате обработки данных слой также возвращает трехмерный тензор. Первое измерение остается неизменным, оно равно размеру пакета обрабатываемых данных. Второе измерение изменяется в зависимости от указанных параметров свертки. А третье измерение будет равно заданному количеству используемых фильтров.

Надо понимать, что каждый фильтр применяется ко всем исходным данным. За один раз обрабатываются исходные данные в размере третьего измерения помноженного на размер окна свертки. Здесь есть небольшое отличие от нашей реализации сверточного слоя на языке MQL5. Там мы определяли окно свертки как количество элементов, а здесь окно свертки определяет количество элементов второго измерения трехмерного тензора исходных данных.

Один фильтр возвращает по одному значению для каждого окна свертки. Так как у нас в процессе свертки участвует все третье измерение, то в нем мы получаем по одному элементу от каждого фильтра. Как следствие, размер третьего измерения выходного тензора изменяется на количество используемых фильтров.

Как и полносвязный слой, класс сверточного слоя предлагает довольно широкий спектр параметров для тонкой настройки работы. Посмотрим на них.

- ***filters*** — количество фильтров, используемых в свертке;
- ***kernel_size*** — размер окна одномерной свертки;
- ***strides*** — размер шага свертки;
- ***padding*** — допускается одно из значений: "*valid*", "*same*" или "*causal*" (без учета регистра); "*valid*" — означает отсутствие отступов; "*same*" — приводит к равномерному дополнению нулями исходных данных для получения размера вывода равного размеру входа; "*causal*" — приводит к возникновению причинных (расширенных) изменений, например *output[t]* не зависит от *input[t+1:]*. Полезно при моделировании временных данных, когда модель не должна нарушать временной порядок;
- ***data_format*** — допускается одно из значений: "*channels_last*" или "*channels_first*"; определяет в каком измерении входного тензора содержатся данные для свертки; по умолчанию используется "*channels_last*";
- ***dilation_rate —*** используется для расширенной свертки и определяет скорость расширения;
- ***groups*** — количество групп, на которые вход разделяется по оси канала; каждая группа сворачивается отдельно с помощью фильтров, на выходе получаем объединение всех результатов по оси канала;
- ***activation*** — функция активации;
- ***use_bias*** — указывает на необходимость использования вектора смещения;
- ***kernel_initializer*** — задает метод инициализации матрицы весов;
- ***bias_initializer*** — задает метод инициализации вектора смещения;
- ***kernel_regularizer*** — указывает на метод регуляризации матрицы весов;
- ***bias_regularizer —*** указывает на метод регуляризации вектора смещения;
- ***activity_regularizer*** — указывает на метод регуляризации результатов;
- ***kernel_constraint*** — указывает функцию ограничения для матрицы весов;
- ***bias_constraint*** — указывает функцию ограничения для вектора смещения.

Для таймсерий обычно предлагают использовать одномерную свертку Conv1D. Свертка осуществляется по временным интервалам. При этом каждый фильтр проверяет наличие своего паттерна в конкретном временном интервале. Применительно к решению нашей задачи фильтры будут оценивать состояние всех используемых индикаторов в рамках количества свечей, заданных параметром *strides*. Процесс свертки, как и нейроны полносвязного слоя, не оценивает взаимовлияние отдельных составляющих исходных данных. Он лишь оценивает сходство исходных данных с некоторым заданным паттерном. Конечно, мы не задаем эти паттерны при конструировании нейронной сети. Мы их подбираем в процессе обучения. Но предполагается, что в процессе промышленной эксплуатации паттерны будут статичны между периодами переобучения.

Да, сверточные слои более устойчивы к различным искажениям исходных данных благодаря тому, что как под микроскопом изучают небольшие отдельно взятые блоки. Тем не менее, может возникнуть необходимость в изучении паттернов отдельно взятых индикаторов. Для решения такой задачи нам может потребоваться использовать сверточные слои другой размерности.

К примеру, объекты класса Conv2D оперируют свертками с исходными данными в двух измерениях. При этом следует понимать, что отличие между одномерными и двухмерными сверточными слоями лежит не только в названии. Объекты двухмерного сверточного слоя

ожидают на вход четырехмерный тензор. По аналогии с тензором *Conv1D*, первое измерение определяет размер пакета обрабатываемых данных (batch size), второе и третье — определяют измерения свертки, а в четвертом измерении содержатся исходные данные для свертки. Тут возникает справедливый вопрос: где нам взять данные для еще одного измерения? Как нам поделить наш набор исходных данных на четыре измерения? Нам нужно перевести наши исходные данные из плоской в объемную таблицу. Наиболее простое решение лежит на поверхности. Мы говорим, что глубина таблицы исходных данных равна 1. Перед объявлением двухмерного нейронного слоя изменим размерность тензора, подаваемого на вход сверточного слоя Conv2D, до четырехмерного, указав в размере четвертого измерения 1.

Обратите внимание: поскольку четвертое измерение равно 1, то и длина вектора исходных данных для свертки равна 1. Следовательно, чтобы процесс свертки был эффективным, необходимо чтобы окно свертки было больше 1 как минимум в одном измерении.

На параметрах сверточного слоя Conv2D мы не будем сильно останавливаться, так как они идентичны параметрам одномерного массива. Отличия лишь в параметрах *kernel_size*, *strides* и *dilation_rate*, которые, кроме скалярного значения, могут принимать вектор из двух элементов. Каждый элемент такого вектора содержит значения параметра для соответствующего измерения. В то же время указанные параметры могут принимать скалярные значения. В этом случае для обоих измерений будет использоваться указанное значение.

Для более сложных архитектурных решений нейронных сетей может потребоваться использование трехмерных сверточных слоев Conv3D. Их использование может быть оправдано, к примеру, для построения арбитражных торговых систем, когда нам потребуется отдельное измерение для разделения исходных данных по инструментам.

Как и в случае с двухмерным сверточным слоем, использование трехмерного пространства требует увеличение размерности исходных данных. На входе Conv3D ожидается пятимерный тензор.

Параметры же класса Conv3D переходят из выше рассмотренных классов практически без изменений. Разница лишь в размерности векторов окна свертки и его шага.

Следует обратить внимание еще на одну особенность процесса свертки. При выполнении операций возможно как уменьшение размера тензора данных (сжатие данных), так и его увеличение. Первый вариант полезен при обработке больших массивов данных, когда из общего объема подаваемой на вход информации необходимо извлечь некую составляющую. Этим часто пользуются в задачах компьютерного зрения, когда на снимках с большим разрешением каждый пиксель является отдельным значением в общем тензоре исходных данных.

Второй вариант, увеличение размерности, может быть полезен при недостаточном объеме исходных данных, когда небольшой объем исходных данных необходимо расщепить на отдельные составляющие с поиском неочевидных зависимостей.

Надо сказать, что это не полный список сверточных слоев, предлагаемых библиотекой Keras. Но описание всех возможностей библиотеки выходит за рамки этой книги. Вы всегда можете с ними ознакомиться на сайте библиотеки. Там же можно найти актуальную версию библиотеки и инструкцию по ее установке и использованию.

Аналогично сверточным слоям библиотека Keras предлагает несколько вариантов классов для подвыборочного слоя. Среди них:

- ***AvgPool1D*** — одномерное усреднение данных;
- ***AvgPool2D*** — двухмерное усреднение данных;

- ***AvgPool3D*** — трехмерное усреднение данных;
- ***MaxPool1D*** — одномерное извлечение максимального значения;
- ***MaxPool2D*** — двухмерное извлечение максимального значения;
- ***MaxPool3D*** — трехмерное извлечение максимального значения.

Все указанные подвыборочные слои имеют одинаковый набор параметров:

- *pool_size* — целое число или вектор целых чисел, определяет размер окна;
- *strides* — целое число или вектор целых чисел, определяет шаг окна;
- *padding* — допускается одно из значений: "*valid*", "*same*" или "*causal*" (без учета регистра); "*valid*" — означает отсутствие отступов; "*same*" — приводит к равномерному дополнению нулями исходных данных для получения размера вывода равного размеру входа;
- *data_format* — допускается одно из значений: "*channels_last*" или "*channels_first*"; определяет в каком измерении входного тензора содержатся данные для свертки; по умолчанию используется "*channels_last*".

Реализацию моделей сверточных нейронных сетей мы также будем осуществлять в нашем шаблоне. Как и при тестировании моделей перцептрона мы создадим три модели нейронной сети с различной архитектурой и сравним результаты их обучения. Поэтому для реализации мы возьмем ранее созданный файл *perceptron.py* и создадим его копию с именем *convolution.py*. В этом созданном файле заменим блоки объявления моделей.

Первым мы создадим перцептрон с тремя скрытыми слоями и регуляризацией матрицы весов. Он будет нам служить базой для сравнения работы сверточных нейронных сетей с результатами обучения полносвязного перцептрона.

```
# Создание модели перцептрона с тремя скрытыми слоями и регуляризацией
model1 = keras.Sequential([keras.Input(shape=inputs),
                           keras.layers.Dense(40, activation=tf.nn.swish,
                    kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(40, activation=tf.nn.swish,
                    kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(40, activation=tf.nn.swish,
                    kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(targerts, activation=tf.nn.tanh)
                         ])
```

Такая модель насчитывает 9802 параметра. Ниже на скриншоте приведена структура созданной нами нейронной сети. В первой колонке таблицы указаны имя и тип нейронного слоя, во второй — размерность тензора результатов каждого слоя. Обратите внимание, что первое измерение не задано, вместо размера указано *None*. Это означает, что данное измерение строго не задано и может быть переменной длины. Данное измерение задается размером пакета данных *batch size*. В третьей колонке указано количество параметров в матрице весов каждого слоя.

Во второй модели мы вставим сразу за исходными данными одномерный сверточный слой *Conv1D* с 8 фильтрами, окно свертки и шаг укажем равными 1. Такой слой будет осуществлять свертку всех заданных индикаторов в рамках одной свечи. При этом не забудем изменить размерность тензора исходных данных с двухмерного на трехмерный.

Обратите внимание: несмотря на то, что мы переносим данные в трехмерный тензор, в параметрах слоя *Reshape* мы указываем два измерения. Это связано с тем, что первое измерение тензора является переменным и задается размером пакета исходных данных *batch size*.

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 40)	6440
dense_1 (Dense)	(None, 40)	1640
dense_2 (Dense)	(None, 40)	1640
dense_3 (Dense)	(None, 2)	82
Total params: 9,802		

Структура перцептрона

И еще один момент. В векторе размерности, передаваемом в параметрах класса *Reshape*, в первом измерении стоит −1. Это указывает классу самостоятельно рассчитать размер данного измерения исходя из размера тензора исходных данных и указанных размерностей других измерений.

```
# Добавляем в модель 1D сверточный слой
model2 = keras.Sequential([keras.Input(shape=inputs),
                           # Переформатируем тензор в трехмерный.
    # Указываем 2 измерения, т.к. 3-е измерение определяется размером пакета
                           keras.layers.Reshape((-1,4)),
                           # Сверточный слой с 8-ю фильтрами
                           keras.layers.Conv1D(8,1,1,activation=tf.nn.swish,
            kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
```

За сверточным слоем мы расположим одномерный подвыборочный слой с выбором максимального значения *MaxPool1D*. Как уже говорилось выше, сверточный слой оперирует с трехмерными тензорами. В то время как последующие полносвязные слои работают с двухмерными тензорами. Поэтому для нормальной работы полносвязных слоев нам необходимо вернуть данные в двухмерную размерность. Для этого мы воспользуется нейронным слоем класса *Flatten*.

```
            # Подвыборочный слой
            keras.layers.MaxPooling1D(2,strides=1),
        # Переформатируем тензор в двухмерный для полносвязных слоев
            keras.layers.Flatten(),
            keras.layers.Dense(40, activation=tf.nn.swish,
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
            keras.layers.Dense(40, activation=tf.nn.swish,
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
            keras.layers.Dense(40, activation=tf.nn.swish,
kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
            keras.layers.Dense(targerts, activation=tf.nn.tanh)
          ])
```

Обратите внимание. В исходных данных каждая свеча описана четырьмя значениями. Использование восьми фильтров увеличивает размерность обрабатываемого тензора. В итоге модель с одномерным сверточным слоем содержит уже 15922 параметра.

В третьей модели мы заменим 1одномерный сверточный слой на двухмерный. Как следствие, изменим подвыборочный слой и размерность данных. Как уже было сказано выше, четвертое измерение зададим равным 1. Теперь мы можем управлять размером окна свертки в двух измерениях: время и индикатор. Так как мы бы хотели оценить различные паттерны в показаниях каждого отдельно взятого индикатора, то в своем сверточном слое мы укажем размер окна свертки по первому временному измерению равным 3 (оцениваем паттерны из 3 последовательных свечей), а размер окна во втором измерении индикаторов равным 1. Это позволит нам выявить закономерности в движении каждого индикатора в отдельности. Шаг окна свертки в обоих направлениях укажем равным 1.

Layer (type)	Output Shape	Param #
reshape (Reshape)	(None, 40, 4)	0
conv1d (Conv1D)	(None, 40, 8)	40
max_pooling1d (MaxPooling1D)	(None, 39, 8)	0
flatten (Flatten)	(None, 312)	0
dense_4 (Dense)	(None, 40)	12520
dense_5 (Dense)	(None, 40)	1640
dense_6 (Dense)	(None, 40)	1640
dense_7 (Dense)	(None, 2)	82
Total params: 15,922		

Структура нейронной сети с одномерным сверточным слоем

С такими параметрами первое измерение (измерение времени) уменьшится на два элемента в результате проведения операций свертки. Второе измерение (измерение индикаторов) останется неизменным, так как окно свертки и его шаг в данном измерении равно 1. В то же время у нас увеличится третье измерение — оно станет равным количеству фильтров. Напомню, что до операции свертки третье измерение было рано 1. В результате всех итераций количество параметров сети увеличилось до 50794. Структура новой нейронной сети приведена ниже. Как можно заметь, сверточный слой имеет только 32 параметра. Такой рост количества параметров сети вызван увеличением размера тензора после операции свертки по причинам, указанным выше. Это видно по количеству параметров в первом после свертки полносвязном слое.

```
# Заменяем в модели сверточный слой на двухмерный
model3 = keras.Sequential([keras.Input(shape=inputs),
                           # Переформатируем тензор в четырехмерный.
          #Указываем 3 измерения, т.к. 4-е измерение определяется размером пакета
                           keras.layers.Reshape((-1,4,1)),
                           # Сверточный слой с 8 фильтрами
                           keras.layers.Conv2D(8,(3,1),1,activation=tf.nn.swish,
                  kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           # Подвыборочный слой
                           keras.layers.MaxPooling2D((2,1),strides=1),
                     # Переформатируем тензор в двухмерный для полносвязных слоев
                           keras.layers.Flatten(),
                           keras.layers.Dense(40, activation=tf.nn.swish,
                  kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(40, activation=tf.nn.swish,
                  kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(40, activation=tf.nn.swish,
                  kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(targerts, activation=tf.nn.tanh)
                         ])
```

Layer (type)	Output Shape	Param #
reshape_1 (Reshape)	(None, 40, 4, 1)	0
conv2d (Conv2D)	(None, 38, 4, 8)	32
max_pooling2d (MaxPooling2D)	(None, 37, 4, 8)	0
flatten_1 (Flatten)	(None, 1184)	0
dense_8 (Dense)	(None, 40)	47400
dense_9 (Dense)	(None, 40)	1640
dense_10 (Dense)	(None, 40)	1640
dense_11 (Dense)	(None, 2)	82

Total params: 50,794

Структура нейронной сети с двухмерным сверточным слоем

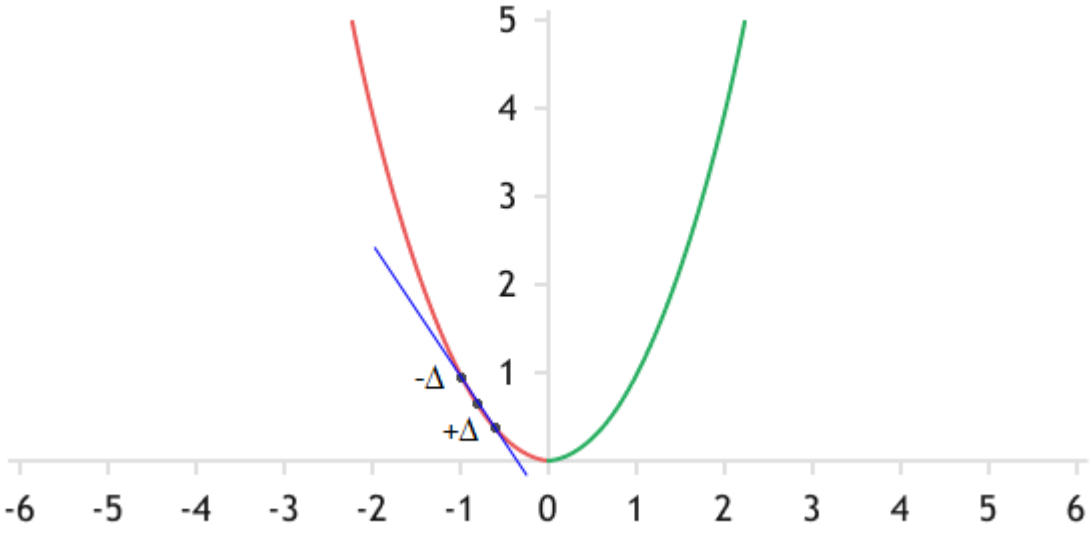
В остальном наш скрипт останется без изменений. С результатами работы скрипта познакомимся в следующем разделе.

4.1.5 Практическое тестирование сверточных моделей

Посмотрите, какой объем работы мы уже выполнили. Нам есть чем гордиться. Мы с вами уже реализовали три типа нейронных слоев, которые уже позволяют решать некоторые практические задачи. С их помощью мы можем создавать полносвязные перцептроны различной сложности. А можем создать модели сверточных нейронных сетей и сравнить результаты работы двух моделей на одном наборе исходных данных.

Но прежде чем приступить к оценке практических возможностей различных моделей нейронных сетей, мы должны проверить корректность работы методов распределения градиента ошибки по сверточной нейронной сети. Мы уже проводили такую процедуру для полносвязных нейронных слоев в разделе «Проверка корректности распределения градиента».

Напомню суть процедуры. Градиент ошибки представляет собой число, определяющее уровень наклона касательной линии к графику функции в текущей точке. Он демонстрирует, как изменится значение функции при изменении параметра.



Геометрический смысл градиента — это наклон касательной к графику функции в текущей точке

Конечно, мы имеем дело с нелинейными функциями, и аналитически вычисленный градиент ошибки дает лишь приблизительное значение. Но при использовании достаточно малого шага изменения параметров такая ошибка становится минимальной.

Кроме того, мы всегда можем определить изменение значения функции при изменении одного параметра экспериментальным путем: можем просто взять нашу функцию, изменить только один параметр и вычислить новое значение. Разница между двумя значениями функции и покажет влияние анализируемого параметра на общий результат в текущей точке.

Разумеется, с ростом количества параметров растут и затраты на оценку влияния каждого параметра на общий результат. Поэтому, пренебрегая небольшой погрешностью, все используют аналитический способ определения градиента ошибки. В то же время, используя экспериментальный метод, мы можем оценить точность работы реализованного нами алгоритма аналитического и скорректировать его работу в случае необходимости.

При сравнении результатов аналитического и экспериментального методов определения градиентов ошибки следует учесть один момент. Для построения прямой на плоскости

необходимы две точки. Но если мы построим прямую через текущую и новую точку, то такая прямая не будет касательной к графику функции в текущей точке. Скорее всего она будет касательная в некой точке между текущим и будущим положением. Поэтому, чтобы построить касательную к графику функции в текущей точке, нужно будет увеличить и уменьшить текущее значение показателя на одно и тоже малое число и вычислить значение функции в обоих точках. Тогда линия будет касаться функции в нужной нам точке, а влияние параметра на значение функции будет средним ее между двумя отклонениями.

При анализе отклонений градиента ошибки между методами в полносвязном слое мы создали скрипт *check_gradient_percp.mq5*. Сделаем копию скрипта с именем *check_gradient_conv.mq5*. В полученной копии мы изменим только функцию *CreateNet*. В ней мы после слоя исходных данных добавим один сверточный слой и один подвыборочный слой.

```
bool CreateNet(CNet &net)
  {
   CArrayObj *layers = new CArrayObj();
   if(!layers)
     {
      PrintFormat("Error creating CArrayObj: %d", GetLastError());
      return false;
     }
//--- слой исходных данных
   CLayerDescription *descr = new CLayerDescription();
   if(!descr)
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      delete layers;
      return false;
     }
   descr.type = defNeuronBase;
   int prev_count = descr.count = BarsToLine;
   descr.window = 0;
   descr.activation = AF_NONE;
   descr.optimization = None;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete layers;
      delete descr;
      return false;
     }
```

Сверточный слой будет состоять из двух фильтров. Размер окна свертки равен двум, его шаг укажем равным единице. Функция активации — *Swish*. Метод оптимизации не имеет значения, так как на данном этапе мы не будем осуществлять обучение нейронной сети. Размер одного фильтра пересчитаем исходя из размера предыдущего слоя и параметров свертки.

```
//--- Сверточный слой
   descr = new CLayerDescription();
   if(!descr)
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      delete layers;
      return false;
     }
   descr.type = defNeuronConv;
   int m_iWindow = descr.window = 2;
   int prev_wind_out = descr.window_out = 2;
   int m_iStep = descr.step = 1;
   prev_count=descr.count=(prev_count-descr.window+2*descr.step-1)/descr.step;
   descr.activation = AF_SWISH;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete layers;
      delete descr;
      return false;
     }
```

После сверточного нейронного слоя расположим подвыборочный слой. Для него мы укажем окна равным двум и шаг — единице. Функцию активации укажем *AF_AVERAGE_POOLING*, что соответствует вычислению среднего значения для каждого окна исходных данных.

```
//--- Подвыборочный слой
   descr = new CLayerDescription();
   if(!descr)
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      delete layers;
      return false;
     }
   descr.type = defNeuronProof;
   descr.window = 2;
   descr.window_out = prev_wind_out;
   descr.step = 1;
   descr.count = (prev_count - descr.window + 2 * descr.step - 1) / descr.step;
   descr.activation = (ENUM_ACTIVATION_FUNCTION)AF_AVERAGE_POOLING;
   descr.optimization = None;
   descr.activation_params[0] = 1;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete layers;
      delete descr;
      return false;
     }
```

Далее код скрипта остается неизменным.

Подготовленный скрипт запускаем в двух режимах: с использованием технологии *OpenCL* и без. В результате тестирования получили довольно приемлемые результаты. В обоих случаях получили отклонения в 11-м знаке после запятой.

```
Use OpenCL true
Delta at input gradient between methods 1.48910e-11
Delta at weights gradient between methods -1.51489e-11
Use OpenCL false
Delta at input gradient between methods 3.91087e-11
Delta at weights gradient between methods -1.36456e-12
```

Результаты проверки отклонений градиента ошибки между аналитическим и экспериментальным методами определения

Теперь, когда мы уверены в корректности работы наших классов нейронных слоев, можно приступать к построению и обучению сверточных нейронных сетей. Для начала нам нужно определиться каким образом мы хотим использовать свертку.

В нашем наборе данных каждая свеча представлена несколькими показателями. В частности, при создании обучающей выборки для каждой свечи мы определили четыре показателя:

- RSI,
- гистограмма MACD,
- сигнальная линия MACD,
- отклонение между сигнальной линией и гистограммой MACD.

Каждый из вас может сам провести ряд тестов и определить свой подход к использованию сверточных моделей. Для меня наиболее очевидным является два варианта использования.

1. Мы можем использовать свертку, чтобы на уровне каждой свечи определить некие паттерны из показателей индикаторов. В таком варианте мы определяем количество искомых паттернов в виде количества фильтров свертки. На выходе сверточного слоя мы получаем степень схожести каждой свечи с искомыми паттернами.
2. Следует помнить, что полносвязный нейронный слой является линейной функцией. Только функция активации добавляет нелинейности. Поэтому в общем случае нейроны не оценивают зависимости между элементами исходных данных, а обучаются на определение паттернов из набора исходных данных. Поэтому каждый нейрон оценивает текущий паттерн без оглядки на прошлые данные.

Но при анализе временных рядов порой бывает динамика изменения показателя имеет большее значение, чем его абсолютное значение. Мы можем использовать свертку для определения паттернов динамики показателей. Для этого нам потребуется немного переставить показатели, выстроив в один ряд последовательно значения каждого показателя. К примеру, сначала выстроим в буфере исходных данных все показатели индикатора RSI. Затем — все элементы последовательности гистограммы MACD, за ними — данные сигнальной линии. Завершим буфер данными отклонений между сигнальной линией и гистограммой MACD. Конечно, было бы нагляднее расположить данные в табличной форме, где каждая строка представляла бы значения отдельного индикатора. Но, к сожалению, в контексте OpenCL используются только одномерные буферы. Поэтому мы воспользуемся виртуальным делением буфера на блоки.

После выстраивания каждого показателя в отдельный ряд, мы можем с помощью свертки определять паттерны в последовательных значениях одного показателя. Тем самым мы как бы определяем тенденции внутри анализируемого окна данных. Количество фильтров сверточного слоя определит количество тенденций к распознаванию моделью.

Тестирование свертки в рамках одной свечи

Для проведения тестирования работы сверточных моделей нейронных сетей создадим копию скрипта *perceptron_test.mq5* с именем *convolution_test.mq5*. В начале скрипта по-прежнему мы указываем параметры для работы скрипта.

Как и при проверке корректности распределения градиента, нам достаточно изменить лишь функцию описания архитектуры модели CreateLayersDesc. В нем мы после слоя исходных данных добавляем сверточный и подвыборочный нейронные слои.

```
bool CreateLayersDesc(CArrayObj &layers)
  {
   CLayerDescription *descr;
//--- создаем слой исходных данных
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   descr.type         = defNeuronBase;
   descr.count        = NeuronsToBar * BarsToLine;
   descr.window       = 0;
   descr.activation   = AF_NONE;
   descr.optimization = None;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      return false;
     }
```

Обратите внимание, что для сверточного слоя в поле *count* объекта описания слоя мы указываем не общее количество нейронов, а количество элементов в одном фильтре. В поле *window_out* указываем количество используемых фильтров. В полях *window* и *step* укажем количество элементов на один бар. С такими параметрами мы получим свертку без перекрытия, и каждый фильтр будет сравнивать состояние индикаторов на каждом баре с неким паттерном. Функцию активации я указал *Swish*, а метод оптимизации — *Adam*. Данный метод оптимизации мы будем использовать и для всех последующих слоев. Кроме, конечно, подвыборочного, который не содержит матрицы весов.

```
//--- Сверточный слой
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   descr.type = defNeuronConv;
   descr.count = BarsToLine;
   descr.window = NeuronsToBar;
   descr.window_out = 8;
   descr.step = NeuronsToBar;
   descr.activation = AF_SWISH;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
```

За сверточным слоем следует подвыборочный слой. В данной реализации я использовал *Max Pooling* — выбор максимального элемента в пределах входного окна. Мы используем скользящее

окно из двух элементов и шагом в один элемент. С таким набором параметров количество элементов в одном фильтре уменьшится на один. Функцию активации на данном слое мы не используем. Количество фильтров равно аналогичному параметру предыдущего слоя.

```
//--- Подвыборочный слой
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   descr.type = defNeuronProof;
   descr.count = BarsToLine - 1;
   descr.window = 2;
   descr.window_out = 8;
   descr.step = 1;
   descr.activation = (ENUM_ACTIVATION_FUNCTION)AF_MAX_POOLING;
   descr.optimization = None;
   descr.activation_params[0] = 0;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
```

Далее идет массив скрытых полносвязных слоев. Их мы будем создавать в цикле с одинаковыми параметрами. Количество создаваемых скрытых слоев задается в параметрах скрипта. Все скрытые слои будут иметь одинаковое количество элементов, которое указывается в параметрах скрипта. Функцию активации будем использовать *Swish*, а метод оптимизации параметров матрицы весов — *Adam*, как и для сверточного слоя.

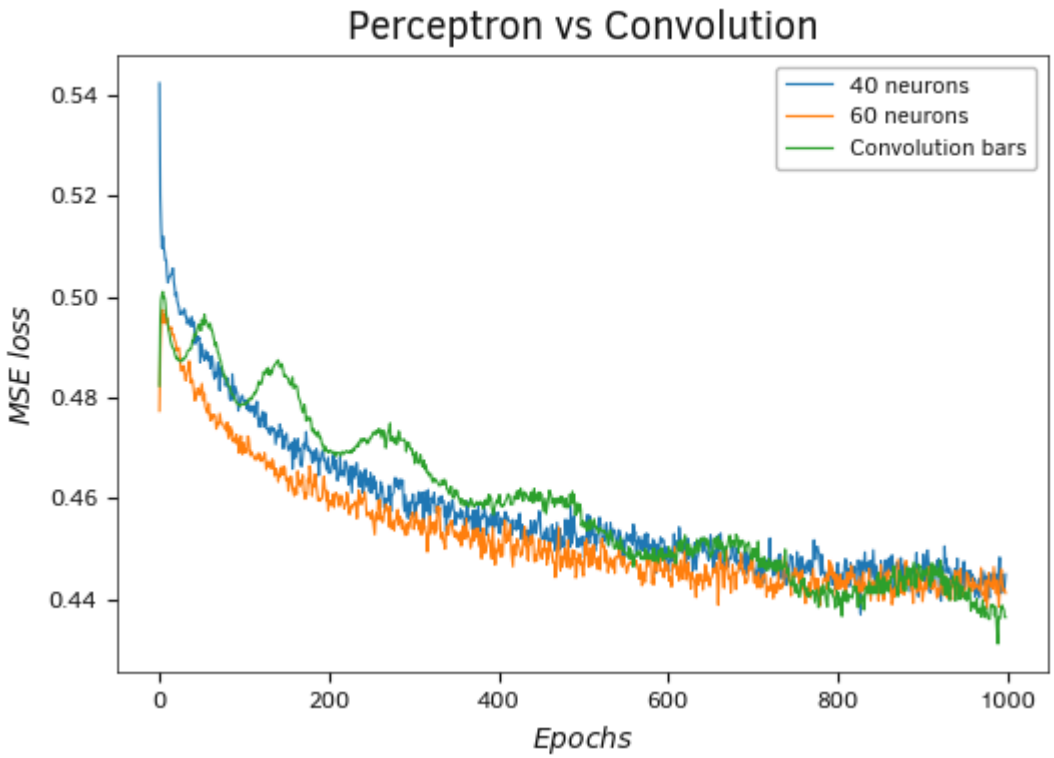
```
//--- Блок скрытых полносвязных слоев
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   descr.type         = defNeuronBase;
   descr.count        = HiddenLayer;
   descr.activation   = AF_SWISH;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
   for(int i = 0; i < HiddenLayers; i++)
      if(!layers.Add(descr))
        {
         PrintFormat("Error adding layer: %d", GetLastError());
         return false;
        }
```

И в завершение функции инициализации нейронной сети укажем параметры выходного слоя. Он у нас будет, как и в созданных ранее моделях перцептрона, содержать два элемента с линейной функцией активации. Будем использовать метод оптимизации *Adam*, как и для всех других нейронных слоев.

```
//---  Слой результатов
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   descr.type         = defNeuronBase;
   descr.count        = 2;
   descr.activation   = AF_LINEAR;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      return false;
     }
   return true;
  }
```

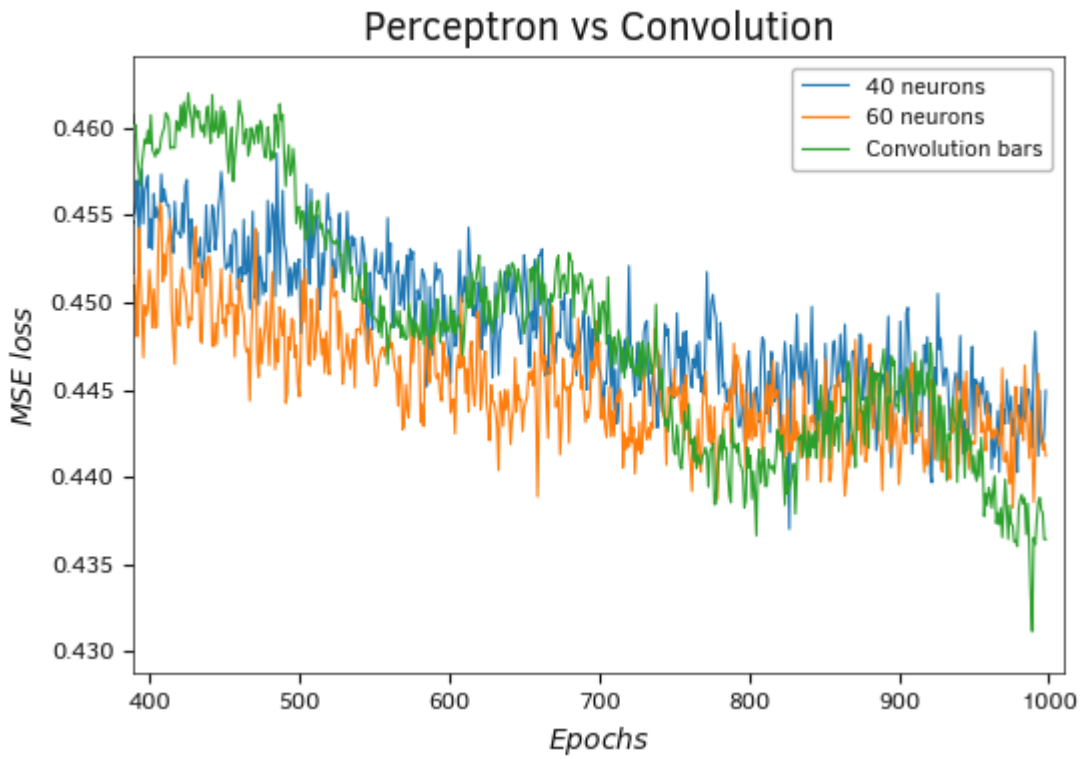
Весь остальной код скрипта остался без изменений.

По результатам тестирования модель сверточной нейронной сети показала менее ровный график. На нем мы наблюдаем волнообразное снижение значения функции ошибки. Но при этом после 1000 итераций обновления матрицы весов мы получили более низкое значение функции потерь.



Сравнительный график обучения перцептрона и свёрточной нейронной сети

При увеличении масштаба можно заметить тенденцию и к потенциальному снижению значения функции потерь при продолжении обучения.



Сравнительный график обучения перцептрона и свёрточной нейронной сети

Тестирование свертки скользящего окна по значениям индикатора

Для эксперимента с поиском паттернов динамики значений индикаторов нам необходимо немного изменить ранее созданный скрипт *convolution_test.mq5*. Создадим его копию с именем *convolution_test2.mq5*. Первые изменения мы внесем в объявление сверточного слоя. На этот раз мы создаем слой с окном свертки равным трем элементам и шагом в один элемент. С такими параметрами количество элементов в одном фильтре будет на два элемента меньше предыдущего слоя, но при этом общее количество элементов выходного буфера увеличится кратно количеству используемых фильтров. Функция активации и метод оптимизации остаются без изменений.

```
//--- Сверточный слой
   int prev_count = descr.count;
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   descr.type = defNeuronConv;
   prev_count = descr.count = prev_count - 2;
```

```
   descr.window = 3;
   descr.window_out = 8;
   descr.step = 1;
   descr.activation = AF_SWISH;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
```

В подвыборочном слое изменения коснулись только количества элементов в одном фильтре.

```
//--- Подвыборочный слой
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   descr.type = defNeuronProof;
   descr.count = prev_count - 1;
   descr.window = 2;
   descr.window_out = 8;
   descr.step = 1;
   descr.activation = (ENUM_ACTIVATION_FUNCTION)AF_MAX_POOLING;
   descr.optimization = None;
   descr.activation_params[0] = 0;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
```

Как уже было сказано выше, для этого теста нам нужно изменить последовательность исходных данных, подаваемых на вход нейронной сети. Поэтому нам потребовалось внести изменения в функцию загрузки обучающей выборки из файла *LoadTrainingData*.

Как и ранее, в начале функции проводим подготовительную работу. Объявляем экземпляры необходимых локальных объектов, открываем для считывания файл обучающей выборки. Имя и путь файла указываются в параметрах функции. Напомню, что файл с обучающей выборкой должен находится в пределах песочницы вашего терминала.

Результат процедуры открытия файла проверяем по полученному хендлу.

```
bool LoadTrainingData(string path, CArrayObj &data, CArrayObj &result)
  {
   CBufferType *pattern;
   CBufferType *target;
//--- открываем файл с обучающей выборкой
   int handle = FileOpen(path, FILE_READ | FILE_CSV | FILE_ANSI | FILE_SHARE_READ,
                                                                  ",", CP_UTF8);
   if(handle == INVALID_HANDLE)
     {
      PrintFormat("Error opening study data file: %d", GetLastError());
      return false;
     }
//--- выводим прогресс загрузки данных обучения в комментарий чарта
   uint next_comment_time = 0;
   uint OutputTimeout = 250; // не чаще 1 раза в 250 миллисекунд
```

После успешного открытия для чтения файла обучающей выборки запускаем цикл непосредственной загрузки данных. Итерации цикла будем повторять до завершения файла. При этом перед каждой итерацией будем проверять не поступила ли команда на закрытие программы.

В теле цикла мы сначала подготовим новые экземпляры объектов для загрузки паттерна и целевых результатов.

```
//--- организовываем цикл загрузки обучающей выборки
   while(!FileIsEnding(handle) && !IsStopped())
     {
      if(!(pattern = new CBufferType()))
        {
         PrintFormat("Error creating Pattern data array: %d", GetLastError());
         return false;
        }
      if(!pattern.BufferInit(NeuronsToBar, BarsToLine))
        {
         delete pattern;
         return false;
        }
      if(!(target = new CBufferType()))
        {
         PrintFormat("Error creating Pattern Target array: %d", GetLastError());
         delete pattern;
         return false;
        }
      if(!target.BufferInit(1, 2))
        {
         delete pattern;
         delete target;
         return false;
        }
```

Для загрузки данных мы по прежнему используем динамические массивы:

- ***data*** — массив паттернов исходных данных;

- ***result*** — массив паттернов целевых значений для каждого паттерн;
- ***pattern*** — буфер элементов одного паттерна;
- ***target*** — буфер целевых значений одного паттерна.

Но для того, чтобы изменить последовательность загружаемых данных, мы сначала изменим размер матрицы буфера *pattern* таким образом, что первое столбцы матрицы будет соответствовать количеству используемых индикаторов, а строки — количеству анализируемых исторических баров.

Мы организуем систему вложенных циклов. Внешний цикл имеет число итераций равное количеству анализируемых свечей. Количество итераций во внутреннем цикле равно числу элементов на одну свечу. В теле этой системы циклов мы будем записывать исходные данные в матрицу буфера *pattern*. Так как в файле обучающей выборки данные расположены в хронологической последовательности, то и мы будем записывать их в той же последовательности. Но по мере считывания будем распределять информацию по соответствующим строкам и столбцам матрицы.

```
for(int i = 0; i < BarsToLine; i++)
   for(int y = 0; y < NeuronsToBar; y++)
      pattern.m_mMatrix[y, i] = (TYPE)FileReadNumber(handle);
```

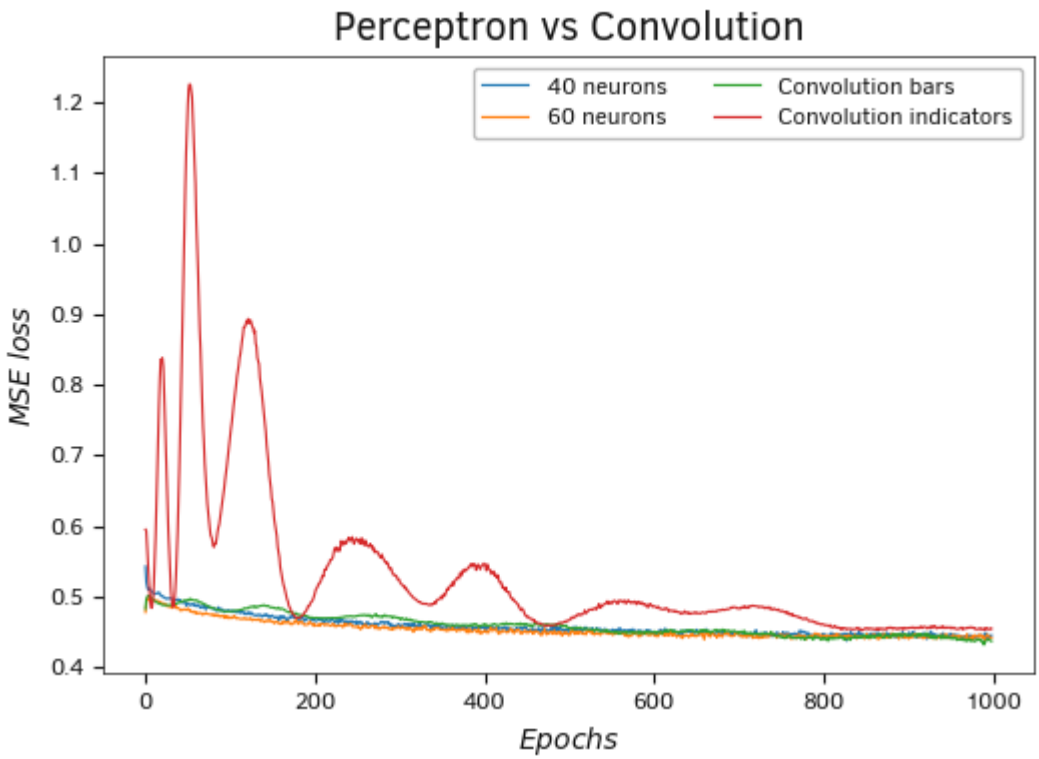
После завершения итераций системы циклов нам достаточно лишь переформатировать полученную матрицу.

```
if(!pattern.Reshape(1, BarsToLine * NeuronsToBar))
  {
   delete pattern;
   delete target;
   return false;
  }
```

Дальнейший процесс загрузки обучающей выборки перенесен без изменений.

```
      for(int i = 0; i < 2; i++)
         target.m_mMatrix[0, i] = (TYPE)FileReadNumber(handle);
      if(!data.Add(pattern))
        {
         PrintFormat("Error adding study data to array: %d", GetLastError());
         delete pattern;
         delete target;
         return false;
        }
      if(!result.Add(target))
        {
         PrintFormat("Error adding study data to array: %d", GetLastError());
         delete target;
         return false;
        }
      //--- выводим прогресс загрузки в комментарий чарта (не чаще 1 раза в 250 милли
      if(next_comment_time < GetTickCount())
        {
         Comment(StringFormat("Patterns loaded: %d", data.Total()));
         next_comment_time = GetTickCount() + OutputTimeout;
        }
     }
   FileClose(handle);
   return true;
  }
```

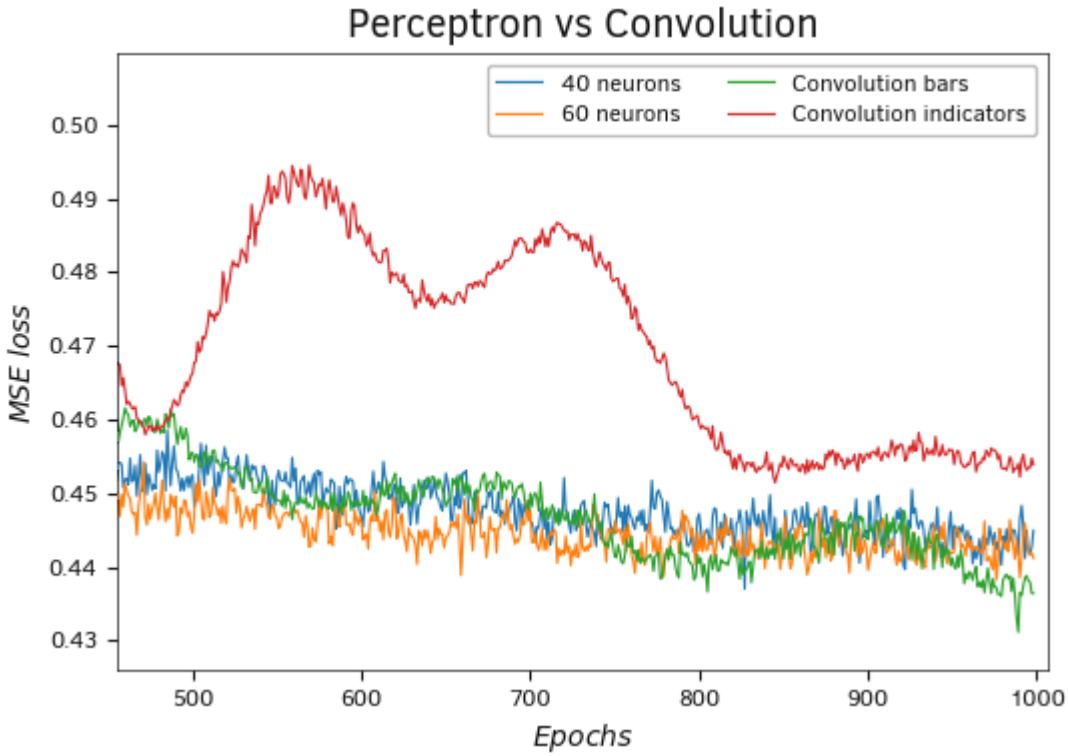
В результате обучения такая нейронная сеть продемонстрировала еще большую амплитуду волн в динамике изменения значений функции потерь. В процессе обучения амплитуда волн была снижена. Такое поведение может говорить об использовании завышенного коэффициента обучения, влияние которого было снижено методом обучения Adam. Напомню, что данный метод обучения использует алгоритм индивидуальной адаптации коэффициента обучения для каждого элемента матрицы весов.



Сравнительный график обучения перцептрона и двух моделей свёрточных нейронных сетей

Но вернемся к результатам нашего теста. К сожалению, в данном случае наши изменения модели не дали желаемого снижения ошибки модели. Напротив, она даже возросла. Тем не менее есть надежда на улучшение результатов при снижении коэффициента обучения.

Увеличение масштаба графика подтверждает сделанные выше выводы.



Сравнительный график обучения перцептрона и двух моделей свёрточных нейронных сетей

Комбинированная модель

Мы посмотрели на работу двух моделей сверточных нейронных сетей. В первой модели мы осуществляли свертку значений индикаторов в рамках одной свечи. Во второй модели мы транспонировали (перевернули) исходные данные и провели свертку в разрезе значений индикаторов. При этом в первом случае мы осуществляли свертку сразу всех индикаторов в рамках одного бара. Напомню, для каждого бара мы берем четыре значения от двух индикаторов. Во второй модели мы использовали свертку скользящим окном из трех баров и шагом окна свертки в один бар. И возникает очевидный вопрос: каких результатов можно добиться, если совместить оба подхода? Ответ на этот вопрос нам даст еще один эксперимент.

Для проведения этого тестирования нам потребуется построение еще одной модели. На практике создание такой модели у меня не забрало много времени. Давайте рассуждать. В первом случае мы брали значения индикаторов для каждой свечи, во втором случае — по три последовательных значения (три последовательных бара) каждого отдельного индикатора. Если мы хотим объединить два подхода, то, наверное, логично будет взять для свертки все значения для трех последовательных баров. В обоих подходах мы использовали шаг в один бар. Следовательно, такой шаг мы и сохраним.

Для построения такой модели нам нет необходимости транспонировать данные. Поэтому новую модель будем строить на базе скрипта *convolution_test.mq5*. В начале мы создадим его копию с именем *convolution_test3.mq5*. В нем мы изменим параметры сверточного слоя. В обучающей выборке данные идут в хронологическом порядке, следовательно, окно свертки из полных трех баров будет равно *3 * NeuronsToBar*. Тогда шаг окна свертки в размере одного бара будет равен *NeuronsToBar*. С такими параметрами количество элементов в одном фильтре составит *BarsToLine - 2*. Функцию активации и метод оптимизации параметров оставляем без изменений.

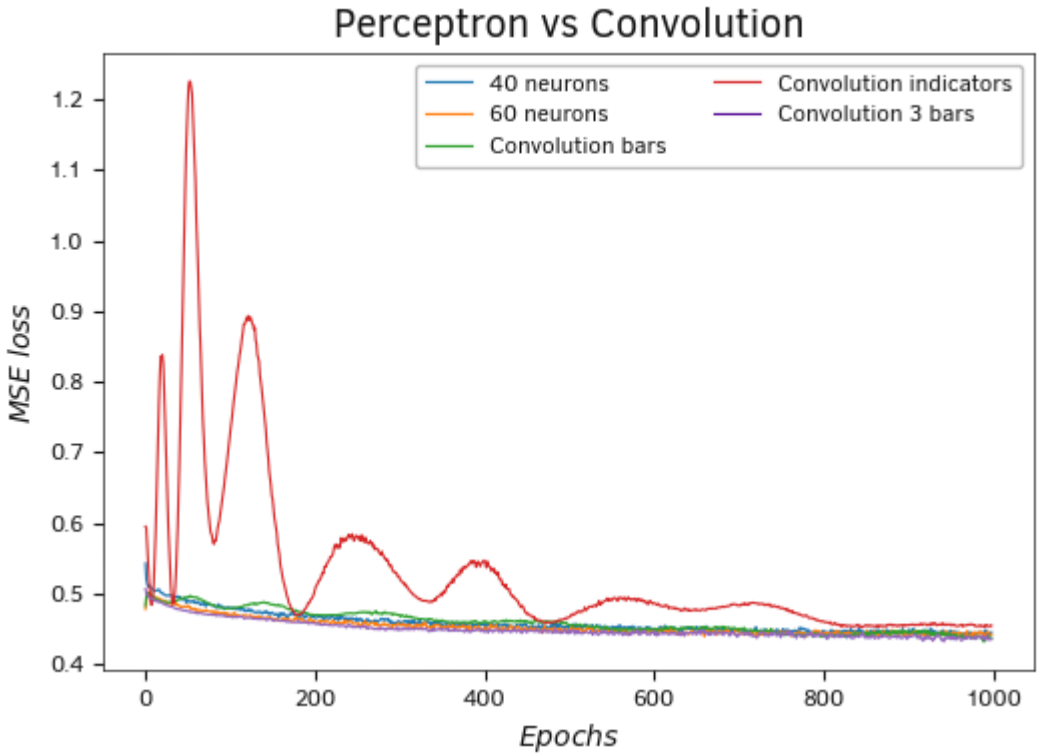
```
//--- Сверточный слой
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   descr.type = defNeuronConv;
   descr.count = BarsToLine - 2;
   descr.window = 3 * NeuronsToBar;
   descr.window_out = 8;
   descr.step = NeuronsToBar;
   descr.activation = AF_SWISH;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
```

Изменения, внесенные в параметры сверточного слоя, потребовали небольшую правку параметров подвыборочного слоя. Здесь мы внесли лишь изменения в количество элементов в одном фильтре.

```
//--- Подвыборочный слой
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   descr.type = defNeuronProof;
   descr.count = BarsToLine - 3;
   descr.window = 2;
   descr.window_out = 8;
   descr.step = 1;
   descr.activation = (ENUM_ACTIVATION_FUNCTION)AF_MAX_POOLING;
   descr.optimization = None;
   descr.activation_params[0] = 0;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
```

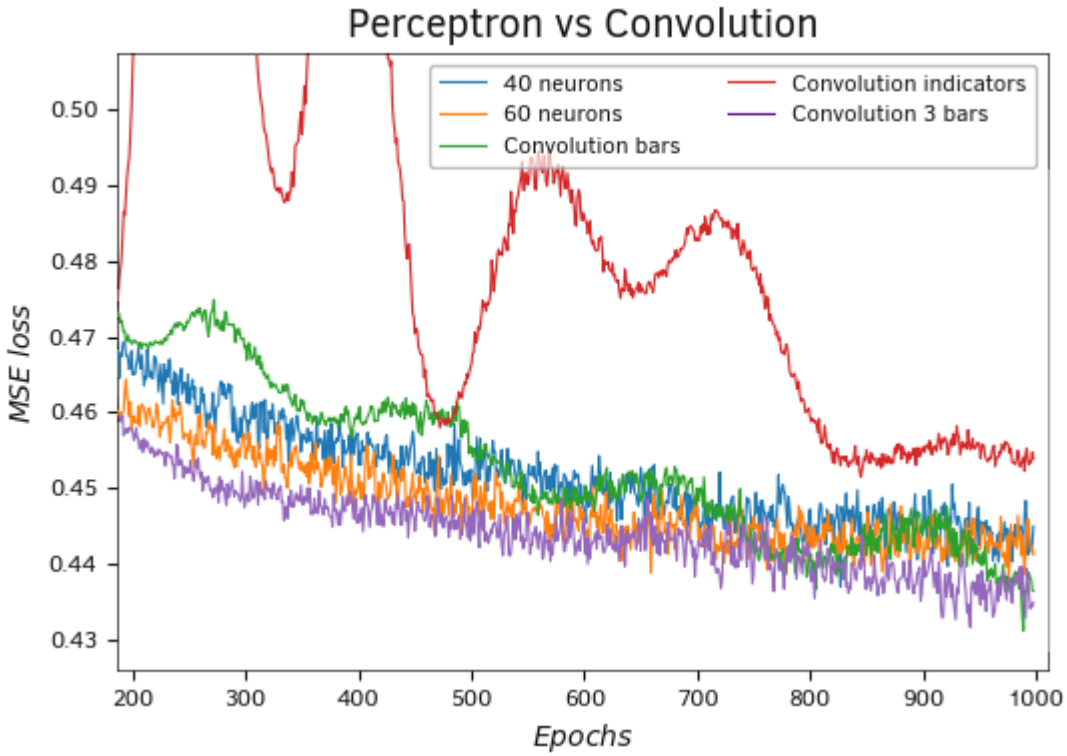
Весь остальной код скрипта остался без изменений.

Результаты обучения новой модели оказались лучше всех предыдущих. График динамики значений функции потерь без амплитудных волн и лежит немного ниже графиков всех ранее проведенных тестов.



Сравнительный график обучения перцептрона и трех моделей свёрточных нейронных сетей

Увеличение масштаба графика только подтверждает сделанные выше выводы.

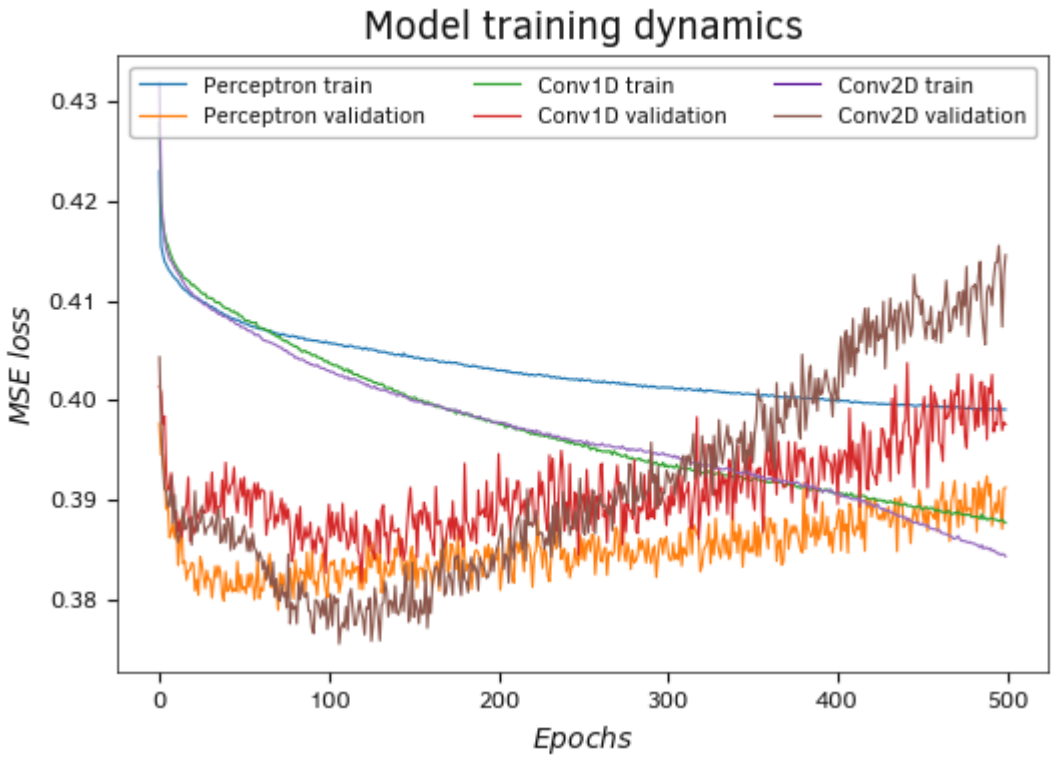


Сравнительный график обучения перцептрона и трех моделей свёрточных нейронных сетей

Тестирование моделей Python

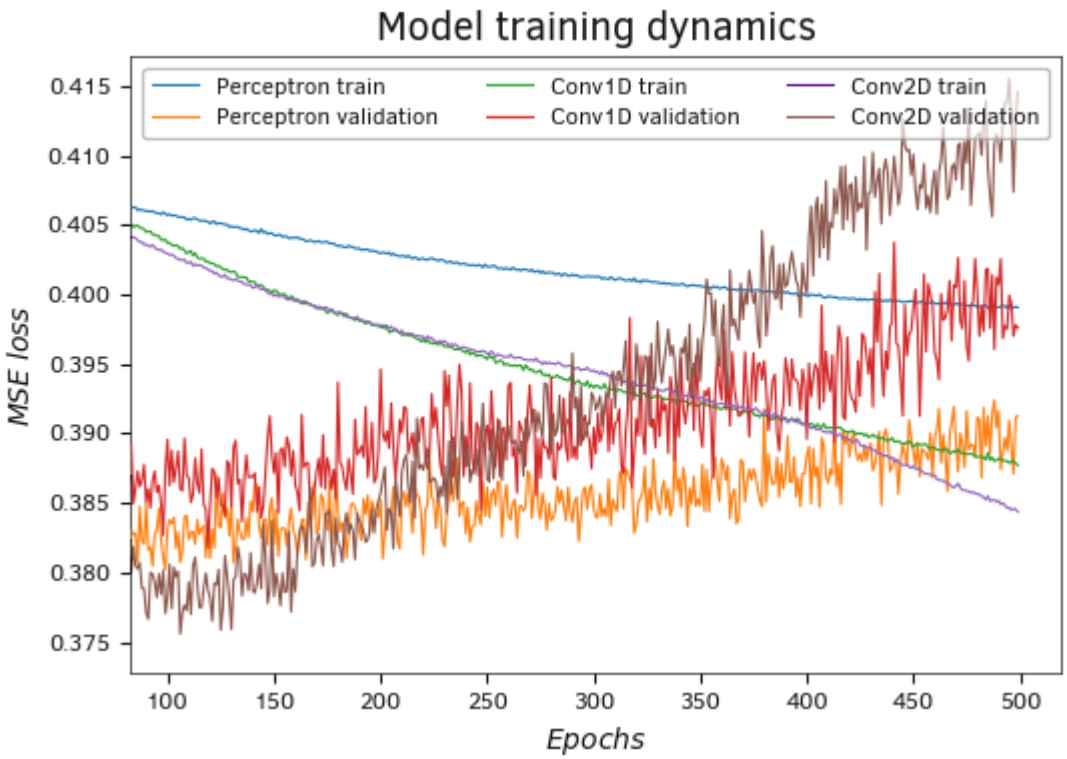
В предыдущем разделе мы создали скрипт с тремя моделями нейронных сетей на языке Python. Напомню, первая модель перцептрона имеет три скрытых слоя, во второй мы добавили сверточный нейронный слой *Conv1D* перед моделью перцептрона, а в третьей модели заменили сверточный слой *Conv1D* на *Conv2D*. При этом в каждой последующей модели увеличивалось число параметров. По логике работы созданные нами модели на языке Python повторяют проведенные выше эксперименты с моделями нейронных сетей, созданных средствами MQL5. Поэтому результаты тестирования были вполне ожидаемые и полностью подтвердили сделанные ранее выводы. Для нас это является дополнительным подтверждением корректности работы нашей библиотеки, написанной на языке MQL5. А значит, мы вполне можем использовать ее в нашей дальнейшей работе. Кроме того, получение схожих результатов при тестировании моделей, полностью созданных различными средствами, исключает случайность полученных результатов и минимизирует вероятность допущения ошибки в процессе создания моделей.

Но вернемся к результатам тестирования. В процессе обучения модель со сверточным слоем *Conv2D* показала наилучшие результаты по снижению ошибки, что полностью подтверждает полученные выше результаты. Значительный разрыв между графиками динамики ошибки на обучающей и валидационной выборке у перцептрона может свидетельствовать о недообученности нейронной сети.



Сравнительный график обучения перцептрона и 2-х моделей свёрточных нейронных сетей (Python)

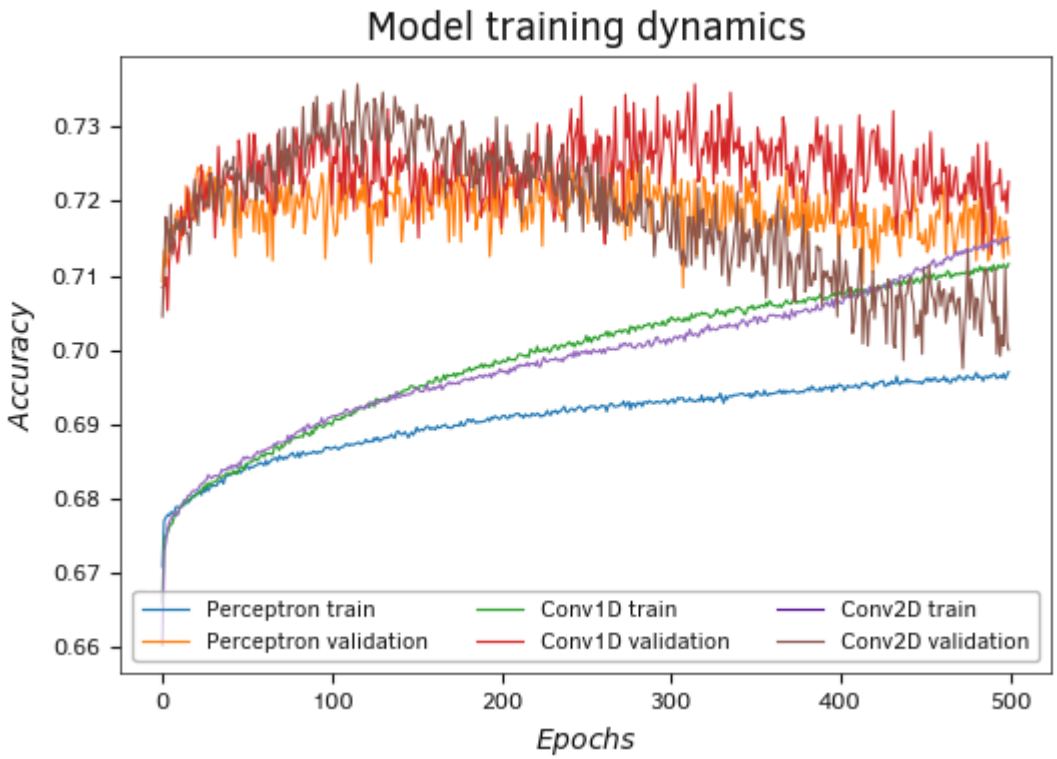
Динамика ошибки сверточных моделей очень близка. Их графики практически идут параллельно. Но все же модель с *Conv2D* сверточным слоем демонстрирует меньшую ошибку на протяжении всего обучения.



Сравнительный график обучения перцептрона и двух моделей свёрточных нейронных сетей (Python zoom)

На валидационной выборке графики ошибки сверточной модели *Conv2D* сначала снижается, но после 100 эпох обучения наблюдается рост ошибки. Наряду со снижением ошибки на обучающей выборке это может свидетельствовать о склонности моделей к переобучению.

График второй метрики обучения *Accuracy* демонстрирует схожие результаты.



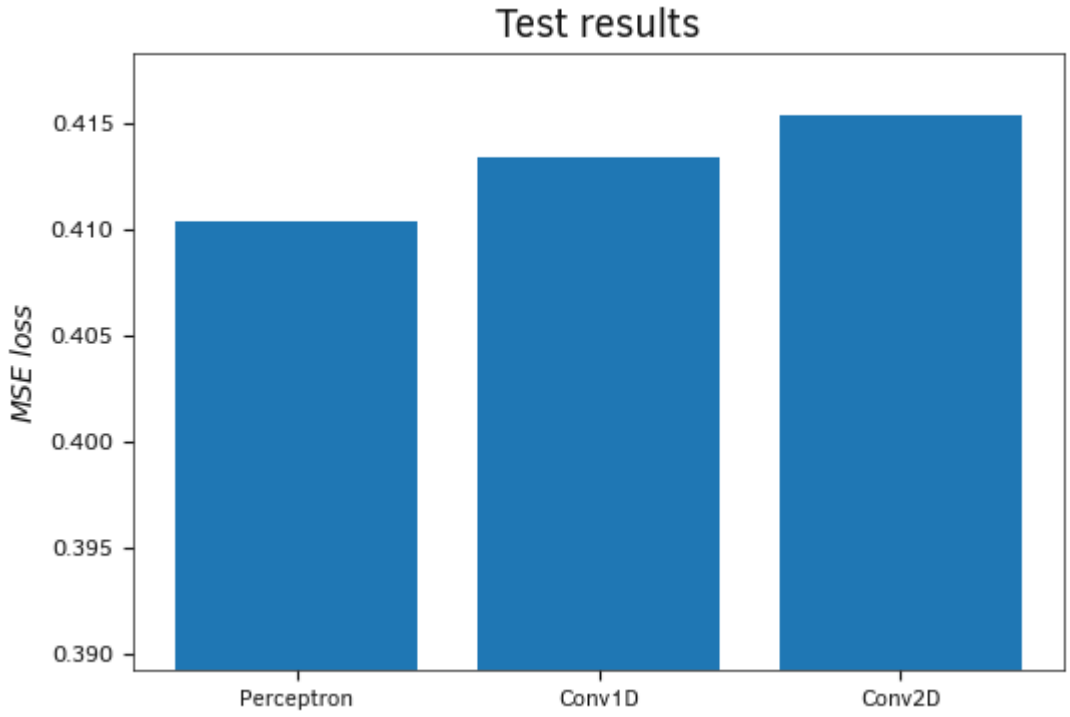
Сравнительный график обучения перцептрона и двух моделей свёрточных нейронных сетей (Python)

На валидационной выборке графики всех трех моделей тесно переплетены в диапазоне 0,71–0,73. На графике видно пересечение графиков обучающей и валидационной выборки после 400.

Напомню, валидационная выборка значительно меньше обучающей, для нее берутся последние паттерны без перемешивания общей выборки. Поэтому с большой долей вероятности в валидационную выборку попадут не все возможные паттерны. К тому же валидационная выборка может быть подвержена влиянию локальных тенденций.

Проверка работы всех трех обученных моделей на тестовой выборке показала довольно схожие, хотя и немного противоречивые результаты.

Тестирование среднеквадратичной ошибки моделей показала наилучшие результаты у сверточной модели со сверткой *Conv2D*. Данная модель анализирует паттерны в рамках одного индикатора со скользящим окном свертки, на обучении она была лучшей. Конечно, разрыв показателей не столь существенный, и можно считать, что все модели показали равные результаты.



Сравнение ошибки моделей на тестовой выборке

Сравнение Accuracy моделей на тестовой выборке

Сравнение результатов по метрике *Accuracy*, в противоречие только что рассмотренного графика *MSE*, демонстрирует наилучшие результаты у модели Conv1D. Модель анализирует паттерны

каждой отдельной свечи, наименьший результат — у перцептрона. Но, как и по MSE, разрыв между результатами небольшой.

Предлагаю считать, что на обучающей выборке все три модели показали приблизительно равные результаты. Точные значения метрик на тестовой выборке приведены на скриншоте ниже.

	Perceptron model
	Test accuracy: 0.6872221827507019
	Test loss: 0.41034841537475586
	Conv1D model
	Test accuracy: 0.6905249953269958
	Test loss: 0.41335004568099976
	Conv2D model
	Test accuracy: 0.6902002692222595
	Test loss: 0.4153774380683899

Точные значения проверки моделей на тестовой выборке

Выводы

По результатам проведенных тестов можно сказать:

- Построенные нами модели средствами *MQL5* при обучении демонстрируют результаты аналогичные моделям построенными с помощью библиотеки *Keras* на языке *Python*. Данный факт подтверждает корректность работы создаваемой нами библиотеки. Мы уверенно можем продолжить нашу работу.
- В общем итоге сверточные модели позволяют улучшить результаты работы модели на той же обучающей выборке.
- Подходы к свертке исходных данных могут быть различные, и от выбранного подхода могут зависеть результаты работы модели.
- Объединение различных подходов в рамках одной модели не всегда позволяет улучшить результаты работы модели.
- Не бойтесь экспериментировать. Создавая свою модель, попробуйте различные архитектуры и различные варианты обработки данных.

В своих тестах мы использовали лишь один сверточный и один подвыборочный слой. Это можно назвать подходом построения простых моделей. Наиболее успешные сверточные модели, применяемые для решения практических задач, используют несколько комплектов сверточных и подвыборочных слоев. При этом в каждом комплекте меняется размерность окна свертки и количество фильтров. Как я уже сказал, не бойтесь экспериментировать. Только сравнив работу различных моделей вы сможете выбрать лучшую архитектуру для решения вашей задачи.

4.2 Рекуррентные нейронные сети

Ранее мы уже рассмотрели многослойный перцептрон и сверточные нейронный сети. Все они работают со статичными данными в рамках марковских процессов, когда последующее состояние системы зависит только от ее текущего состояния и не зависит от состояния системы в прошлом. Да, чтобы компенсировать этот недостаток, мы подавали на вход нейронной сети не только последние ценовые данные и состояния индикаторов, но и исторические данные за несколько последних баров. Но сама сеть не запоминала обработанные данные и полученные

результаты. На каждой новой итерации прямого прохода на вход нейронной сети мы заново подавали полный набор исторических данных, даже если мы их уже передавали нейронной сети ранее. По существу, каждый прямой проход нейронная сеть начинала с «чистого листа». Единственная память, которой обладает такая нейронная сеть, это матрица весов, которую она выучила в процессе обучения.

Еще один недостаток использования таких нейронных сетей для таймсерий заключается в том, что на значения результата абсолютно не влияет положение того или иного значения/паттерна в массиве данных. Напомню математическую модель нейрона.

$$OUT = f\left(\sum_{i=1}^{n} w_i x_i\right)$$

Обратите внимание, что в центре всей модели стоит сумма значений. От перестановки мест слагаемых сумма не меняется, поэтому с точки зрения математики абсолютно не важно, где появляется значение — в начале массива или в конце. В практике же использования таймсерий довольно часто последние значения имеют большее влияние на результат по сравнению с более старыми значениями.

Сейчас я предлагаю посмотреть в сторону рекуррентных нейронных сетей (*Recurrent Neural Network*). Это особый вид нейронных сетей, призванный работать с временными последовательностями. Основной особенностью рекуррентных нейронов является передача своего состояния на вход самому себе в следующей итерации.

$$OUT_t = f\left(\sum_{i=1}^{n} w_i x_i^t + w_{out} OUT_{t-1}\right)$$

Нейрон, получая на вход новое состояние внешней среды, вместе с новыми данными каждый раз как бы оценивает результат своей работы в прошлом.

4.2.1 Описание архитектуры и принципов реализации

Рассмотренные ранее типы нейронных сетей работают с заранее заданным объемом данных. В нашем же случае при работе с графиками цен трудно сказать, каким будет идеальный размер анализируемых данных. Разные закономерности могут проявляться в различные временные интервалы. Да и сами эти интервалы не всегда статичны и могут варьироваться в зависимости от текущей ситуации. Какие-то события могут быть редкими на рынке, но с большой долей вероятности отрабатываться. И хорошо, если такое событие осталось в рамках анализируемого окна. Но как только оно выпадает из него, то нейронная сеть уже не примет его во внимание. Хотя, возможно, именно в этот момент рынок будет отрабатывать реакцию на это событие. Увеличение анализируемого окна ведет к росту потребления вычислительных ресурсов и, как следствие, потребуется больше времени на обучение такой нейронной сети. А в практике реального применения потребуется больше времени на принятие решения.

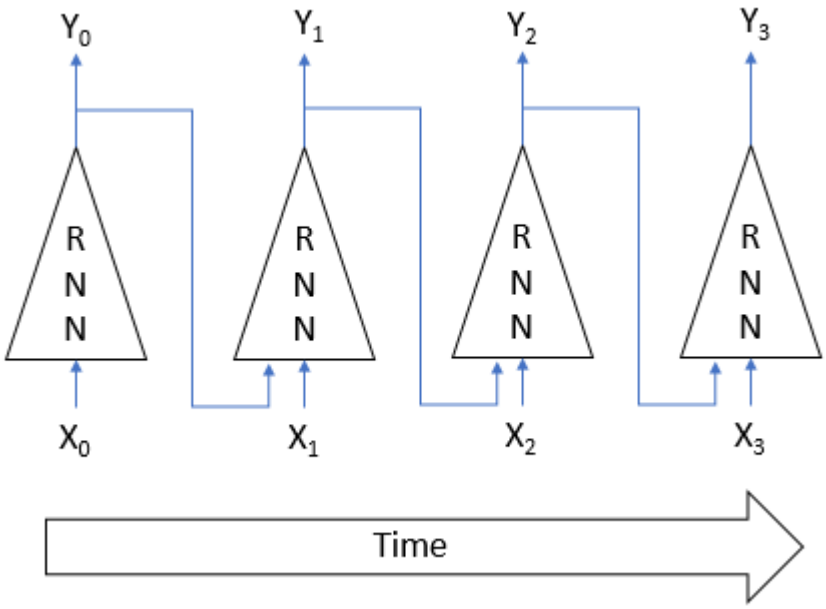
Для решения данной проблемы работы с временными рядами было предложено использование в нейронных сетях рекуррентных нейронов. Это попытка реализации краткосрочной памяти в нейронных сетях, когда на вход нейрона вместе с информацией о текущем состоянии системы подается и предыдущее состояние этого же нейрона. Такое решение основано на предположении о том, что значение на выходе нейрона учитывает влияние всех факторов (в том числе и свое предыдущее состояние) и на следующем шаге передаст все свои знания в свое

будущее состояние. Это схоже с опытом человека, когда новые действия совершаются на основе действий, совершенных ранее. Срок такой памяти и ее влияние на текущее состояния нейрона будет зависеть от весовых коэффициентов.

Здесь может быть использована любое архитектурное решение нейрона, в том числе и рассмотренные нами ранее полносвязные и сверточные слои. Мы лишь сцепляем два тензора: исходных данных и результатов предыдущей итерации, и подаем полученный тензор на вход нейронного слоя. В начале работы нейронной сети, когда еще нет тензора результатов с предыдущей итерации, недостающие элементы заполняются нолями.

Схема рекуррентного нейрона

Обучение рекуррентных нейронных сетей осуществляется уже хорошо известным нам методом обратного распространения ошибки. По аналогии с обучением сверточных нейронных сетей, цикличность процесса во времени раскладывают в многослойный перцептрон. В таком перцептроне каждый временной отрезок выступает в роли скрытого слоя. Только для всех слоев такого перцептрона используется одна матрица весовых коэффициентов. Поэтому для корректировки весов берем сумму градиентов по всем слоям и считаем дельту весовых коэффициентов один раз для суммарного по всем слоям градиента.



Алгоритм обучения рекуррентной нейронной сети

К сожалению, столь простое решение не лишено недостатков. Подобный подход позволяет сохранить «память» на коротком временном отрезке. Цикличность умножения сигнала на коэффициент меньше единицы и применение функции активации нейрона ведет к постепенному затуханию сигнала с ростом количества таких циклов. Для решения данной проблемы в 1997 году Зепп Хохрайтер и Юрген Шмидху́бер предложили использовать архитектуру «Долгая краткосрочная память» (*Long short-term memory* — *LSTM*). На сегодняшний день алгоритм *LTSM* считается одним из лучших для решения задач классификации и прогнозирования временных рядов, когда значимые события разделены во времени и растянуты по временным интервалам.

LSTM сложно назвать нейроном. Скорее он уже является нейронной сетью с тремя каналами входа данных и тремя каналами выхода данных. Из них только по двум каналам осуществляется обмен данными с окружающим миров (один для входа и один для выхода). Остальные четыре канала замкнуты попарно для циклического обмена информацией (*Memory* — память и *Hidden state* — скрытое состояние).

Внутри блока *LSTM* содержится два основных потока информации, которые связаны между собой четырьмя полносвязными нейронными слоями. Все нейронные слои содержат одинаковое количество нейронов, которое равно размеру выходного потока и потока памяти. Рассмотрим более детально алгоритм.

Поток данных *Memory* (память) служит для хранения и передачи во времени важной информации. На начальной стадии инициализируется нулевыми значениями и заполняется в процессе работы нейронной сети. Можно сравнить с живым человеком, который рождается без знаний и обучается на протяжении всей жизни.

Поток *Hidden state* (скрытое состояние) предназначен для передачи во времени выходного состояния системы. Размер канала данных равен каналу данных «памяти».

Каналы *Input data* (входные данные) и *Output state* (выходное состояние) предназначены для обмена информацией с окружающим миром.

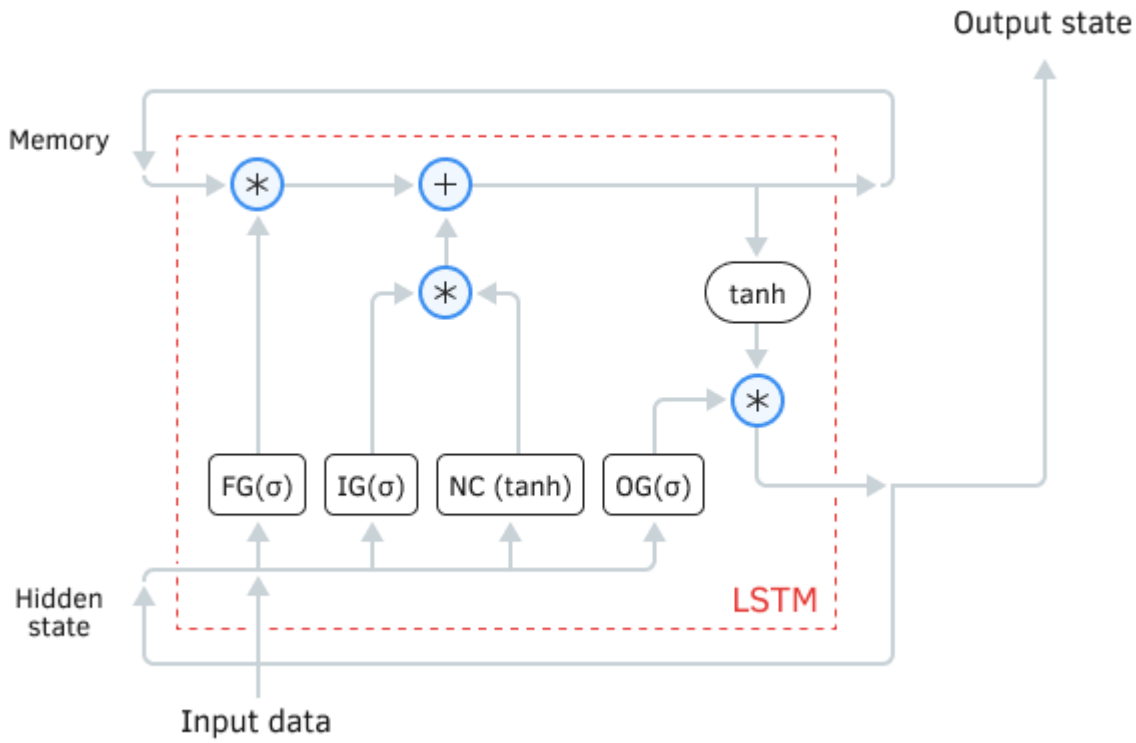


Схема LSTM модуля

На вход алгоритма поступаю три потока данных:

- *Input data* — описывает текущее состояние системы.
- *Memory* и *Hidden state* — получаем из предыдущего состояния.

В начале работы алгоритма информация из *Input data* и *Hidden state* объединяются в единый массив данных, который в последующем подается на все четыре скрытых нейронных слоя *LSTM*.

Первый нейронный слой Forget gate (врата забвения) определяет, какую из хранящейся в памяти информацию можно забыть, а какую нужно помнить. Организован в виде полносвязного нейронного слоя с сигмоидной функцией активации. Количество нейронов в слое соответствует количеству ячеек памяти в потоке Memory. Каждый нейрон слоя получает на входе суммарный массив данных потоков *Input data* и *Hidden state*, а на выходе выдает число в диапазоне от 0 (полностью забыть) до 1 (сохранить в памяти). Поэлементное произведение выходных данных нейронного слоя с потоком памяти возвращает скорректированную память.

$$FG_t = \sigma(W_{FG} INP_t + U_{FG} HS_{t-1})$$

где:

- σ — логистическая функция активации;
- W_{FG} — матрица весов для входного вектора;
- INP_t — вектор входного сигнала на текущей итерации;
- U_{FG} — матрица весов для скрытого состояния;
- HS_{t-1} — вектор скрытого состояния с предыдущей итерации.

На следующем шаге алгоритм определяет, какую из вновь полученной информации на данном шаге необходимо сохранить в памяти. Для этого используются два нейронных слоя:

- *New Content* (новый контент) — полносвязный нейронный слой с гиперболическим тангенсом в качестве функции активации нормализует полученную информации в диапазоне от −1 до 1.

$$NC_t = tanh(W_{NC}INP_t + U_{NC}HS_{t-1})$$

- *Input gate* (входные врата) — полносвязный нейронный слой с сигмоидой в качестве функции активации. Аналогичен Forget gate и определяет какую новую информацию нужно запомнить.

$$IG_t = \sigma(W_{IG}INP_t + U_{IG}HS_{t-1})$$

Использование гиперболического тангенса в качестве функции активации для нейронного слоя нового контента позволяет разделить полученную информацию на положительную и отрицательную. Поэлементное произведение *New Content* и *Input gate* определяет важность полученной информации и объем, в котором ее необходимо сохранить в памяти.

Полученные в результате операций вектор значений поэлементно складываем с вектором текущей памяти. В результате получаем обновленное состояние памяти, которое впоследствии передается на вход следующего цикла итераций.

$$M_t = FG_t \circ M_{t-1} + IG_t \circ NC_t$$

После обновления памяти сформируем значения выходного потока. Для этого нормализуем текущее значение памяти с помощью гиперболического тангенса. По аналогии с *Forget gate* и *Input gate* пересчитаем *Output gate* (ворота выходного сигнала), которые также активируются сигмоидной функцией.

$$OG_t = \sigma(W_{OG}INP_t + U_{OG}HS_{t-1})$$

Поэлементное произведение двух полученных векторов данных дает массив выходного сигнала, который выдается из LSTM во внешний мир. Этот же массив данных передадим на следующий цикл итераций в качестве потока скрытого состояния.

$$OUT_t = HS_t = OG_t \circ tanh(M_t)$$

С момента появления *LSTM*-блока свет увидел много его различных модификаций. Некоторые пытались сделать его «легче» для более быстрой обработки информации и обучения. Другие наоборот усложняли в попытке получить лучшие результаты. Наиболее удачной из них считается модель *GRU* (*Gated Recurrent Unit* — рекуррентный блок с вратами), которую представил Кенхен Чо (Kyunghyun Cho) и команда в сентябре 2014 года. Данное решение можно считать упрощенной версией стандартного блока *LSTM*. В нем ворота забвения и входные врата объединяются в одни врата обновления. При этом отказываются от использования отдельного потока памяти. Для передачи информации сквозь время используется только скрытое состояние.

В начале алгоритма *GRU*, как и в *LSTM*, определяются значения врат обновления и сброса. Математическая формула вычисления значений аналогична определению значений ворот в *LSTM*.

$$UG_t = \sigma(W_{UG}INP_t + U_{UG}HS_{t-1})$$
$$RG_t = \sigma(W_{RG}INP_t + U_{RG}HS_{t-1})$$

Затем обновляется текущее состояние памяти. При этом вначале скрытое состояние с предыдущей итерации умножается на соответствующую матрицу весов и затем поэлементно умножается на значение ворот сброса. Полученный вектор складывается с произведение исходных данных на свою матрицу весов. Суммарный вектор активируется гиперболическим тангенсом.

$$HS'_t = tanh(W_{HS}, INP_t + RG \circ U_{HS}, HS_{t-1})$$

В завершение алгоритма скрытое состояние с предыдущей итерации поэлементно умножается на значение врат обновления, а текущее состояние памяти умножается на разницу единицы и значение врат обновления. Сумма этих произведений подается на выход из блока и в качестве срытого состояния для следующей итерации.

$$HS_t = UG_t \circ HS_{t-1} + (1 - UG_t) \circ HS'_{t-1}$$

Таким образом, в модели *GRU* врата сброса регулируют скорость забывания данных. Врата обновления определяют, какую долю информации взять из предыдущего состояния, а какую из новых данных.

4.2.2 Построение LSTM-блока средствами MQL5

Для реализации в нашей библиотеке из всех вариантов архитектурных решений рекуррентных нейронов я выбрал классический *LSTM*-блок. На мой взгляд, наличие фильтров новой информации и содержимого памяти в виде ворот поможет минимизировать влияние шумовой составляющей сигнала. А отдельный канал памяти поможет сохранить информацию на более длительный срок.

Как и ранее, для создания нового типа нейронного слоя мы создадим новый класс *CNeuronLSTM*. Для сохранения наследственности новый класс создадим на основе нашего базового класса нейронного слоя *CNeuronBase*.

```
class CNeuronLSTM     :  public CNeuronBase
  {
public:
                      CNeuronLSTM(void);
                     ~CNeuronLSTM(void);
   //--- method of identifying the object
   virtual int        Type(void)               const { return(defNeuronLSTM); }
  };
```

Благодаря механизму наследования наш новый класс сразу после создания обладает базовым функционалом, который мы ранее заложили в родительский класс. Теперь нам предстоит доработать этот функционал для корректной работы нашего рекуррентного блока. Для начала перепишем виртуальный метод идентификации.

Как вы знаете из представленного в предыдущей главе описания архитектуры *LSTM*-блока, для его нормальной работы нам потребуется четыре полносвязных слоя. Их мы объявим в блоке *protected* нашего класса. А чтобы сохранить удобочитаемость кода, назовем их в соответствии с заложенным алгоритмом функционалом.

```
class CNeuronLSTM     :  public CNeuronBase
  {
protected:
   CNeuronBase*       m_cForgetGate;
   CNeuronBase*       m_cInputGate;
   CNeuronBase*       m_cNewContent;
   CNeuronBase*       m_cOutputGate;
```

Помимо созданных нейронных слоев алгоритмом блока предусматривается использование потоков памяти и скрытого состояния. Для их хранения нам потребуются отдельные буферы. Также нам потребуется в процессе обучения использовать хронологию состояния внутренних нейронов. Поэтому для хранения такой информации мы создадим динамические массивы, которые также объявим в блоке *protected*:

- *m_cMemorys* — состояние памяти;
- *m_cHiddenStates* — скрытое состояние;
- *m_cInputs* — конкатенированный массив исходных данных и скрытого состояния;
- *m_cForgetGateOuts* — состояние врат забвения;
- *m_cInputGateOuts* — состояние входных врат;
- *m_cNewContentOuts* — новый контент;
- *m_cOutputGateOuts* — состояние врат выходного сигнала.

```
class CNeuronLSTM     :  public CNeuronBase
  {
protected:
   ....
   CArrayObj*       m_cMemorys;
   CArrayObj*       m_cHiddenStates;
   CArrayObj*       m_cInputs;
   CArrayObj*       m_cForgetGateOuts;
   CArrayObj*       m_cInputGateOuts;
   CArrayObj*       m_cNewContentOuts;
   CArrayObj*       m_cOutputGateOuts;
```

Конечно, в процессе работы нейронной сети мы не можем бесконечно накапливать историю состояний, ведь ресурсы наши конечны. Поэтому нам понадобится некий ориентир для понимания заполнения буфера. При переполнении буфера выше этого предела мы будем удалять наиболее старые данные и замещать их новыми. Таким ориентиром нам послужит глубина истории для обучения рекуррентного блока. Этот параметр будет задаваться пользователем и храниться в переменной *m_iDepth*.

```
class CNeuronLSTM     :  public CNeuronBase
  {
protected:
   ....
   int              m_iDepth;
```

Продолжая разговор об объявлении вспомогательных переменных класса, следует обратить внимание еще на один момент. Все четыре внутренних нейронных слоя используют одни и те же исходные данные — конкатенированный тензор исходных данных и срытого состояния. Метод передачи градиента через скрытый слой *CalcHiddenGradient* нашего базового класса построен таким образом, что он заменяет значения градиента ошибки в буфере предыдущего слоя. А нам

нужно суммировать градиент ошибки со всех внутренних потоков. Поэтому для накопления суммы градиентов добавим еще один буфер — *m_cInputGradient*.

```
class CNeuronLSTM    :  public CNeuronBase
  {
protected:
   ....
   CBufferDouble*       m_cInputGradient;
```

Кажется, с переменными разобрались. Теперь приступаем к построению методов класса. Первое, с чего начинается работа класса, — это конструктор *CNeuronLSTM::CNeuronLSTM*. В данном методе мы создаем экземпляры используемых объектов и задаем начальные значения внутренним переменным.

```
CNeuronLSTM::CNeuronLSTM(void)    : m_iDepth(2)
  {
   m_cForgetGate = new CNeuronBase();
   m_cInputGate = new CNeuronBase();
   m_cNewContent = new CNeuronBase();
   m_cOutputGate = new CNeuronBase();
   m_cMemorys = new CArrayObj();
   m_cHiddenStates = new CArrayObj();
   m_cInputs = new CArrayObj();
   m_cForgetGateOuts = new CArrayObj();
   m_cInputGateOuts = new CArrayObj();
   m_cNewContentOuts = new CArrayObj();
   m_cOutputGateOuts = new CArrayObj();
   m_cInputGradient = new CBufferType();
  }
```

Сразу же создаем деструктор класса *CNeuronLSTM::~CNeuronLSTM*, в котором осуществляется обратная операция — очистка памяти после окончания работы класса. Тут важно проследить за полной очисткой памяти, чтобы ничего не упустить.

```
CNeuronLSTM::~CNeuronLSTM(void)
  {
   if(m_cForgetGate)
      delete m_cForgetGate;
   if(m_cInputGate)
      delete m_cInputGate;
   if(m_cNewContent)
      delete m_cNewContent;
   if(m_cOutputGate)
      delete m_cOutputGate;
   if(m_cMemorys)
      delete m_cMemorys;
   if(m_cHiddenStates)
      delete m_cHiddenStates;
   if(m_cInputs)
      delete m_cInputs;
   if(m_cForgetGateOuts)
      delete m_cForgetGateOuts;
   if(m_cInputGateOuts)
      delete m_cInputGateOuts;
   if(m_cNewContentOuts)
      delete m_cNewContentOuts;
   if(m_cOutputGateOuts)
      delete m_cOutputGateOuts;
   if(m_cInputGradient)
      delete m_cInputGradient;
  }
```

Инициализация объекта

Далее давайте посмотрим на метод инициализации экземпляра класса *CNeuronLSTM::Init*. Именно в этом методе создаются и инициализируются все внутренние объекты и переменные, а также подготавливается необходимая база для нормальной работы нейронного слоя в соответствии с заданными пользователем требованиями. Подобный виртуальный метод мы создали в нашем базовом классе нейронных слоев и постоянно переопределяем в каждом нашем новом классе.

```
class CNeuronLSTM    :  public CNeuronBase
  {
protected:
   ....
public:
                     CNeuronLSTM(void);
                    ~CNeuronLSTM(void);
   //---
   virtual bool      Init(const CLayerDescription *desc) override;
```

Как вы знаете, аналогичный метод базового класса в параметрах получает описание создаваемого нейронного слоя. Так и наш метод в параметрах получит указатель на экземпляр класса *CLayerDescription*. Следовательно, в начале метода мы осуществляем проверку корректности полученного указателя и заданных в нем параметров. Прежде всего, указанный в

нем тип нейронного слоя должен соответствовать нашему классу. Также наш блок LSTM не может использоваться в качестве слоя исходных данных и должен содержать хотя бы один нейрон на выходе.

```
bool CNeuronLSTM::Init(const CLayerDescription *desc)
  {
//--- блок контролей
   if(!desc || desc.type != Type() || desc.count <= 0 || desc.window == 0)
      return false;
```

Использование блока *LSTM* в качестве слоя исходных данных попросту пустая трата ресурсов. Мы создаем большое количество дополнительных объектов, которые никогда не будут использоваться, ведь в слой исходных данных мы записываем информацию напрямую в выходной буфер.

Далее нам предстоит инициализировать наши внутренние нейронные слои. Для этого мы будем вызывать аналогичный метод *Init* наших объектов. Следовательно, нам нужно им передать соответствующий экземпляр класса *CLayerDescription*. Просто передать полученный от пользователя объект описания рекуррентного блока мы не можем, так как нам нужно создать другие объекты. Поэтому сначала мы подготовим описание создаваемых объектов:

- Все внутренние нейронные слои являются полносвязными. Значит, мы создаем объекты базового класса. Поэтому в параметре *type* укажем тип *defNeuronBase*.
- Все они получают на вход один тензор, являющийся объединением вектором исходных данных и скрытого состояния. Размер вектора исходных данных мы получаем в параметрах метода (параметр *CLayerDescription.window*). Размер вектора скрытого состояния равен размеру буфера результатов текущего слоя. Это значение мы тоже получаем в параметрах метода (параметр *CLayerDescription.count*). Сумму двух указанных значений запишем в параметр *window*.
- Если внимательно посмотреть на схему *LSTM*-блока, приведенную в предыдущем разделе, то можно заметить: все внутренние потоки информации имеют одинаковый размер. Вектор результатов врат забвения поэлементно умножается на поток памяти. Значит, их размеры равны. Аналогично вектор результатов входных врат поэлементно умножается на результат слоя нового контента. Потом это произведение поэлементно суммируется с потоком памяти. В заключение все атомарно умножается на врата контроля выходных данных. Становится очевидно, что все потоки равны размеру выходного буфера текущего блока. Поэтому в параметр *count* перенесем значение аналогичного элемента из внешних параметров метода.
- Функция активации определена архитектурой LSTM блока. Все врата активируются сигмоидой, а слой нового контента — гиперболическим тангенсом. Вместе с функцией активации укажем соответствующие ей параметры.
- Метод оптимизации перенесем указанный пользователем.

```
//--- создаем описание для внутренних нейронных слоев
   CLayerDescription *temp = new CLayerDescription();
   if(!temp)
      return false;
   temp.type = defNeuronBase;
   temp.window = desc.window + desc.count;
   temp.count = desc.count;
   temp.activation = AF_SIGMOID;
   temp.activation_params[0] = 1;
   temp.activation_params[1] = 0;
   temp.optimization = desc.optimization;
```

После подготовки описания для внутренних нейронных слоев вернемся к нашему наследию от родительского класса. Все параметры блока скрыты во внутренних нейронных слоях, и нам нет необходимости держать в памяти дополнительную матрицу весов, как и сопутствующие ей буферы дельт и моментов. Кроме того, мы не планируем использовать объект класса активации *CActivation*. По существу, от базового класса нам достаточно функционала слоя исходных данных. Чтобы инициализировать нужные объекты и удалить излишние, обнулим размер исходных данных в описании рекуррентного блока и вызовем метод инициализации родительского класса.

```
//--- вызываем метод инициализации родительского класса
   CLayerDescription *temp2=new CLayerDescription();
   if(!temp2 || !temp2.Copy(desc))
     return false;
   temp2.window = 0;
   if(!CNeuronBase::Init(temp2))
      return false;
   delete temp2;
```

Для получения от пользователя информации о глубине истории для обучения рекуррентного блока будем использовать элемент *window_out*. Сохраним полученное значение в специально подготовленную переменную. Мы не проверяли это значение в начале метода, чтобы не блокировать работу нейронной сети. Вместо этого мы просто ограничили нижнюю границу сохраняемого значения. Поэтому, если пользователь забудет указать значение или укажет заведомо низкое значение, то нейронная сеть будет использовать указанное нами значение.

```
   if(!InsertBuffer(m_cHiddenStates, m_cOutputs, false))
      return false;
   m_iDepth = (int)fmax(desc.window_out, 2);
```

Далее переходим к инициализации наших ворот. Первыми будут инициализированы врата забвения. Прежде чем вызывать метод инициализации объекта врат, нам необходимо проверить действительность указателя на объект. При необходимости создадим новый экземпляр объекта. Если попытка создания нового экземпляра объекта будет неудачна, то мы выходим из метода с результатом *false*. При наличии актуального экземпляра объекта осуществляем инициализацию ворот.

```
//--- инициализируем ForgetGate
   if(!m_cForgetGate)
     {
      if(!(m_cForgetGate = new CNeuronBase()))
         return false;
     }
   if(!m_cForgetGate.Init(temp))
      return false;
   if(!InsertBuffer(m_cForgetGateOuts, m_cForgetGate.GetOutputs(), false))
      return false;
```

Аналогичные итерации осуществляем и для двух других врат.

```
//--- инициализируем InputGate
   if(!m_cInputGate)
     {
      if(!(m_cInputGate = new CNeuronBase()))
         return false;
     }
   if(!m_cInputGate.Init(temp))
      return false;
   if(!InsertBuffer(m_cInputGateOuts, m_cInputGate.GetOutputs(), false))
      return false;

//--- инициализируем OutputGate
   if(!m_cOutputGate)
     {
      if(!(m_cOutputGate = new CNeuronBase()))
         return false;
     }
   if(!m_cOutputGate.Init(temp))
      return false;
   if(!InsertBuffer(m_cOutputGateOuts, m_cOutputGate.GetOutputs(), false))
      return false;
```

Слой нового контента будем инициализировать таким же способом. Предварительно лишь изменим в описании слоя тип функции активации.

```
//--- инициализируем NewContent
   if(!m_cNewContent)
     {
      if(!(m_cNewContent = new CNeuronBase()))
         return false;
     }
   temp.activation = AF_TANH;
   if(!m_cNewContent.Init(temp))
      return false;
   if(!InsertBuffer(m_cNewContentOuts, m_cNewContent.GetOutputs(), false))
      return false;
```

После инициализации внутренних слоев перейдем к остальным объектам нашего рекуррентного блока LSTM. Проведем инициализацию буфера накопления градиента. Как и в случае нейронных слоев, сначала проверяем действительность указателя на объект. При необходимости создаем

новый экземпляр класса. Потом заполняем весь буфер нулевыми значениями. Размер буфера берем из подготовленного ранее описания внутренних нейронных слоев.

```
//--- инициализируем буфер InputGradient
   if(!m_cInputGradient)
     {
      if(!(m_cInputGradient = new CBufferType()))
         return false;
     }
   if(!m_cInputGradient.BufferInit(1, temp.window, 0))
      return false;
   delete temp;
```

Надо сказать, что после инициализации буфера накопления значений градиента мы больше не будем использовать объект описания внутренних нейронных слоев. Поэтому с чистой совестью удаляем ненужный объект.

В заключение нам осталось создать и заполнить нулевыми значениями буферы потока памяти и скрытого состояния. Обратите внимание, что оба буфера будут использоваться при первом же прямом проходе, а их отсутствие парализует работу всей нейронной сети. Для создания указанных буферов был создан отдельный метод *CreateBuffer*, с алгоритмом которого мы познакомимся немного позже.

Итак, сначала создаем буфер памяти. Объявляем временную переменную и вызываем метод создания буфера *CreateBuffer*. В результате работы метода мы ожидаем получить указатель на объект буфера. Разумеется, после получения указателя мы проверяем его действительность. В случае ошибки выходим из метода с результатом *false*.

Далее мы проверяем наличие в стеке памяти существующих объектов. Напомню, мы рассматриваем метод инициализации экземпляра класса, поэтому ожидаем наличие пустого стека. Если же стек содержит какую-либо информацию, мы очищаем стек и заполняем нулевыми значениями созданный буфер. После этого помещаем наш буфер в стек памяти.

```
//--- инициализируем Memory
   CBufferType *buffer =  CreateBuffer(m_cMemorys);
   if(!buffer)
      return false;
   if(!InsertBuffer(m_cMemorys, buffer, false))
     {
      delete buffer;
      return false;
     }
```

В результате отработки данного блока кода метода мы ожидаем получить стек памяти, содержащий один нулевой буфер памяти. Обратите внимание, что в конце работы блока мы не удаляем объект буфера, хотя жизнь переменной не выходит за границы данного метода. Дело в том, что здесь мы оперируем указателями на объекты. Поместив указатель в стек, мы всегда можем его оттуда получить. И наоборот, если удалить объект по указателю переменной в стеке, у нас тоже окажется указатель на удаленный объект со всеми вытекающими последствиями. А непосредственно удаление объекта будет или при переполнении стека, или при попытке закрытия всего экземпляра класса.

Повторим все итерации для буфера скрытого состояния.

```
//--- инициализируем HiddenStates
   if(!(buffer =  CreateBuffer(m_cHiddenStates)))
      return false;
   if(!InsertBuffer(m_cHiddenStates, buffer, false))
     {
      delete buffer;
      return false;
     }
```

Напоследок передадим всем внутренним объектам текущий указатель на объект *OpenCL* и выйдем из метода.

```
//---
   SetOpenCL(m_cOpenCL);
//---
   return true;
  }
```

Выше мы рассмотрели алгоритм метода инициализации класса. Но, как вы могли заметить, по ходу выполнения алгоритма мы использовали два метода класса: *SetOpenCL* и *CreateBuffer*. Первый метод существует в родительском классе, но для корректной работы нам нужно будет его переопределить. Второй же метод новый.

Метод *CreateBuffer* в методе инициализации был использован для создания нового буфера. Но, забегая немного вперед, мы будем его использовать немного шире. Как вам известно из архитектуры выстраиваемого нами рекуррентного блока *LSTM*, на каждом прямом проходе нам потребуется извлечение из стека последних векторов скрытого состояния и памяти. Этот функционал мы также перенесем в метод *CreateBuffer*.

Так как мы предполагаем работу метода с несколькими стеками, то в параметрах метода будем указывать указатель на конкретный стек. А результатом работы метода будет указатель на нужный нам буфер. Объявим метод в блоке *protected* нашего класса.

```
class CNeuronLSTM    :  public CNeuronBase
  {
protected:
   ....
   CBufferDouble*     CreateBuffer(CArrayObj *&array);
```

В начале тела метода мы как обычно проверяем полученный указатель на стек. Но в случае получения недействительного указателя мы не спешим выйти из метода с сообщением об ошибке. Вместо этого мы попытаемся создать новый стек. И только в случае не возможности создания нового стека выйдем из метода.

Помните, код, вызвавший метод, ожидает получить назад не логическое состояние выполнения метода, а указатель на буфер. Поэтому в случае ошибки мы вернем *NULL* вместо ожидаемого указателя.

```
CBufferType *CNeuronLSTM::CreateBuffer(CArrayObj *&array)
  {
   if(!array)
     {
      array = new CArrayObj();
      if(!array)
         return NULL;
     }
```

Далее мы создаем новый буфер и сразу проверяем результат.

```
   CBufferType *buffer = new CBufferType();
   if(!buffer)
      return NULL;
```

После успешного создания буфера мы разделяем алгоритм на два потока. В одном случае, когда в стеке нет ни одного буфера, мы заполняем созданный нами буфер нулевыми значениями. Если же в стеке уже есть информация, то мы копируем в буфер последние состояние. После этого возвращаем указатель на буфер вызвавшей программе.

```
   if(array.Total() <= 0)
     {
      if(!buffer.BufferInit(m_cOutputs.Rows(), m_cOutputs.Cols(), 0))
        {
         delete buffer;
         return NULL;
        }
     }

   else
     {
      CBufferType *temp = array.At(0);
      if(!temp)
        {
         delete buffer;
         return NULL;
        }
      buffer.m_mMatrix = temp.m_mMatrix;
     }
//---
   if(m_cOpenCL)
     {
      if(!buffer.BufferCreate(m_cOpenCL))
         delete buffer;
     }
//---
   return buffer;
  }
```

Обратите внимание, я говорю о последних данных и при этом копирую буфер с индексом 0. В данном классе реализована логика обратного стека. Каждый новый буфер мы будем вставлять в начала стека, смещая более старые, а при заполнении стека будем удалять последние.

И второй момент: мы не берем указатель на существующий буфер, а создаем новый. Это связано с тем, что в процессе прямого прохода мы будем изменять содержимое буфера. При этом нам важно сохранить предыдущее состояние. В случае же использования указателя на старый буфер мы просто перепишем его значения, удалив нужные нам прошлые состояния.

Второй метод *SetOpenCL* является переопределением метода родительского класса и имеет тот же функционал — передача указателя на контекст *OpenCL* до всех внутренних объектов, участвующих в процессе вычислений. Как и метод родительского класса наш метод в параметрах получит указатель на контекст *OpenCL* и вернет логический результат готовности класса к работе в указанном контексте.

```
class CNeuronLSTM     :  public CNeuronBase
  {
protected:
   ....
public:
   ....
   virtual bool      SetOpenCL(CMyOpenCL *opencl) override;
```

Алгоритм метода довольно прост. Мы сначала вызываем метод родительского класса и передаем ему полученный указатель. Проверка корректности полученного указателя уже реализована в методе родительского класса. Поэтому нам нет необходимости повторять ее здесь.

Затем мы передаем во все внутренние объекты указатель на контекст *OpenCL*, сохраненный в переменной нашего класса. Фокус в том, что метод родительского класса проверил полученный указатель и сохранил в переменную соответствующий указатель. И чтобы все объекты работали в одном контексте, мы распространяем уже обработанный указатель.

```
bool CNeuronLSTM::SetOpenCL(CMyOpenCL *opencl)
  {
//--- вызов метода родительского класса
   CNeuronBase::SetOpenCL(opencl);
//--- вызываем аналогичный метод для всех внутренних слоев
   m_cForgetGate.SetOpenCL(m_cOpenCL);
   m_cInputGate.SetOpenCL(m_cOpenCL);
   m_cOutputGate.SetOpenCL(m_cOpenCL);
   m_cNewContent.SetOpenCL(m_cOpenCL);
   m_cInputGradient.BufferCreate(m_cOpenCL);
   for(int i = 0; i < m_cMemorys.Total(); i++)
     {
      CBufferType *temp = m_cMemorys.At(i);
      temp.BufferCreate(m_cOpenCL);
     }
   for(int i = 0; i < m_cHiddenStates.Total(); i++)
     {
      CBufferType *temp = m_cHiddenStates.At(i);
      temp.BufferCreate(m_cOpenCL);
     }
//---
   return(!!m_cOpenCL);
  }
```

На этом можно сказать, что мы завершили работу по созданию алгоритма инициализации класса. Теперь мы можем перейти к следующему этапу нашей работы — созданию алгоритма прямого прохода.

4.2.2.1 Метод прямого прохода LSTM-блока

Как всегда, алгоритм прямого прохода мы будем создавать в методе *FeedForward*. Метод прямого прохода является одним из основных методов, который определен виртуальным методом в базовом классе и переопределяется во всех наследниках.

```
class CNeuronLSTM    :  public CNeuronBase
  {
protected:
   ....
public:
   ....
   virtual bool      FeedForward(CNeuronBase *prevLayer) override;
```

Метод *FeedForward* получает в параметрах указатель на предыдущий нейронный слой, который содержит исходные данные для работы метода, и возвращает логическое значение состояния выполнения операций метода.

В начале метода мы проверяем действительность указателей на все объекты, критичные для выполнения операций метода. При наличии хотя бы одного недействительного указателя выходим из метода с результатом *false*.

```
bool CNeuronLSTM::FeedForward(CNeuronBase *prevLayer)
  {
//--- проверяем актуальность всех объектов
   if(!prevLayer || !prevLayer.GetOutputs() || !m_cOutputs ||
      !m_cForgetGate || !m_cInputGate || !m_cOutputGate ||
      !m_cNewContent)
      return false;
```

После успешного прохождения блока контролей создадим заготовки под новые буфера памяти и скрытого состояния. Для этого воспользуемся рассмотренным выше методом создания буфера *CreateBuffer*. И, конечно, не забываем проконтролировать результат выполнения операций.

```
//--- подготавливаем заготовки для новых буферов
   if(!m_cForgetGate.SetOutputs(CreateBuffer(m_cForgetGateOuts), false))
      return false;
   if(!m_cInputGate.SetOutputs(CreateBuffer(m_cInputGateOuts), false))
      return false;
   if(!m_cOutputGate.SetOutputs(CreateBuffer(m_cOutputGateOuts), false))
      return false;
   if(!m_cNewContent.SetOutputs(CreateBuffer(m_cNewContentOuts), false))
      return false;
   CBufferType *memory = CreateBuffer(m_cMemorys);
   if(!memory)
      return false;
   CBufferType *hidden = CreateBuffer(m_cHiddenStates);
   if(!hidden)
     {
      delete memory;
      return false;
     }
```

Далее нам предстоит подготовить исходные данные для корректной работы внутренних слоев. Эта процедура не так проста, как может показаться на первый взгляд. Дело в том, что для вызова методов прямого прохода наших ворот нам необходим не просто буфер, а нейронный слой. Поставить указатель на полученный в параметрах предыдущий слой мы не можем, так как он не содержит всей необходимой информации. В нем отсутствуют данные скрытого состояния, необходимые для корректной работы алгоритма. Поэтому нам потребуется создать пустой нейронный слой и заполнить его выходной буфер необходимыми данными.

Но прежде чем создать новый нейронный слой, проверим действительность указателя на стек хранения нейронных слоев исходных данных. При необходимости создадим новый, ведь после осуществления прямого прохода нам потребуется сохранить созданный нейронный слой для последующего обучения нейронной сети. Контроль наличия стека осуществляется до проведения полного цикла операций прямого прохода, чтобы сэкономить ресурсы на проведения излишних операций.

```
//--- создаем буфер исходных данных
   if(!m_cInputs)
     {
      m_cInputs = new CArrayObj();
      if(!m_cInputs)
        {
         delete memory;
         delete hidden;
         return false;
        }
     }
```

Следует обратить внимание, что перед выходом из метода после неудачной попытки создания нового стека нам необходимо будет удалить созданные в методе объекты, указатели на которые не переданы глобальным переменным класса.

Далее мы создаем новый экземпляр объекта базового нейронного слоя. И, как всегда, проверяем результат проведения операции.

```
CNeuronBase *inputs = new CNeuronBase();
if(!inputs)
  {
   delete memory;
   delete hidden;
   return false;
  }
```

После успешного создания экземпляра объекта базового нейронного слоя для его инициализации нам потребуется создать объект описания структуры нейронного слоя. Этим мы и займемся. Создадим экземпляр объекта *CLayerDescription* и наполним его необходимыми данными. Тип нейронного слоя укажем *defNeuronBase*. Количество элементов в нейронном слое будет равно сумме элементов в буферах результатов предыдущего и текущего слоев. Так как буфер результатов создаваемого слоя будем заполнять напрямую из других источников, то размер окна исходных данных задаем равным 0.

```
CLayerDescription *desc = new CLayerDescription();
if(!desc)
  {
   delete inputs;
   delete memory;
   delete hidden;
   return false;
  }
desc.type = defNeuronBase;
desc.count = (int)(prevLayer.GetOutputs().Total() + m_cOutputs.Total());
desc.window = 0;
```

После создания описания нейронного слоя производим его инициализацию и после успешной операции удаляем уже не нужный объект описания слоя.

```
if(!inputs.Init(desc))
  {
   delete inputs;
   delete memory;
   delete hidden;
   delete desc;
   return false;
  }
delete desc;
inputs.SetOpenCL(m_cOpenCL);
```

После этого нам остается лишь заполнить буфер результатов нового слоя необходимыми исходными данными. Для начала получим указатель на нужный буфер и проверим его актуальность.

```
CBufferType *inputs_buffer = inputs.GetOutputs();
if(!inputs_buffer)
  {
   delete inputs;
   delete memory;
   delete hidden;
   return false;
```

```
   }
```

После этого заполним буфер содержимым буферов результатов предыдущего слоя и скрытого состояния. Функционал переноса данных мы вынесли в отдельный метод *Concatenate*, с кодом которого познакомимся позже.

```
   if(!inputs_buffer.Concatenate(prevLayer.GetOutputs(), hidden,
                                 prevLayer.Total(), hidden.Total()))
     {
      delete inputs;
      delete memory;
      delete hidden;
      return false;
     }
```

Теперь, когда мы закончили подготовительную работу, можно приступить непосредственно к операциям прямого прохода. И начнем мы эту работу с вызова методов прямого прохода внутренних нейронных слоев. Первым выполним прямой проход для ворот забвения. Мы лишь вызываем одноименный метод для соответствующего объекта. В параметрах методу передадим указатель на только что созданный экземпляр нейронного слоя исходных данных и проверяем результат выполнения операций.

```
//--- делаем прямой проход внутренних нейронных слоев
   if(!m_cForgetGate.FeedForward(inputs))
     {
      delete inputs;
      delete memory;
      delete hidden;
      return false;
     }
```

Повторим операцию для всех внутренних слоев.

```
   if(!m_cInputGate.FeedForward(inputs))
     {
      delete inputs;
      delete memory;
      delete hidden;
      return false;
     }

   if(!m_cOutputGate.FeedForward(inputs))
     {
      delete inputs;
      delete memory;
      delete hidden;
      return false;
     }

   if(!m_cNewContent.FeedForward(inputs))
     {
      delete inputs;
      delete memory;
      delete hidden;
      return false;
     }
```

После успешного прохождения прямого прохода всех внутренних нейронных слоев, в буфере результатов каждого объекта будет храниться готовая информация о состоянии всех врат и нормализованные данные нового контента. И теперь нам лишь остается объединить все информационные потоки в соответствии с алгоритмом работы *LSTM*-блока. Прежде чем построить этот процесс, нам предстоит организовать разветвление алгоритма в зависимости от используемого устройства вычислительных операций: *CPU* стандартными средствами MQL5 или контекст *OpenCL*.

Может возникнуть резонный вопрос: почему мы разделяем потоки операций только сейчас? Почему мы не использовали мощь многопоточных операций при расчете состояния врат и нового контекста? Но поверьте, в этих операциях тоже использовалась технология многопоточных вычислений, предлагаемая OpenCL, хотя и не так явно. К примеру, в методе *CNeuronLSTM::SetOpenCL* мы передали указатель на контекст *OpenCL* всем внутренним нейронным слоям, а буквально несколькими строками выше вызывали методы прямого прохода для каждого внутреннего слоя. А теперь посмотрите в метод прямого прохода родительского класса *CNeuronBase::FeedForward*, там тоже есть разделение потоков.

```
bool CNeuronBase::FeedForward(CNeuronBase *prevLayer)
  {
   ....
//--- Разделение алгоритма по вычислительному устройству
   if(!m_cOpenCL)
     {
   ....
     }
   else
     {
   ....
     }
//---
   return false;
  }
```

Иными словами, ранее мы использовали уже готовые методы базового класса нейронных слоев с уже готовым функционалом в обоих направлениях. Сейчас мы вводим дополнительные операции, присущие только *LSTM*-блоку. Поэтому у нас появилась необходимость разделение потока операций и организация процесса для обоих технологий. Как и при построении предыдущих классов, сейчас мы разберем построение алгоритма средствами *MQL5*. На саму организацию процесса в контексте *OpenCL* посмотрим в следующей главе.

При выполнении операций средствами MQL5 вначале для удобства доступа к буферам результатов внутренних нейронных слоев получим указатели на данные буферы в локальные переменные. Затем воспользуемся матричными операциями *MQL5*.

Вначале мы поэлементно умножим состояние памяти (*Memory*) на значения врат забвения (*Forget Gate*). Затем мы поэлементно умножим нормализованную матрицу нового состояния (*New Content*) на входные врата (*Input Gate*). Результат прибавим к обновленному состоянию памяти (*Memory*). В завершение нормализуем результат выполненных выше операций с помощью функции гиперболического тангенса и поэлементно умножим на матрицу врат результатов (*Output Gate*). Результат запишем в буфер скрытого состояния (*Hidden*).

```
//--- разветвление алгоритма по вычислительному устройству
   CBufferType *fg = m_cForgetGate.GetOutputs();
   CBufferType *ig = m_cInputGate.GetOutputs();
   CBufferType *og = m_cOutputGate.GetOutputs();
   CBufferType *nc = m_cNewContent.GetOutputs();
   if(!m_cOpenCL)
     {
      memory.m_mMatrix *= fg.m_mMatrix;
      memory.m_mMatrix += ig.m_mMatrix * nc.m_mMatrix;
      hidden.m_mMatrix = MathTanh(memory.m_mMatrix) * og.m_mMatrix;
     }
```

Для направления алгоритма контекста OpenCL мы пока сделаем выход с отрицательным результатом, который будет заменен позже корректным кодом. Такой вариант позволит нам протестировать готовый код и предупредит о выборе неправильного параметра.

```
   else
     {
      delete inputs;
      delete memory;
      delete hidden;
      return false;
     }
```

После завершения полного цикла обновления памяти и скрытого состояния нашего блока *LSTM* перенесем значения скрытого состояния в буфер результатов.

```
//--- копируем скрытое состояние в буфер результатов нейронного слоя
   m_cOutputs = hidden;
```

На этом можно было бы завершить прямой проход. Но нам еще нужно сохранить текущее состояние для последующего обучения нашего рекуррентного блока. Вначале сохраним в стек исходные данные. Как уже говорилось выше, вставлять новые объекты в стек мы будем с индексом 0.

```
//--- сохраним текущее состояние
   if(!m_cInputs.Insert(inputs, 0))
     {
      delete inputs;
      delete memory;
      delete hidden;
      return false;
     }
```

После добавления нового элемента проверим стек на переполнение и удалим излишние исторические данные. Для выполнения этого функционала создадим метод *ClearBuffer*. На алгоритм работы данного метода посмотрим немного позже.

```
   ClearBuffer(m_cInputs);
```

Здесь надо сказать, что исходные данные мы сохраняем в виде нейронного слоя. Это нам позволяет решить сразу две задачи:

1. Методы прямого и обратного прохода базового нейронного слоя требуют на вход указатель предыдущего нейронного слоя. Следовательно один объект мы можем использовать и для прямого и для обратного прохода без каких-либо доработок базового нейронного слоя.
2. В одном объекте мы сохраняем и буфер исходных данных и буфер градиентов. У нас нет необходимости настраивать синхронизацию использования буферов.

В остальных стеках мы будем сохранять буферы. Поэтому для повторяющейся работы по сохранению данных в стеки мы создадим дополнительный метод *InsertBuffer*. На алгоритм метода мы посмотрим чуть позже, а сейчас мы будем использовать его для копирования информации в стеки. Повторим вызов указанного метода для каждого стека и соответствующего буфера.

```
   if(!InsertBuffer(m_cForgetGateOuts, m_cForgetGate.GetOutputs(), false))
     {
      delete memory;
      delete hidden;
      return false;
     }

   if(!InsertBuffer(m_cInputGateOuts, m_cInputGate.GetOutputs(), false))
     {
      delete memory;
      delete hidden;
      return false;
     }

   if(!InsertBuffer(m_cOutputGateOuts, m_cOutputGate.GetOutputs(), false))
     {
      delete memory;
      delete hidden;
      return false;
     }

   if(!InsertBuffer(m_cNewContentOuts, m_cNewContent.GetOutputs(), false))
     {
      delete memory;
      delete hidden;
      return false;
     }
```

Обратите внимание, что выше мы сохраняли буфера результатов внутренних слоев. Данные объекты принадлежат к структуре нейронного слоя и будут удалены из памяти вместе при удалении соответствующего нейронного слоя. Поэтому в методе *InsertBuffer* мы не будем создавать новый экземпляр объекта буфера и копировать данные.

Здесь надо четко понимать различия между указателем на объект и самим объектом. Каждый раз, когда мы создаем некий объект, под него выделяется некий объем памяти. Туда записывается необходимая информация. Это наш объект. Для обращения к нему сохраняется указатель на объект. Он содержит ссылку на участок памяти, в которую записан объект. Следовательно, при обращении к объекту мы берем указатель, переходим к нужному нам участку памяти и считываем необходимую информацию.

Когда мы копируем указатель на объект, мы не создаем новый объект, а лишь делаем копию ссылки. Поэтому когда кто-то внесет изменения в содержимое объекта, мы также увидим эти изменения, обратившись к объекту по нашему указателю. Добро это или зло, зависит от метода использования объекта. Когда нам нужна синхронизация работы с объектом из разных источников — это добро. Все будут обращаться к одному объекту. А значит нет необходимости в синхронизации данных в разных хранилищах. К тому же указатель требует меньше ресурсов, чем создание нового объекта. Но когда нам нужно сохранить некие данные от изменения, тут лучше создать новый объект и скопировать необходимую информацию.

```
      if(!InsertBuffer(m_cMemorys, memory, false))
        {
         delete hidden;
         return false;
        }

      if(!InsertBuffer(m_cHiddenStates, hidden, false))
         return false;
  //---
      return true;
     }
```

После успешного сохранения всей необходимой информации в стеке выходим из метода с положительным результатом.

Поздравляю! Мы добрались до конца метода прямого прохода. Его не назовешь самым простым. Но, надеюсь, мои комментарии позволили вам понять его алгоритм и идею построения процесса. Правда, у нас еще остались открытые вопросы в виде вспомогательных методов.

По ходу алгоритма метода прямого прохода первым мы упоминали метод *ClearBuffer*. Здесь все довольно просто и прозаично. В параметрах метод получает указатель на стек. Как всегда, в начале метода мы проверяем действительность полученного указателя. После успешного прохождения проверки указателя мы проверяем размер буфера. И если размер буфера превышает указанный пользователем размер, удаляем последние элементы. Тем самым мы приводим размер буфера в рамки заданного размера. Как видите, весь код метода уместился буквально в пять строчек.

```
void CNeuronLSTM::ClearBuffer(CArrayObj *buffer)
  {
   if(!buffer)
      return;
   int total = buffer.Total();
   if(total > m_iDepth + 1)
      buffer.DeleteRange(m_iDepth + 1, total);
  }
```

Потом мы говорили о методе добавления буфера в стек *InsertBuffer*. Этот метод имеет три параметра, последний из них имеет значение по умолчанию и не обязателен для указания при вызове метода:

- *CArrayObj *array* — указатель на стек для добавления буфера;
- *CBufferType *element* — указатель на добавляемый буфер;
- *bool create_new* — логическая переменная, указывающая на необходимость создания дубликата буфера. По умолчанию создается дубликат буфера.

В результате операций метод возвращает логическое значение о состоянии выполнения операций.

Как всегда, в начале метода проверим действительность полученных указателей. И здесь есть один нюанс. Первым мы проверяем указатель на буфер для добавления в стек. При наличии недействительного указателя нам нечего добавлять в стек. Естественно, в такой ситуации мы выходим из метода с отрицательным результатом.

А вот если указатель на стек окажется недействительным, мы сначала попытаемся создать новый стек. И только после неудачной попытки мы выйдем из метода с отрицательным результатом. Но если нам удастся создать новый стек, мы продолжим работу в стандартном режиме.

```
bool CNeuronLSTM::InsertBuffer(CArrayObj *&array,
                               CBufferType *element,
                               bool create_new = true)
  {
//--- блок контролей
   if(!element)
      return false;
   if(!array)
     {
      array = new CArrayObj();
      if(!array)
         return false;
     }
```

Далее мы разделяем алгоритм на две отдельные ветки в зависимости от необходимости создания дубликата буфера. Если нам нужен дубликат буфера, то мы сначала создаем новый экземпляр объекта буфера и сразу проверяем результат проведенной операции по полученному указателю на объект.

```
   if(create_new)
     {
      CBufferType *buffer = new CBufferType();
      if(!buffer)
         return false;
```

Затем мы перенесем содержимое буфера-источника в новый буфер. Только после этого добавим указатель на новый буфер в стек. Напомню, что новые элементы мы добавляем в стек с нулевым индексом.

```
      buffer.m_mMatrix = element.m_mMatrix;
      if(!array.Insert(buffer, 0))
        {
         delete buffer;
         return false;
        }
     }
```

Если же нам нет необходимости создавать новый экземпляр буфера, то тут все намного проще. Мы лишь добавляем полученный в параметр указатель на буфер в стек.

```
      else
        {
         if(!array.Insert(element, 0))
           {
            delete element;
            return false;
           }
        }
```

После добавления нового элемента в стек проверим его размер и удалим излишнюю историю. Для этого мы воспользуемся уже известным методом *ClearBuffer*.

```
  //--- удалим из буфера излишнюю историю
     ClearBuffer(array);
  //---
     return true;
    }
```

После завершения операций выходим из метода с положительным результатом.

Мы полностью рассмотрели алгоритм прямого прохода и участвовавшие в нем методы. Дальше рассмотрим обратный проход.

4.2.2.2 Методы обратного прохода LSTM-блока

Прямой проход представляет собой стандартный режим работы нейронной сети. Но прежде чем ее использовать в промышленной эксплуатации, нам нужно обучить свою модель. Обучение рекуррентных нейронных сетей осуществляется уже знакомым нам методом обратного распространения ошибки с небольшим дополнением. Дело в том, что в отличие от рассмотренных нами ранее типов нейронных слоев, только рекуррентные слои используют результаты своей работы в качестве своих исходных данных на будущих итерациях. И все они имеют свои весовые коэффициенты, которые тоже надо обучать. В процессе обучения нам предстоит развернуть рекуррентные слои по хронологии как многослойный перцептрон. С той лишь разницей, что все слои будут использовать одну матрицу весов. Именно с этой целью при прямом проходе мы сохраняли хронологию состояния всех объектов. Теперь настало время воспользоваться ими по назначению.

За процесс обратного прохода в базовом классе нейронного слоя у нас отвечают три метода:

- *CalcHiddenGradient* — распределение градиента через скрытый слой;
- *CalcDeltaWeights* — распределение градиента до матрицы весовых коэффициентов;
- *UpdateWeights* — метод обновления весовых коэффициентов.

```
class CNeuronLSTM    :  public CNeuronBase
  {
protected:
   ....
public:
   ....
   virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer)  override;
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer)    override
                                                          { return true; }
   virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                   VECTOR &Beta, VECTOR &Lambda) override;
```

Их нам и предстоит переопределить.

Первым мы переопределим метод распределения градиента через скрытый слой *CalcHiddenGradient*. Здесь нам потребуется развернуть всю историческую цепочку и провести градиент ошибки через все состояния. Кроме того, не будем забывать, что помимо распределения градиента внутри LSTM-блока мы должны выполнить и вторую функцию данного метода — передать градиент ошибки на предыдущий слой.

В параметрах рассматриваемый метод получает указатель на объект предыдущего слоя, а возвращает логический результат выполнения операций.

В начале метода мы осуществляем проверку всех используемых объектов. Проверяем как поступившие в параметрах указатели на объекты предыдущего слоя, так и внутренние объекты.

```
bool CNeuronLSTM::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
//--- проверяем актуальность всех объектов
   if(!prevLayer || !prevLayer.GetGradients() ||
      !m_cGradients || !m_cForgetGate || !m_cForgetGateOuts ||
      !m_cInputGate || !m_cInputGateOuts || !m_cOutputGate ||
      !m_cOutputGateOuts || !m_cNewContent || !m_cNewContentOuts)
      return false;
```

Не будем забывать, что обратный проход возможен только после прямого прохода. Ведь именно при проведении прямого прохода формируется база исходных данных для обратного прохода. Поэтому следующим шагом мы проверяем наличие информации в стеках памяти и скрытого состояния. К тому же наполняемость стека подскажет нам глубину распространения градиента в историю.

```
//--- проверяем наличие данных прямого прохода
   int total = (int)fmin(m_cMemorys.Total(), m_cHiddenStates.Total()) - 1;
   if(total <= 0)
      return false;
```

Продолжая подготовительную работу, создадим указатели на буферы результатов и градиентов внутренних слоев. Думаю, необходимость в указателях на буферы градиентов вполне понятна. Нам потребуется записывать в них градиенты ошибки, распространяя их по *LSTM*-блоку. А вот необходимость в буферах результатов не столь очевидна. Как вы знаете, каждый нейрон имеет функцию активации. Наши внутренние слои активируются логистической функцией и гиперболическим тангенсом. Полученный на входе нейронного слоя градиент ошибки необходимо скорректировать на производную функции активации. А производная указанных функций активации легко пересчитывается исходя из результата самой функции. Таким образом, для осуществления корректного обратного прохода нам необходимы соответствующие исходные данные. Для ранее рассмотренных нейронных слоев такой вопрос не поднимался, потому что корректные данные в буфер результатов записывались при прямом проходе. В случае же рекуррентного блока в буфере результатов будет сохранен только результат последнего прямого прохода. Для проработки глубины истории нам придется переписывать значения буфера результата значениями соответствующего временного шага.

```
//--- делаем указатели на буферы градиентов и результатов внутренних слоев
   CBufferType *fg_grad = m_cForgetGate.GetGradients();
   if(!fg_grad)
      return false;
   CBufferType *fg_out = m_cForgetGate.GetOutputs();
   if(!fg_out)
      return false;

   CBufferType *ig_grad = m_cInputGate.GetGradients();
   if(!ig_grad)
      return false;
   CBufferType *ig_out = m_cInputGate.GetOutputs();
   if(!ig_out)
      return false;

   CBufferType *og_grad = m_cOutputGate.GetGradients();
   if(!og_grad)
      return false;
   CBufferType *og_out = m_cOutputGate.GetOutputs();
   if(!og_out)
      return false;

   CBufferType *nc_grad = m_cNewContent.GetGradients();
   if(!nc_grad)
      return false;
   CBufferType *nc_out = m_cNewContent.GetOutputs();
   if(!nc_out)
      return false;
```

В завершение подготовительного процесса мы сохраним в локальную переменную размер буферов внутреннего потока.

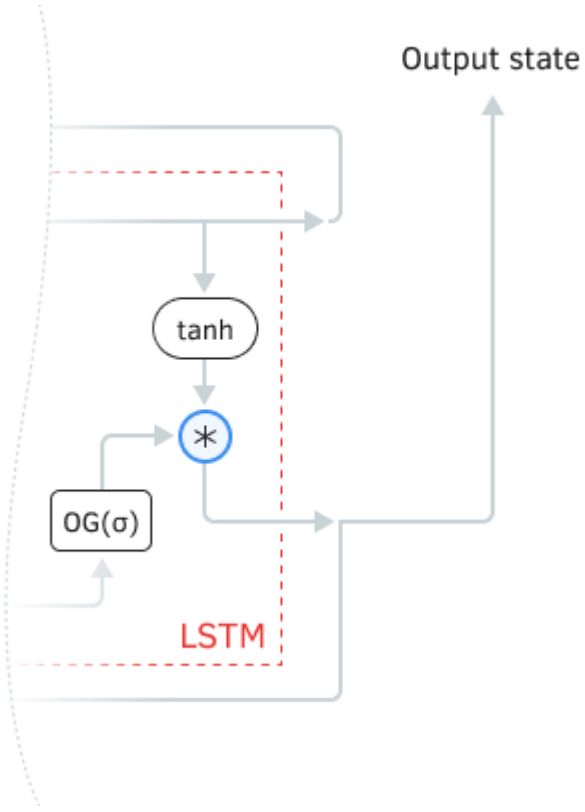
```
   uint out_total = m_cOutputs.Total();
```

Далее мы организуем цикл перебора исторических данных. Именно в теле данного цикла и будут осуществляться основные операции нашего метода. В начале цикла мы загрузим информацию соответствующего исторического шага из наших стеков. Обратите внимание, что все буферы загружаются для анализируемого хронологического шага, а буфер памяти берется с предшествующего анализируемому. Причины этого я поясню чуть ниже.

```
//--- цикл перебора накопленной истории
   for(int i = 0; i < total; i++)
     {
      //--- получаем указатели на буферы из стека
      CBufferType *fg = m_cForgetGateOuts.At(i);
      if(!fg)
         return false;
      CBufferType *ig = m_cInputGateOuts.At(i);
      if(!ig)
         return false;
      CBufferType *og = m_cOutputGateOuts.At(i);
      if(!og)
         return false;
      CBufferType *nc = m_cNewContentOuts.At(i);
      if(!nc)
         return false;
      CBufferType *memory = m_cMemorys.At(i + 1);
      if(!memory)
         return false;
      CBufferType *hidden = m_cHiddenStates.At(i);
      if(!hidden)
         return false;
      CNeuronBase *inputs = m_cInputs.At(i);
      if(!inputs)
         return false;
```

Далее нам предстоит распределить градиент ошибки, поступивший на вход *LSTM*-блока между внутренними нейронными слоями. В этом месте мы строим новый процесс. Следуя нашей концепции построения класса, создадим разделение алгоритма в зависимости от устройства выполнения математических операций.

Распределение градиента ошибки осуществляется в обратном порядке прямого потока информации. Следовательно, алгоритм его распространения мы будем строить от выхода ко входу. Посмотрим на узел результатов нашего *LSTM*-блока. При прямом проходе обновленное состояние памяти активируется гиперболическим тангенсом и умножается состояние выходных ворот. Таким образом, на результат работы блока у нас влияют две составляющие: значение памяти и ворот.



Узел результатов LSTM-блока

Для того, чтобы снизить ошибку на выходе блока, нам нужно скорректировать значения обоих составляющих. Для этого нам нужно распределить общий градиент ошибки через функцию умножения, которая объединяет два потока информации. То есть умножить известный нам градиент ошибки на производную функции по каждому направлению. Из школьного курса математики мы знаем, что производная от произведения константы на переменную является константа. Мы применяем следующий подход: при определении влияния одного из факторов принимаем, что все остальные составляющие имеют постоянные значения. Следовательно, мы можем составить следующие математические формулы.

$$OUT = OG * tanh(Mem)$$

$$\frac{\partial Out}{\partial OG} = tanh(Mem)$$

$$\frac{\partial Out}{\partial tanh(Mem)} = OG$$

И тогда мы легко распределим производную по обоим направлениям, воспользовавшись нижеследующими математическими формулами.

$$Grad_{OG} = Grad_{Out} * tanh(Mem)$$

$$Grad_{tanh(Mem)} = Ggrad_{Out} * OG$$

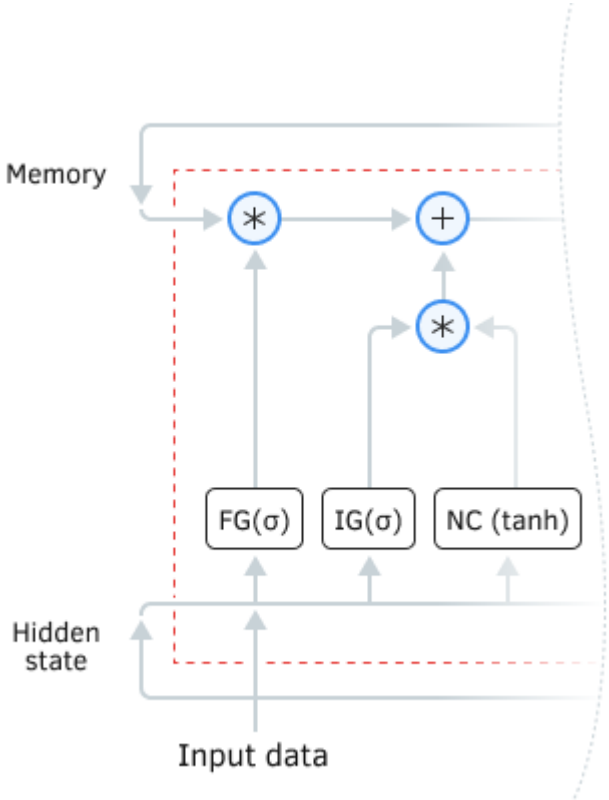
Мы не создавали отдельный буфер для активированного состояния памяти. Но мы легко можем его посчитать, заново активировав соответствующее состояние или разделив скрытое состояние на значение выходных ворот. Я выбрал второй путь, и весь алгоритм распределения градиента ошибки на данном участке выразился в нижеследующем коде.

```
//--- разветвление алгоритма по вычислительному устройству
if(!m_cOpenCL)
  {
   //--- посчитаем градиент на выходе каждого внутреннего слоя
   MATRIX m = hidden.m_mMatrix / (og.m_mMatrix + 1e-8);
   //--- OutputGate градиент
   MATRIX grad = m_cGradients.m_mMatrix;
   og_grad.m_mMatrix = grad * m;
   //--- градиент памяти
   grad *= og.m_mMatrix;
```

Перед распределением градиента памяти по остальным внутренним слоям мы должны скорректировать полученный градиент на производную функции активации.

```
   //--- скорректируем градиент на производную
   grad *= MathPow(m, 2) * (-1) + 1;
```

Продолжаем распределение градиента ошибки межу внутренними слоями. Нам необходимо распределить градиент ошибки от потока памяти еще по трем внутренним слоям. Двигаясь по потоку информации внутри блока *LSTM* в обратном направлении, первой мы встречаем функцию суммирования. Производная суммы равна 1. Поэтому передаем градиент ошибки в обоих направлениях без изменений.



Распределение градиента ошибки внутри LSTM блока

Далее в обоих направлениях мы встречаем произведение. Принципы проведения градиента через произведение двух чисел подробно описано выше, и я не вижу смысла повторять. Я лишь хочу напомнить, что в отличие от всех буферов из стека только буфер памяти мы взяли на один шаг дальше в историю. Я обещал пояснить этот момент — и сейчас для этого самое подходящее время. Посмотрите на схему *LSTM*-блока. Для обновления памяти мы умножаем выход врат забвения на состояние памяти, перешедшее с предыдущей итерации. Следовательно, чтобы

определить градиент ошибки на выходе врат забвения, нам нужно умножить градиент ошибки в потоке памяти на состояние памяти предыдущей итерации. Буфер именно этого состояния мы и загрузили вначале цикла.

MQL5-код описанных операций представлен ниже.

```
   //--- InputGate градиент
   ig_grad.m_mMatrix = grad * nc.m_mMatrix;
   //--- NewContent градиент
   nc_grad.m_mMatrix = grad * ig.m_mMatrix;
   //--- ForgetGates градиент
   fg_grad.m_mMatrix = grad * memory.m_mMatrix;
  }
```

На этом завершается блок разделения потоков по устройству вычислительных операций, и мы объединяем потоки алгоритма. Мы лишь установим «заглушку» для ответвления *OpenCL* и пойдем дальше.

```
else
  {
   return false;
  }
```

Выше мы уже обсуждали необходимость использования исторических состояний буферов результатов внутренних слоев. Теперь нам необходимо внедрить это на практике и заполнить буферы результатов соответствующими историческими данными.

```
//--- скопируем соответствующие исторические данные в буфера внутренних слоев
if(!m_cForgetGate.SetOutputs(fg, false))
   return false;
if(!m_cInputGate.SetOutputs(ig, false))
   return false;
if(!m_cOutputGate.SetOutputs(og, false))
   return false;
if(!m_cNewContent.SetOutputs(nc, false))
   return false;
```

Далее нам необходимо провести градиент от выхода к входу внутренних нейронных слоев. Этот функционал легко реализуется методом базового класса. Но есть одно обстоятельство. Все четыре внутренних нейронных слоя используют одни исходные данные. Также и градиент ошибки нам нужно собрать вместе в одном буфере. Разработанные нами ранее методы базового класса нейронного слоя построены таким образом, что они перезаписывают значения. Поэтому нам необходимо организовать процесс суммирования градиентов ошибки от каждого внутреннего нейронного слоя.

Первым мы проведем градиент через врата забвения. Напомню, что для передачи исходных данных во внутренние нейронные слои мы создавали базовый слой исходных данных и после проведения операций прямого прохода сохраняли указатель на него в стеке исходных данных. Данный тип объектов уже содержит буферы для записи данных и градиента ошибки. Поэтому сейчас мы просто берем этот указатель и передаем в параметрах метода *CNeuronBase::CalcHiddenGradient*. После этого отработает наш метод базового класса и заполнит буфер градиентов ошибки на уровне исходных данных врат забвения. Но это лишь одни врата, а нам нужно собрать информацию со всех. Чтобы не потерять уже посчитанный градиент ошибки

при вызове аналогичного метода других внутренних слоев, мы скопируем данные в созданный нами заранее буфер для накопления градиентов ошибки *m_cInputGradient*.

```
//--- проведем градиент через внутренние слои
if(!m_cForgetGate.CalcHiddenGradient(inputs))
   return false;
if(!m_cInputGradient)
  {
   m_cInputGradient = new CBufferType();
   if(!m_cInputGradient)
      return false;
   m_cInputGradient.m_mMatrix = inputs.GetGradients().m_mMatrix;
   m_cInputGradient.BufferCreate(m_cOpenCL);
  }
else
  {
   m_cInputGradient.Scaling(0);
   if(!m_cInputGradient.SumArray(inputs.GetGradients()))
      return false;
  }
```

Повторим операции для остальных внутренних слоев. Только теперь мы будем новые значения градиента ошибки прибавлять к полученным ранее.

```
if(!m_cInputGate.CalcHiddenGradient(inputs))
   return false;
if(!m_cInputGradient.SumArray(inputs.GetGradients()))
   return false;
if(!m_cOutputGate.CalcHiddenGradient(inputs))
   return false;
if(!m_cInputGradient.SumArray(inputs.GetGradients()))
   return false;
if(!m_cNewContent.CalcHiddenGradient(inputs))
   return false;
if(!inputs.GetGradients().SumArray(m_cInputGradient))
   return false;
```

Стоит обратить внимание на один момент. Если при обработке трех первых внутренних слоев мы переносили значения во временный буфер *m_cInputGradient*, то при обработке последнего слоя мы перенесем накопленный ранее градиент ошибки в буфер слоя исходных данных. Таким образом, мы сохраним общий градиент ошибки на уровне исходных данных вместе с самими исходными данными в одном слое исходных данных. Также он автоматически сохранится в нашем стеке. Вспомните, что я писал об объектах и указателях на них.

И тут возникает один узкий момент. Давайте вспомним для чего мы все это делали. Распространение градиента ошибки по всем элементам нейронной сети нам необходимо, чтобы иметь ориентир, в какую сторону и на сколько нам нужно изменить элементы матриц весов для снижения общей ошибки работы нашей нейронной сети. Следовательно, в результате операций данного метода мы должны:

- довести градиент ошибки до предыдущего слоя;
- довести градиент ошибки до матриц весов внутренних нейронных слоев.

Если в таком состоянии мы запустим следующий цикл итераций с новыми данными по пересчету градиентов ошибки внутренних слоев, то мы просто заменим только что посчитанные значения. А нам необходимо довести градиенты ошибки до матриц весов внутренних нейронных слоев. Поэтому, не дожидаясь вызова от внешней программы, мы вызовем метод *CNeuronBase::CalcDeltaWeights* для всех внутренних слоев, который пересчитает градиент на уровне весовых матриц.

```
      //--- спроецируем градиент на матрицы весов внутренних слоев
      if(!m_cForgetGate.CalcDeltaWeights(inputs))
         return false;
      if(!m_cInputGate.CalcDeltaWeights(inputs))
         return false;
      if(!m_cOutputGate.CalcDeltaWeights(inputs))
         return false;
      if(!m_cNewContent.CalcDeltaWeights(inputs))
         return false;
```

На предыдущий нейронный слой мы будем передавать градиент ошибки только текущего состояния. Исторические данные остаются только для внутреннего пользователя *LSTM*-блока. Поэтому мы проверяем индекс итерации и только потом передаем градиент ошибки в буфер предыдущего слоя. Важно не забыть, что наш буфер градиентов ошибки на уровне исходных данных содержит больше буфера предыдущего слоя. Ведь он содержит еще и градиент ошибки скрытого состояния. Следовательно, на предыдущий слой мы передадим лишь необходимую часть данных.

Остаток перенесем в буфер градиентов ошибки нашего *LSTM*-блока. Помните, в начале цикла именно из этого буфера мы брали градиент ошибки для распространения по всему LSTM блоку? Настало время подготовить исходные данные для следующей итерации нашего цикла по хронологии итераций прямого прохода и распространения градиента ошибки.

```
      //--- если посчитан градиент текущего состояния, то передаем на предыдущий слой
      //--- и запишем градиент скрытого состояния в буфер градиентов для новой итерации
         if(!inputs.GetGradients().Split((i == 0 ? prevLayer.GetGradients() :
                                                   inputs.GetGradients()), m_cGradients,
                                                   prevLayer.GetOutputs().Total()))
            return false;
        }
    //---
      return true;
     }
```

После успешного выполнения всех итераций выходим из метода с положительным результатом.

Мы с вами разобрали два наиболее сложных и запутанных метода построения рекуррентного алгоритма *LSTM*-блока. Оставшаяся часть будет намного проще. К примеру, метод *CalcDeltaWeights*. Функционал этого метода предусматривает передачу градиента ошибки на уровень матрицы весов. Какой-либо отдельной матрицы весов *LSTM*-блок не имеет. Все параметры расположены внутри вложенных нейронных слоев, а градиент ошибки до уровня их матриц весов мы уже довели в предыдущем методе. Поэтому метод переписываем пустой заглушкой с положительным результатом.

```
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer) { return true; }
```

Еще один метод обратного прохода *UpdateWeights* — метод обновления матрицы весов. Метод также наследуется из базового класса нейронного слоя и переопределяется при необходимости. *LSTM*-блок в отличие от ранее рассмотренных типов нейронных слоев не имеет единой матрицы весов. Вместо этого используются внутренние нейронные слои со своими матрицами весов. Поэтому мы не можем просто воспользоваться методом родительского класса и вынуждены переопределить его.

Метод *CNeuronLSTM::UpdateWeights* от внешней программы получает параметры, необходимые для выполнения алгоритма обновления матрицы весов, и возвращает логическое значение результата выполнения операций метода.

Несмотря на то, что в параметрах метода нет ни одного указателя на объект, в начале метода мы все равно организовываем блок контролей. Здесь мы проверяем актуальность указателей на внутренние нейронные слои и значение параметра анализируемой глубины истории, который должен быть больше 0.

```
bool CNeuronLSTM::UpdateWeights(int batch_size, TYPE learningRate, VECTOR &Beta,
                                                                    VECTOR &Lambda)
  {
//--- проверяем состояние объектов
   if(!m_cForgetGate || !m_cInputGate || !m_cOutputGate ||
      !m_cNewContent || m_iDepth <= 0)
      return false;
```

Следует обратить внимание на такой параметр, как размер пакета *batch_size*. Этот параметр указывает на количество итераций обратного прохода между обновлениями весовых коэффициентов. Он отслеживается внешней программой и передается методу в параметрах. Надо понимать, что для внешней программы и для рассмотренных ранее типов нейронных сетей количество прямого и обратного проходов равны, так как за каждым прямым проходом следует обратный проход. В обратном проходе определяется отклонение расчетного результата работы нейронной сети от ожидаемого результата и распространяется градиент ошибки по всей нейронной сети. В случае же рекуррентного блока ситуация немного иная: на каждый прямой проход один рекуррентный блок совершает несколько итераций обратного прохода, определяемых глубиной анализируемой истории. Следовательно, поступивший от внешней программы размер пакета мы должны скорректировать на глубину исторических данных.

```
   int batch = batch_size * m_iDepth;
```

После этого мы можем воспользоваться методами обновления матрицы весов, передав им в параметрах корректные данные.

```
//--- обновляем матрицы весов внутренних слоев
   if(!m_cForgetGate.UpdateWeights(batch, learningRate, Beta, Lambda))
      return false;
   if(!m_cInputGate.UpdateWeights(batch, learningRate, Beta, Lambda))
      return false;
   if(!m_cOutputGate.UpdateWeights(batch, learningRate, Beta, Lambda))
      return false;
   if(!m_cNewContent.UpdateWeights(batch, learningRate, Beta, Lambda))
      return false;
//---
   return true;
  }
```

После успешного обновления матриц весов всех внутренних нейронных слоев выходим из метода с положительным результатом.

На этом мы заканчиваем рассмотрение методов обратного прохода LSTM-блока. Можем двигаться дальше в построении нашей системы.

4.2.2.3 Сохранение и восстановление LSTM-блока

Мы с вами уже рассмотрели методы инициализации прямого и обратного проходов LSTM-блока. Этого достаточно для каких-то не больших экспериментов, но слишком мало для промышленной эксплуатации. Ведь одним из ключевых требований промышленной эксплуатации является повторное многоразовое использование однажды обученной нейронной сети. Мы научились создавать и обучать нашу нейронную сеть. Мы даже можем получать результаты ее работы на реальных данных. Но мы еще не можем сохранить обученный *LSTM*-блок, чтобы потом восстановить его из ранее сохраненных данных. Для выполнения этого функционала в наших классах нейронных слоев предусмотрено два метода:

- *Save* — сохранение класса;
- *Load* — восстановление функций класса по ранее сохраненным данным.

Прежде чем приступить к созданию методов, давайте посмотрим на структуру класса нашего LSTM-блока и определим, какие данные нам необходимо сохранить, а какие мы просто можем инициализировать начальными значениями.

```
class CNeuronLSTM    :  public CNeuronBase
  {
protected:
   CNeuronBase*      m_cForgetGate;
   CNeuronBase*      m_cInputGate;
   CNeuronBase*      m_cNewContent;
   CNeuronBase*      m_cOutputGate;
   CArrayObj*        m_cMemorys;
   CArrayObj*        m_cHiddenStates;
   CArrayObj*        m_cInputs;
   CArrayObj*        m_cForgetGateOuts;
   CArrayObj*        m_cInputGateOuts;
   CArrayObj*        m_cNewContentOuts;
   CArrayObj*        m_cOutputGateOuts;
   CBufferType*      m_cInputGradient;
   int               m_iDepth;

   void              ClearBuffer(CArrayObj *buffer);
   bool              InsertBuffer(CArrayObj *&array, CBufferType *element,
                                  bool create_new = true);
   CBufferType*      CreateBuffer(CArrayObj *&array);

public:
                     CNeuronLSTM(void);
                    ~CNeuronLSTM(void);
   //---
   virtual bool      Init(const CLayerDescription *desc) override;
   virtual bool      SetOpenCL(CMyOpenCL *opencl) override;
   virtual bool      FeedForward(CNeuronBase *prevLayer) override;
   virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer) override
                                                         { return true; }
   virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                   VECTOR &Beta, VECTOR &Lambda) override;
   //---
   virtual int       GetDepth(void)                const { return m_iDepth; }
   //--- методы работы с файлами
   virtual bool      Save(const int file_handle) override;
   virtual bool      Load(const int file_handle) override;
   //--- метод идентификации объекта
   virtual int       Type(void)  override     const { return(defNeuronLSTM); }
  };
```

Прежде всего мы четко понимаем, что никакие константы и методы не изменяются в процессе работы класса. Поэтому сохранять мы будем только переменные величины.

При объявлении переменных класса первыми мы объявили переменные для хранения указателей на внутренние нейронные слои. Конечно, сохранять указатели на объекты класса абсолютно не имеет смысла. А вот сохранить содержимое этих объектов мы просто обязаны — именно в них содержатся обученные матрицы весов.

Далее мы объявляли указатели на стеки хронологических данных. Сами стеки, впрочем как и их содержимое, при сохранении данных для нас не имеют ценности. Стеки — это объекты

динамических массивов, которые без труда будут заново созданы. Касательно их содержимого ситуация следующая. Для рекуррентных сетей очень важна последовательность данных и отсутствие пропусков. На момент сохранения данных мы не понимаем, когда повторно они будут использоваться. Следовательно, на момент загрузки данных очень велика вероятность наличия пропусков между текущим состоянием анализируемой системы и данными на момент сохранения. В такой ситуации их использование будет не только не полезным, но и наоборот исказит результаты. Поэтому сохранение этих данных только увеличит объем сохраняемой информации без какой-либо пользы для последующего использования.

Буфер накопления градиентов ошибки *m_cInputGradient* является вспомогательным объектом для накопления данных и перезаписывается новыми данными при каждом обратном проходе. Он не содержит информации важной для последующих итераций и не целесообразен для сохранения.

Последняя глобальная переменная, которую мы объявляли, это глубина анализируемых хронологических итераций *m_iDepth*. Она является составляющей архитектурного решения блока и подлежит сохранению.

После определения масштаба работы мы можем перейти к ее выполнению. Сначала создадим метод сохранения данных *CNeuronLSTM::Save*. В параметрах метод получает хендл файла для сохранения данных. Но мы не будем организовывать контрольный блок проверки поступивших параметров, как обычно. Вместо этого мы передадим полученный параметр в аналогичный метод базового класса, в котором уже реализованы все необходимые контроли. Кроме того, выше мы с вами проанализировали только переменные, объявленные в классе *LSTM*-блока, но не оценили необходимость сохранение содержимого родительского класса. Зато эту работу мы провели при создании метода сохранения данных базового класса. Поэтому, вызывая метод родительского класса, мы в одной строке кода выполняем два вышеуказанных функционала.

```
bool CNeuronLSTM::Save(const int file_handle)
  {
//--- вызов метода родительского класса
   if(!CNeuronBase::Save(file_handle))
      return false;
```

После успешного выполнения метода родительского класса мы сохраняем значение глубины анализируемых хронологических итераций.

```
//--- сохраняем константы
   if(FileWriteInteger(file_handle, m_iDepth) <= 0)
      return false;
```

После этого нам остается сохранить содержимое внутренних нейронных слоев. Для этого мы также воспользуемся функционалом базового нейронного слоя. Нам надо лишь вызвать метод сохранения для каждого нашего внутреннего слоя с указанием хендла файла для записи данных, который мы получили в параметрах от внешней программы. При этом не забываем на каждом шаге контролировать процесс выполнения операций.

```
//--- вызываем аналогичный метод для всех внутренних слоев
   if(!m_cForgetGate.Save(file_handle))
      return false;
   if(!m_cInputGate.Save(file_handle))
      return false;
   if(!m_cOutputGate.Save(file_handle))
      return false;
   if(!m_cNewContent.Save(file_handle))
      return false;
//---
   return true;
  }
```

После успешного выполнения всех операций выходим из метода с положительным результатом.

А теперь предлагаю еще раз посмотреть на весь код метода сохранения данных и оценить, насколько он краток и читабелен. Этот эффект достигается благодаря использованию объектно-ориентированного программирования (*ООП*). Создание классов позволяет заметно сократить код и ускорить работу программиста, а использование готовых и протестированных библиотек помогает избежать многих ошибок. И поверьте, каким бы сложным ни показалось создание нашей библиотеки, ее использование позволит легко и без особых трудозатрат со стороны программиста создавать свои нейронные сети. И для этого не нужно быть программистом высокой квалификации.

Но я отвлекся. Предлагаю вернуться к нашему процессу. Мы создали метод сохранения данных. Осталось построить процесс восстановления работоспособности нашего рекуррентного блока из сохраненных данных.

Метод загрузки данных *CNeuronLSTM::Load* строится в четком соответствии с методом сохранения данных. Сохраненные данные должны быть загружены из файла в той же последовательности, иначе мы можем получить как искаженные данные, так и ошибку загрузки.

В параметрах метод получает хендл файла с данными для загрузки. Как и при сохранении данных, вместо организации блока контролей мы вызываем метод родительского класса. В нем уже реализованы все необходимые контроли и загрузка данных родительского класса.

```
bool CNeuronLSTM::Load(const int file_handle)
  {
//--- вызов метода родительского класса
   if(!CNeuronBase::Load(file_handle))
      return false;
```

Далее мы загружаем из файла глубину анализируемых хронологических итераций и содержимое внутренних нейронных слоев. Для выполнения последних операций мы также воспользуемся методами базового класса нейронного слоя. И, как всегда, проверяем результат выполнения операций.

Но здесь надо обратить внимание на один немаловажный момент. Метод сохранения базового нейронного слоя *CNeuronBase::Save* начинается с записи типа сохраняемого объекта. Его значение мы считываем в диспетчерском методе загрузки нейронной сети, чтобы определить тип создаваемого объекта. И следовательно, в методе загрузки нейронного слоя мы начинаем считывание файла со следующего элемента. В данном случае, чтобы сохранить последовательность загрузки данных из файла, мы должны сначала прочитать тип следующего нейронного слоя и только потом вызывать метод загрузки соответствующего внутреннего

нейронного слоя. К тому же это может быть дополнительной точкой контроля загрузки корректного типа внутреннего нейронного слоя.

```
//--- считываем константы
   m_iDepth = FileReadInteger(file_handle);
//--- вызываем аналогичный метод для всех внутренних слоев
   if(FileReadInteger(file_handle) != defNeuronBase ||
      !m_cForgetGate.Load(file_handle))
      return false;
   if(FileReadInteger(file_handle) != defNeuronBase ||
      !m_cInputGate.Load(file_handle))
      return false;
   if(FileReadInteger(file_handle) != defNeuronBase ||
      !m_cOutputGate.Load(file_handle))
      return false;
   if(FileReadInteger(file_handle) != defNeuronBase ||
      !m_cNewContent.Load(file_handle))
      return false;
```

После загрузки данных из файла нам необходимо оставшиеся объекты инициализировать начальными значениями. Сначала мы инициализируем стек памяти и добавим в него буфер с начальными значениями. Для этого мы воспользуемся уже знакомым нам методом *CreateBuffer*. Напомню, что данный метод создает буфер с нулевыми значениями только для пустого стека. Иначе метод вернет последний записанный буфер. Поэтому перед вызовом метода мы проверяем размер стека: если стек содержит данные, то мы очищаем стек и обнуляем все значения буфера.

```
//--- инициализируем Memory
   if(m_cMemorys.Total() > 0)
      m_cMemorys.Clear();
   CBufferType *buffer =  CreateBuffer(m_cMemorys);
   if(!buffer)
      return false;
   if(!m_cMemorys.Add(buffer))
      return false;
```

После выполнения всех операций добавляем вновь созданный буфер в стек. Аналогичные операции повторяем для стека и буфера скрытого состояния.

```
//--- инициализируем HiddenStates
   if(m_cHiddenStates.Total() > 0)
      m_cHiddenStates.Clear();
   buffer =  CreateBuffer(m_cHiddenStates);
   if(!buffer)
      return false;
   if(!m_cHiddenStates.Add(buffer))
      return false;
```

Метод прямого прохода мы построили таким образом, что для нас не критично сейчас создавать и инициализировать остальные стеки. Но мы допускаем, что операция загрузки данных может быть выполнена на рабочей нейронной сети, когда в стеках уже есть какая-то информация. В таких случаях использование данных из стеков, созданных при использование других весовых коэффициентов, будет некорректным. Поэтому мы очистим все ранее созданные стеки.

```
//--- очищаем остальные стеки
   if(!m_cInputs)
      m_cInputs.Clear();
   if(!m_cForgetGateOuts)
      m_cForgetGateOuts.Clear();
   if(!m_cInputGateOuts)
      m_cInputGateOuts.Clear();
   if(!m_cNewContentOuts)
      m_cNewContentOuts.Clear();
   if(!m_cOutputGateOuts)
      m_cOutputGateOuts.Clear();
//---
   return true;
  }
```

После успешного выполнения всех операций метода мы завершаем его работу с положительным результатом.

На этом мы завершаем построение рекуррентного *LSTM*-блока средствами *MQL5* и переходим к дополнению методов нашего класса возможностями совершения многопоточных операций.

4.2.3 Организация параллельных вычислений в LSTM-блоке

В предыдущих главах мы рассмотрели реализацию *LSTM*-блока средствами *MQL5*. Но новый этап развития нейронных сетей настал именно с развитием технологий параллельных вычислений. Особенно это важно для ресурсоемких задач, таких как рекуррентные нейронные сети. Поэтому нам особенно важно добавить возможность использования средств многопоточных параллельных вычислений в класс *LSTM*-блока.

Как было сказано при создании алгоритма блока средствами *MQL5*, в нашем классе уже есть реализация многопоточных вычислений отдельных блоков благодаря использованию объектов ранее рассмотренного класса базового нейронного слоя в качестве ворот в алгоритме *LSTM*-блока. Поэтому в рамках данной главы нам предстоит реализовать лишь недостающую часть:

- поток консолидации и обработки данных от внутренних нейронных слоев в рамках прямого прохода;
- распределение градиента ошибки от выхода *LSTM* блока до внутренних нейронных слоев в рамках обратного прохода.

Это дает нам понимание задания.

Мы уже имеем реализацию процесса средствами *MQL5*. Это дает понимание процесса и алгоритма выполнения операций.

Следовательно, мы можем приступить к работе. Напомню архитектуру построения процесса многопоточных вычислений. Непосредственно выполнение процесса вычислений в параллельных потоках осуществляется в отличной от основной программы среде — контексте *OpenCL*. Для выполнения операций необходимо три основных составляющих:

1. Программа выполняемых операций.
2. Исходные данные для выполнения операций.

3. Команды управления процессом (момент запуска программы, количество создаваемых потоков и т.д.)

Давайте рассмотрим выполнение этих пунктов.

4.2.3.1 Внесение дополнений в программу OpenCL

Первым пунктом у нас указана программы выполняемых операций. Это значит, что нам необходимо дополнить нашу программу *OpenCL* новыми кернелами для выполнения требуемых нам дополнительных операций. Весь код программы OpenCL мы собрали в файле *opencl_program.cl*. Открываем данный файл и дополняем его двумя новыми кернелами: *LSTMFeedForward* и *LSTMCalcHiddenGradient*. Названия кернелов созвучны с названиями методов наших классов. Поэтому несложно догадаться, что первый будет дополнять метод прямого прохода, а второй — метод распределения градиента ошибки.

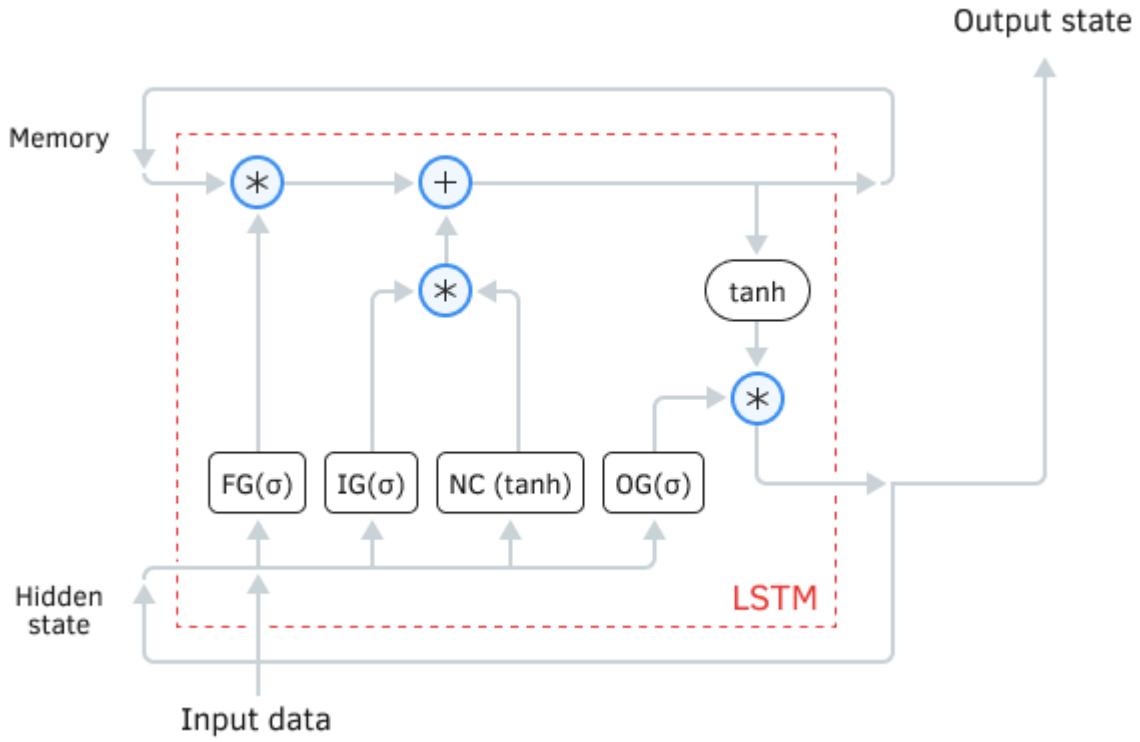


Схема рекуррентного LSTM блока

Начнем мы с кернела прямого прохода *LSTMFeedForward*. В параметрах данный буфер получит указатели на шесть буферов данных (четыре буфера исходных данных и два буфера результатов) и одну константу:

- *forgetgate* — указатель на буфер врат забвения (исходные данные);
- *inputgate* — указатель на буфер входных врат (исходные данные);
- *outputgate* — указатель на буфер врат результатов (исходные данные);
- *newcontent* — указатель на буфер нового контента (исходные данные);
- *memory* — указатель на поток памяти (буфер результатов);
- *hiddenstate* — указатель на поток скрытого состояния (буфер результатов);

- *outputs_total* — количество элементов в потоке данных (константа).

```
__kernel void LSTMFeedForward(__global TYPE *forgetgate,
                              __global TYPE *inputgate,
                              __global TYPE *outputgate,
                              __global TYPE *newcontent,
                              __global TYPE *memory,
                              __global TYPE *hiddenstate,
                              int outputs_total)
```

В начале метода мы, как и раньше, получаем индекс потока, который служит нам указателем на обрабатываемые данные. И сразу же определяем сдвиг в буферах данных для доступа к нужным нам данным.

```
{
 const int n = get_global_id(0);
 const int shift = n * 4;
```

Для повышения производительности нашей программы мы будем использовать векторную арифметику. Воспользуемся векторными переменными типа *TYPE4*. Напомню, что мы используем макроподстановку *TYPE* для быстрого переключения используемого типа данных *double* или *float* в зависимости от требований точности вычислений и используемого *OpenCL*-устройства. Но прежде чем начать проведение операций, мы перенесем данные из наших глобальных буферов данных в локальные векторные переменные.

```
TYPE4 fg = ToVect4(forgetgate, shift, 1, outputs_total, 0);
TYPE4 ig = ToVect4(inputgate, shift, 1, outputs_total, 0);
TYPE4 og = ToVect4(outputgate, shift, 1, outputs_total, 0);
TYPE4 nc = ToVect4(newcontent, shift, 1, outputs_total, 0);
TYPE4 mem = ToVect4(memory, shift, 1, outputs_total, 0);
```

Теперь, по аналогии с кодом программы на *MQL5*, выполним арифметические действия по обновлению состояния потока памяти. Согласно алгоритму работы *LSTM*-блока сначала мы должны скорректировать входящий поток памяти на значение врат забвения, а потом прибавить к полученному значению новый контекст, скорректированный значением входных врат. После выполнения операций значение обновленного потока памяти вернем обратно в буфер.

```
TYPE4 temp = mem * fg;
temp += ig * nc;
D4ToArray(memory, temp, shift, 1, outputs_total, 0);
```

Далее нам необходимо определить новое значение потока скрытого состояния. Оно же будет подаваться на выход *LSTM*-блока для передачи в следующий нейронный слой. Здесь нам необходимо сначала нормализовать текущее состояние памяти с помощью функции гиперболического тангенса, а затем скорректировать полученное значение на значение врат результатов. Результат операций записываем в буфер данных.

```
 temp = tanh(temp) * og;
 D4ToArray(hiddenstate, temp, shift, 1, outputs_total, 0);
}
```

На этом операции кернела прямого прохода завершены. Из результатов работы внутренних слоев нашего рекуррентного *LSTM*-блока мы обновили состояние потока памяти и получили значения для вывода на выход рекуррентного блока.

Во втором кернеле *LSTMCalcHiddenGradient* нам необходимо осуществить обратную операцию, то есть провести градиент ошибки в обратном направлении, от выхода рекуррентного блока до выхода каждого внутреннего нейронного слоя. Специфика работы кернела обратного прохода требует увеличение количества используемых буферов данных до 10:

- *outputs* — указатель на буфер вектора результатов (исходные данные);
- *gradients* — указатель на буфер вектора градиентов текущего слоя (исходные данные);
- *inputgate* — указатель на буфер входных врат (исходные данные);
- *outputgate* — указатель на буфер врат результатов (исходные данные);
- *newcontent* — указатель на буфер нового контента (исходные данные);
- *memory* — указатель на поток памяти (исходные данные);
- *fg_gradients* — указатель на буфер градиентов врат забвения (буфер результатов);
- *ig_gradients* — указатель на буфер градиентов входных врат (буфер результатов);
- *og_gradients* — указатель на буфер градиентов врат результатов (буфер результатов);
- *nc_gradients* — указатель на буфер градиентов нового контента (буфер результатов);
- *outputs_total* — количество элементов в потоке данных (константа).

```
__kernel void LSTMCalcHiddenGradient(__global TYPE *outputs,
                                     __global TYPE *gradients,
                                     __global TYPE *inputgate,
                                     __global TYPE *outputgate,
                                     __global TYPE *newcontent,
                                     __global TYPE *memory,
                                     __global TYPE *fg_gradients,
                                     __global TYPE *ig_gradients,
                                     __global TYPE *og_gradients,
                                     __global TYPE *nc_gradients,
                                     int outputs_total)
```

В начале кернела мы определяем ID потока и смещение в буферах данных до обрабатываемых значений.

```
  {
   const int n = get_global_id(0);
   int shift = n * 4;
```

Как и в кернеле прямого прохода, мы будем использовать операции с векторными переменными типа *TYPE4*. Поэтому на следующем шаге мы переносим исходные данные из глобальных буферов в локальные векторные переменные.

```
   TYPE4 out = ToVect4(outputs, shift, 1, outputs_total, 0);
   TYPE4 grad = ToVect4(gradients, shift, 1, outputs_total, 0);
   TYPE4 ig = ToVect4(inputgate, shift, 1, outputs_total, 0);
   TYPE4 og = ToVect4(outputgate, shift, 1, outputs_total, 0);
   TYPE4 nc = ToVect4(newcontent, shift, 1, outputs_total, 0);
   TYPE4 mem = ToVect4(memory, shift, 1, outputs_total, 0);
```

После завершения подготовительных операций переходим непосредственно к выполнению математической части кернела. Формулы проведения операций и их объяснение представлены при описании построения процесса средствами MQL5. Поэтому в данном разделе будет приведена лишь реализация процесса в *OpenCL*.

Напомню, что при реализации средствами *MQL5* мы определились, что нецелесообразно создавать дополнительный буфер данных для хранения нормализованной величины потока памяти. В параметрах кернела мы получили указатель на поток не текущего состояния памяти, а поступающего на вход рекуррентного блока с предыдущей итерации прямого прохода. Поэтому, прежде чем продолжать операции по распределению градиента ошибки, нам необходимо найти значение нормализованного состояния потока памяти. Его мы определяем как отношение значения буфера результатов к значению врат результатов. Для исключения деления на ноль добавим небольшую константу в знаменателе.

```
TYPE4 m = out / (og + 1.0e-37f);
```

Следуя логике алгоритма обратного распространения градиента ошибки, сначала мы определим градиент ошибки на выходе нейронного слоя ворот забвения. Для этого нам нужно умножить градиент ошибки на выходе нашего *LSTM*-блока на производную произведения. В данном случае она равна значению нормализованного состояния памяти. Полученное значение сразу запишем в соответствующий буфер данных.

```
//--- OutputGate градиент
   TYPE4 temp = grad * m;
   D4ToArray(og_gradients, temp, shift, 1, outputs_total, 0);
```

Далее мы должны аналогичным путем определить градиент ошибки на другом множителе — нормализованное состояние памяти. То есть умножаем градиент ошибки на выходе нашего рекуррентного блока на состояние врат результатов.

И прежде чем продолжить распространение градиента до остальных нейронных слоев, надо провести его через функцию гиперболического тангенса. Иными словами, мы умножаем ранее полученное значении на производную гиперболического тангенса.

$$\frac{df(x)}{dx} = (1 + f(x))(1 - f(x)) = 1 - (f(x))^2$$

```
//--- Градиент памяти корректируем на производную TANH
   grad = grad * og * (1 - pow(m, 2));
```

После этого нам остается лишь распределить градиент ошибки по оставшимся внутренним слоям. Алгоритм будет один для всех нейронных слоев — отличия лишь в использованном буфере в качестве производной функции умножения. После определения градиента ошибки сразу записываем его значение в соответствующий буфер.

```
//--- InputGate градиент
   temp = grad * nc;
   D4ToArray(ig_gradients, temp, shift, 1, outputs_total, 0);
//--- NewContent градиент
   temp = grad * ig;
   D4ToArray(nc_gradients, temp, shift, 1, outputs_total, 0);
//--- ForgetGates градиент
   temp = grad * mem;
   D4ToArray(fg_gradients, temp, shift, 1, outputs_total, 0);
  }
```

После выполнения операций выходим из кернела.

Таким образом, мы реализовали недостающие кернелы для организации прямого и обратного проходов в рамках выполнения операций для рекуррентного *LSTM*-блока. На этом модификация

программы *OpenCL* завершена, и мы переходим к выполнению операций на стороне основной программы.

4.2.3.2 Реализация функционала на стороне основной программы

После внесения изменений в программу *OpenCL* мы должны выполнить вторую часть работы и организовать процесс на стороне основной программы. И первое, что мы сделаем, — это создадим константы для работы с кернелами. Здесь нам необходимо создать константы для идентификации кернелов и их параметров. Указанные константы добавим к ранее созданным в файле *defines.mqh*.

```
#define def_k_LSTMFeedForward          26
#define def_k_LSTMHiddenGradients      27

//--- LSTM Feed Forward
#define def_lstmff_forgetgate          0
#define def_lstmff_inputgate           1
#define def_lstmff_outputgate          2
#define def_lstmff_newcontent          3
#define def_lstmff_memory              4
#define def_lstmff_hiddenstate         5
#define def_lstmff_outputs_total       6
```

При добавлении констант соблюдаем определенные ранее правила именования. Все константы кернелов начинаются с приставки *def_k_*, а константы параметров содержат аббревиатуру кернела: *def_lstmff_* для параметров кернела прямого прохода и *def_lstmhgr_* у параметров кернела распределения градиента.

```
//--- LSTM Hidden Gradients
#define def_lstmhgr_outputs            0
#define def_lstmhgr_gradients          1
#define def_lstmhgr_inputgate          2
#define def_lstmhgr_outputgate         3
#define def_lstmhgr_newcontent         4
#define def_lstmhgr_memory             5
#define def_lstmhgr_fg_gradients       6
#define def_lstmhgr_ig_gradients       7
#define def_lstmhgr_og_gradients       8
#define def_lstmhgr_nc_gradients       9
#define def_lstmhgr_outputs_total      10
```

Затем мы переходим в файл *neuronnet.mqh*, который содержит код нашего класса нейронной сети. Находим метод *CNet::InitOpenCL* — в нем нам надо изменить количество используемых кернелов и одновременно открытых буферов.

```
   if(!m_cOpenCL.SetKernelsCount(28))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
   if(!m_cOpenCL.SetBuffersCount(10))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
```

Изменение последнего параметра для нас не критично, так как в нашем методе создания буфера мы при необходимости изменяем размер массива для хранения хендлов буферов. Но при использовании стандартной библиотеки *OpenCL.mqh* такого функционала нет. Это может привести к ошибке выполнения.

Далее мы объявим кернелы для использования в рамках нашей программы. При этом не забываем контролировать процесс выполнения операций.

```
   if(!m_cOpenCL.KernelCreate(def_k_LSTMFeedForward, "LSTMFeedForward"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }

   if(!m_cOpenCL.KernelCreate(def_k_LSTMHiddenGradients, "LSTMCalcHiddenGradient"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
```

На этом подготовительная работа закончена, и мы переходим к внесению изменений непосредственно в код исполняемых методов нашего класса рекуррентного *LSTM* блока.

По хронологии выполнения алгоритма работы нашей нейронной сети первым мы внесем изменения в метод прямого прохода. В нем мы сначала организуем проверку наличие данных в памяти контекста *OpenCL*.

```
bool CNeuronLSTM::FeedForward(CNeuronBase *prevLayer)
  {
   ....
//--- Разветвление алгоритма по вычислительному устройству
   CBufferType *fg = m_cForgetGate.GetOutputs();
   CBufferType *ig = m_cInputGate.GetOutputs();
   CBufferType *og = m_cOutputGate.GetOutputs();
   CBufferType *nc = m_cNewContent.GetOutputs();
   if(!m_cOpenCL)
     {
     // Здесь пропущен Блок MQL5
     }
   else // Блок работы с OpenCL
     {
      //--- проверяем буферы
      if(fg.GetIndex() < 0 || ig.GetIndex() < 0 || og.GetIndex() < 0 ||
         nc.GetIndex() < 0 || memory.GetIndex() < 0 || hidden.GetIndex() < 0)
         return false;
```

Затем мы передаем указатели на созданные буферы в параметры нашего кернела. Здесь же мы указывает константы, необходимые для корректного выполнения кода программы. И не забываем проверить результат операций.

```
      //--- передаем параметры кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMFeedForward, def_lstmff_forgetgate,
                                                                  fg.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMFeedForward, def_lstmff_inputgate,
                                                                  ig.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMFeedForward, def_lstmff_newcontent,
                                                                  nc.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMFeedForward, def_lstmff_outputgate,
                                                                  og.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMFeedForward, def_lstmff_memory,
                                                                memory.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMFeedForward, def_lstmff_hiddenstate,
                                                                hidden.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_LSTMFeedForward, def_lstmff_outputs_total,
                                                              m_cOutputs.Total()))
         return false;
```

На этом закончен этап подготовительной работы. Переходим к запуску кернела на выполнения операций. Но для начала определим количество необходимых потоков. Напомню, что в теле кернела мы используем векторные операции, и поэтому количество потоков будет в четыре раза меньше размера буферов.

Рассчитанное количество потоков запишем в массив *NDRange*, а нулевое смещение в буферах данных укажем в массиве *off_set*. Поставим наш кернел в очередь выполнения. В случае

возникновения ошибки постановки кернела в очередь выполнения функция *m_cOpenCL.Execute* вернет результат *false*, который мы должны проверить и обработать.

```
      //--- запуск кернела
      int NDRange[] = {(int)(m_cOutputs.Total() + 3) / 4};
      int off_set[] = {0};
      if(!m_cOpenCL.Execute(def_k_LSTMFeedForward, 1, off_set, NDRange))
         return false;
     }
```

На этом работа над методом прямого прохода *LSTM*-блока завершена. Переходим к внесению дополнений в метод распределения градиента ошибки.

Как и в случае прямого прохода, работу в методе распределения градиента ошибки *CNeuronLSTM::CalcHiddenGradient* мы начнем с проверки наличия исходных данных в памяти контекста *OpenCL*.

```
bool CNeuronLSTM::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
   ....
      //--- Разветвление алгоритма по вычислительному устройству
      if(!m_cOpenCL)
        {
         // Здесь пропущен Блок MQL5
        }

      else // Блок работы с OpenCL
        {
         //--- проверяем буферы
         if(hidden.GetIndex() < 0)
            return false;
         if(m_cGradients.GetIndex() < 0)
            return false;
         if(ig.GetIndex() < 0)
            return false;
         if(og.GetIndex() < 0)
            return false;
         if(nc.GetIndex() < 0)
            return false;
         if(memory.GetIndex() < 0)
            return false;
         if(fg_grad.GetIndex() < 0)
            return false;
         if(ig_grad.GetIndex() < 0)
            return false;
         if(og_grad.GetIndex() < 0)
            return false;
         if(nc_grad.GetIndex() < 0)
            return false;
```

Далее полностью повторяем алгоритм работы с кернелами *OpenCL* на стороне основной программы. После создания необходимых буферов в памяти контекста *OpenCL* мы передаем

хендлы буферов данных и значения переменных в параметры кернела. И очень важно проконтролировать выполнение всех операций процесса.

```
//--- передаем параметры кернелу
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                def_lstmhgr_fg_gradients, fg_grad.GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                               def_lstmhgr_gradients, m_cGradients.GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                def_lstmhgr_ig_gradients, ig_grad.GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                     def_lstmhgr_inputgate, ig.GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                    def_lstmhgr_memory, memory.GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                def_lstmhgr_nc_gradients, nc_grad.GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                    def_lstmhgr_newcontent, nc.GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                def_lstmhgr_og_gradients, og_grad.GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                    def_lstmhgr_outputgate, og.GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_LSTMHiddenGradients,
                                   def_lstmhgr_outputs, hidden.GetIndex()))
   return false;
if(!m_cOpenCL.SetArgument(def_k_LSTMHiddenGradients,
                               def_lstmhgr_outputs_total, m_cOutputs.Total()))
   return false;
```

На этом заканчивается этап подготовительной работы Переходим к процедуре запуска кернела. Прежде всего мы здесь записываем количество запускаемых потоков в массив *NDRange* и нулевое смещение в массив *off_set*.

Напомню, что благодаря использованию векторных операций в теле кернела для полного цикла выполнения операций нам требуется в четыре раза меньше потоков. Поэтому, прежде чем записать значение в массив *NDRange*, нам нужно его рассчитать.

После этого отправим наш кернел в очередь выполнения.

```
   //--- запуск кернела
   int NDRange[] = { (int)(m_cOutputs.Total() + 3) / 4 };
   int off_set[] = {0};
   if(!m_cOpenCL.Execute(def_k_LSTMHiddenGradients, 1, off_set, NDRange))
      return false;
  }
```

Наверное, будет звучать навязчиво, но повторю: не забываем проверять результат выполнения каждой операции. Это очень важный момент, так как любая ошибка в выполнении операции может как исказить весь результат работы нашей нейронной сети, так и вызвать критическую ошибку. А это приведет к остановке работы всей программы.

На этом мы полностью завершили работу над классом рекуррентного *LSTM*-блока. В нем мы полностью организовали работу класса в двух средах:

- организовали работу на CPU стандартными средствами MQL5;
- создали возможность осуществления многопоточных параллельных вычислений средствами OpenCL.

Теперь мы можем оценить результат своей работы по средствам создания и тестирования рекуррентной модели нейронной сети.

4.2.4 Реализация рекуррентных моделей в Python

В предыдущих разделах мы рассмотрели принципы организации архитектуры рекуррентных моделей и даже построили рекуррентный нейронный слой с использованием алгоритма *LSTM*-блока. Для построения предыдущих моделей нейронных сетей на языке Python ранее мы использовали библиотеку *Keras* для *TensorFlow*. Эта же библиотека предлагает нам целый ряд вариант для построения рекуррентных нейронных слоев. Среди них есть как классы базовых рекуррентных нейронных слоев, так и более сложные модели.

- *AbstractRNNCell* — абстрактный объект, представляющий ячейку RNN;
- *Bidirectional* — двунаправленная оболочка для RNN;
- *ConvLSTM1D* — 1D сверточный LSTM-блок;
- *ConvLSTM2D* — 2D сверточный LSTM-блок;
- *ConvLSTM3D* — 3D сверточный LSTM-блок;
- *GRU* — рекуррентный блок с вратами Cho et al. 2014;
- *LSTM* — слой долговременной кратковременной памяти Hochreiter 1997;
- *RNN* — базовый класс для рекуррентного слоя;
- *SimpleRNN* — полносвязный рекуррентный слой, в котором выход должен быть возвращен на вход.

В представленном списке вы можете найти кроме базового класса рекуррентного слоя уже знакомые нам *LSTM* и *GRU* модели. Также есть возможность создания двунаправленных рекуррентных слоев, которые чаще всего используются в задачах перевода текстов. Есть модель *ConvLSTM*, которая построена по архитектуре *LSTM*-блока, но вместо полносвязных слоев в качестве ворот и слоя нового контента использует сверточные слои.

Также есть класс абстрактной рекуррентной ячейки для создания пользовательских архитектурных решений рекуррентных моделей.

Мы не будем сейчас глубоко погружаться в API библиотеки *Keras*. Для создания своих тестовых рекуррентных моделей мы воспользуемся *LSTM*-блоком. Именно такую модель мы воссоздали средствами MQL5 и сможем сравнить работу наших моделей, созданных на разных языках программирования.

Класс *LSTM*-блока построен таким образом, что в зависимости от доступного аппаратного обеспечения и ограничений среды выполнения для максимальной производительности автоматически будет выбран вариант реализации на основе *CuDNN* или на чистом *TensorFlow*.

При этом пользователю предлагается довольно широкий спектр параметров для тонкой настройки рекуррентного блока:

- *units* — размерность выходного пространства;
- *activation* — функция активации для использования;
- *recurrent_activation* — функция активации для рекуррентного шага (врат);
- *use_bias* — флаг использования вектора смещения;
- *kernel_initializer* — метод инициализации матрицы весов для слоя нового контекста;
- *recurrent_initializer* — метод инициализации матрицы весов для врат;
- *bias_initializer* — метод инициализации для вектора смещения;
- *kernel_regularizer* — функция регуляризации матрицы весов слоя нового контента;
- *recurrent_regularizer* — функция регуляризации матрицы весов врат;
- *bias_regularizer* — функция регуляризации вектора смещения;
- *activity_regularizer* — функция регуляризации выходного слоя;
- *kernel_constraint* — функция ограничений матрицы весов слоя нового контента;
- *recurrent_constraint* — функция ограничений матрицы весов врат;
- *bias_constraint* — функция ограничений вектора смещения;
- *dropout* — число с плавающей запятой от 0 до 1, определяющее долю отбрасываемых элементов при линейном преобразования входных данных;
- *recurrent_dropout* — число с плавающей запятой от 0 до 1, определяющее долю отбрасываемых элементов при линейном преобразовании состояния памяти;
- *return_sequences* — логический флаг для указания возвращать ли последний результат в выходной последовательности или результаты всей последовательности;
- *return_state* — логический флаг для указания возвращать ли последнее состояние в дополнение к выводу;
- *go_backwards* — логический флаг для указания обрабатывать входную последовательность в обратном порядке и вернуть обратную последовательность;
- *stateful* — логический флаг для указания использовать последнее состояние для каждой выборки с индексом *i* в пакете в качестве начального состояния для выборки с индексом *i* в следующем пакете;
- *time_major* — формат формы тензоров входной и выходной последовательности;
- *unroll* — логический флаг для указания разворачивать рекуррентную сеть или использовать простой цикл, развертывание может ускорить обучение рекуррентной сети, но для этого требуется больше памяти.

После ознакомления с параметрами управления классом *LSTM* слоя посмотрим на практическую реализацию различных моделей с использованием рекуррентного слоя.

4.2.4.1 Построение тестовой рекуррентной модели Python

Для построения тестовых рекуррентных моделей на языке *Python* воспользуемся разработанным ранее шаблоном. Более того, мы возьмем файл скрипта *convolution.py*, который мы использовали при тестровании сверточных моделей. Создадим его копию с именем файла *lstm.py*. В созданной копии мы оставим модель перцептрона и лучшую сверточную модель, удалив остальные. Такой подход позволит нам сравнить работу новых моделей с рассмотренными ранее архитектурными решениями.

```
# Создание модели перцептрона с тремя скрытыми слоями и регуляризацией
model1 = keras.Sequential([keras.Input(shape=inputs),
                           keras.layers.Dense(40, activation=tf.nn.swish,
                  kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(40, activation=tf.nn.swish,
                  kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(40, activation=tf.nn.swish,
                  kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(targerts, activation=tf.nn.tanh)
                          ])

# Модель с 2-мерным сверточным слоем
model3 = keras.Sequential([keras.Input(shape=inputs),
                           # Переформатируем тензор в 4-мерный.
   # Указываем 3 измерения, т.к. 4-е измерение определяется размером пакета
                           keras.layers.Reshape((-1,4,1)),
                           # Сверточный слой с 8-ю фильтрами
                           keras.layers.Conv2D(8,(3,1),1,activation=tf.nn.swish,
                  kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           # Подвыборочный слой
                           keras.layers.MaxPooling2D((2,1),strides=1),
                  # Переформатируем тензор в 2-мерный для полносвязных слоев
                           keras.layers.Flatten(),
                           keras.layers.Dense(40, activation=tf.nn.swish,
                  kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(40, activation=tf.nn.swish,
                  kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(40, activation=tf.nn.swish,
                  kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(targerts, activation=tf.nn.tanh)
                          ])
```

После этого мы создадим три новых модели с использованием рекуррентного *LSTM*-блока. Вначале мы возьмем модель сверточной нейронной сети и заменим в ней сверточный и подвыборочный слои на один рекуррентный слой с 40 нейронами на выходе. Здесь надо сказать, что на вход рекуррентного *LSTM*-блока должен подаваться трехмерный тензор формата [*batch*, *timesteps*, *feature*]. Как и в случае сверточного слоя, при задании размерности слоя в модели мы не указываем измерение batch, так как значение берется из размерности пакета исходных данных.

```
# Добавляем в модель LSTM-блок
model2 = keras.Sequential([keras.Input(shape=inputs),
# Переформатируем тензор в 3-мерный.
# Указываем 2 измерения, т.к. 3-е измерение определяется размером пакета
                           keras.layers.Reshape((-1,4)),
# LSTM-блок содержит 40 элементов и возвращает результата на каждом шаге
                           keras.layers.LSTM(40, return_sequences=False,
                    kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
```

В данной модели мы указали параметр *return_sequences=False*, который указывает рекуррентному слою выдавать результат только после обработки полного пакета. В таком варианте наш *LSTM*-слой возвращает двухмерный тензор в формате [*batch*, *feature*]. При этом размерность измерения *feature* будет равна количеству нейронов, которое мы указали во время создания рекуррентного слоя. Тензор такой же размерности требуется на вход полносвязного нейронного слоя. Следовательно, от нас не требуется дополнительное переформатирование данных, и мы можем использовать полносвязный нейронный слой.

```
                           keras.layers.Dense(40, activation=tf.nn.swish,
                    kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(40, activation=tf.nn.swish,
                    kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(40, activation=tf.nn.swish,
                    kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(targerts, activation=tf.nn.tanh)
                          ])
```

Layer (type)	Output Shape	Param #
reshape (Reshape)	(None, 40, 4)	0
lstm (LSTM)	(None, 40)	7200
dense_4 (Dense)	(None, 40)	1640
dense_5 (Dense)	(None, 40)	1640
dense_6 (Dense)	(None, 40)	1640
dense_7 (Dense)	(None, 2)	82
Total params: 12,202		

Структура рекуррентной модели с четырьмя полносвязными слоями

В такой реализации мы используем рекуррентный слой для предварительной обработки данных, а принятие решения в модели осуществляется несколькими полносвязными слоями перцептрона, следующими за рекуррентным слоем. В итоге мы получили модель с 12 202 параметрами.

Компилировать все нейронные модели мы будем с одинаковыми параметрами. Используем метод оптимизации *Adam*, а в качестве ошибки сети используем среднеквадратичное отклонение. Также добавляем дополнительную метрику *accuracy*.

```
model2.compile(optimizer='Adam',
               loss='mean_squared_error',
               metrics=['accuracy'])
```

С такими же параметрами мы компилировали модели нейронных сетей и ранее.

Следует обратить внимание еще на один момент. Рекуррентные модели чувствительны к последовательности подаваемого на вход сигнала. Поэтому при обучении нейронной сети в отличии от ранее рассмотренных моделей нельзя перемешивать исходные данные. Именно с этой целью при запуске обучения модели мы укажем значение *False* для параметра *shuffle*. Остальные параметры обучения модели остаются без изменений.

```
history2 = model2.fit(train_data, train_target,
                      epochs=500, batch_size=1000,
                      callbacks=[callback],
                      verbose=2,
                      validation_split=0.01,
                      shuffle=False)
```

В первой модели мы использовали рекуррентный слой для предварительной обработки данных перед использованием полносвязного перцептрона для принятия решения. Однако возможно использование рекуррентных нейронных слоев в чистом виде, без последующего использования полносвязных слоев. Именно такую реализацию я предлагаю рассмотреть в качестве второй модели. В ней мы просто все полносвязные слои заменяем одним рекуррентным слоем, а размер слоя указываем равным требуемому размеру выхода нейронной сети.

Следует обратить внимание, что на вход рекуррентного нейронного слоя требуется подать трехмерный тензор, в то время как на выходе из предыдущего рекуррентного слоя мы получили двухмерный. Поэтому перед передачей информации на вход следующего рекуррентного слоя нам нужно переформатировать данные. В данной реализации мы делаем последнее изменение равным двум, а размер измерения временных меток оставляем для расчета модели. Какого-либо искажения данных от такого переформатирования мы не ожидаем, так как мы группируем последовательные данные и, таким образом, просто укрупняем временной интервал. В то же время временной интервал между любыми двумя последующими элементами новой таймсерии остается постоянным.

```
# Модель LSTM-блок без полносвязных слоев
model4 = keras.Sequential([keras.Input(shape=inputs),
# Переформатируем тензор в 3-мерный.
# Указываем 2 измерения, т.к. 3-е измерение определяется размером пакета
                           keras.layers.Reshape((-1,4)),
# 2 последовательных LSTM-блока
# 1-й содержит 40 элементами
                           keras.layers.LSTM(40,
              kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5),
                           return_sequences=False),
# 2-й выдает результат вместо полносвязного слоя
                           keras.layers.Reshape((-1,2)),
                           keras.layers.LSTM(targerts)
                          ])
```

Теперь мы получили нейронную сеть, в которой первый рекуррентный слой осуществляет предварительную обработку данных, а второй рекуррентный слой генерирует результат работы нейронной сети. За счет отказа от использования перцептрона мы сократили количество

нейронных слоев в сети и, соответственно, общее количество параметров, которое в новой модели составило 7 240 параметров.

Layer (type)	Output Shape	Param #
reshape_2 (Reshape)	(None, 40, 4)	0
lstm_1 (LSTM)	(None, 40)	7200
reshape_3 (Reshape)	(None, 20, 2)	0
lstm_2 (LSTM)	(None, 2)	40
Total params: 7,240		

Структура рекуррентной нейронной сети без использования полносвязных слоёв

Компилирование и обучение модели мы осуществляем с теми же параметрами, как и все предыдущие модели.

```
model4.compile(optimizer='Adam',
               loss='mean_squared_error',
               metrics=['accuracy'])

history4 = model4.fit(train_data, train_target,
                      epochs=500, batch_size=1000,
                      callbacks=[callback],
                      verbose=2,
                      validation_split=0.01,
                      shuffle=False)
```

Во второй рекуррентной модели для создания тензора исходных данных второму LSTM-слою мы переформатировали тензор результатов предыдущего слоя. Библиотека Keras дает нам возможность и другого варианта. Мы можем в первом LSTM-слое указать параметр *return_sequences=True*, который переводит рекуррентный слой в режим работы с выводом результатов на каждой итерации. В результате такого действия, на выходе рекуррентного слоя мы сразу получаем трехмерный тензор формата [*batch*, *timesteps*, *feature*]. Это позволит нам отказаться от переформатирования данных перед вторым рекуррентным слоем.

```
# Модель LSTM блок без полносвязных слоев
model5 = keras.Sequential([keras.Input(shape=inputs),
# Переформатируем тензор в 3-мерный.
# Указываем 2 измерения, т.к. 3-е измерение определяется размером пакета
                           keras.layers.Reshape((-1,4)),
# 2 последовательных LSTM блока
# 1-й содержит 40 элементами и возвращает результата на каждом шаге
                           keras.layers.LSTM(40,
                kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5),
                           return_sequences=True),
# 2-й выдает результат вместо полносвязного слоя
                           keras.layers.LSTM(targerts)
                          ])
```

Layer (type)	Output Shape	Param #
reshape_4 (Reshape)	(None, 40, 4)	0
lstm_3 (LSTM)	(None, 40, 40)	7200
lstm_4 (LSTM)	(None, 2)	344
Total params: 7,544		

Структура рекуррентной нейронной сети без использования полносвязных слоёв

Как можно заметить, при таком построении модели изменилась размерность тензора на выходе первого рекуррентного слоя. Как следствие, немного выросло количество параметров во втором рекуррентном слое. Это дало общее увеличение параметров во всей модели до 7 544 параметров. Тем не менее, это меньше общего количества параметров первой рекуррентной модели с использованием перцептрона для принятия решения.

Дополним блок построения графиков новыми моделями.

```
# отрисовка результатов обучения моделей
plt.figure()
plt.plot(history1.history['loss'], label='Perceptron train')
plt.plot(history1.history['val_loss'], label='Perceptron validation')
plt.plot(history3.history['loss'], label='Conv2D train')
plt.plot(history3.history['val_loss'], label='Conv2D validation')
plt.plot(history2.history['loss'], label='LSTM train')
plt.plot(history2.history['val_loss'], label='LSTM validation')
plt.plot(history4.history['loss'], label='LSTM only train')
plt.plot(history4.history['val_loss'], label='LSTM only validation')
plt.plot(history5.history['loss'], label='LSTM sequences train')
plt.plot(history5.history['val_loss'], label='LSTM sequences validation')
plt.ylabel('$MSE$ $loss$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics')
plt.legend(loc='upper right', ncol=2)

plt.figure()
plt.plot(history1.history['accuracy'], label='Perceptron train')
plt.plot(history1.history['val_accuracy'], label='Perceptron validation')
plt.plot(history3.history['accuracy'], label='Conv2D train')
plt.plot(history3.history['val_accuracy'], label='Conv2D validation')
plt.plot(history2.history['accuracy'], label='LSTM train')
plt.plot(history2.history['val_accuracy'], label='LSTM validation')
plt.plot(history4.history['accuracy'], label='LSTM only train')
plt.plot(history4.history['val_accuracy'], label='LSTM only validation')
plt.plot(history5.history['accuracy'], label='LSTM sequences train')
plt.plot(history5.history['val_accuracy'], label='LSTM sequences validation')
plt.ylabel('$Accuracy$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics')
plt.legend(loc='lower right', ncol=2)
```

Кроме того, добавим новые модели в блок проверки работоспособности моделей на тестовой выборке и вывод результатов.

```
# Проверка результатов моделей на тестовой выборке
test_loss1, test_acc1 = model1.evaluate(test_data, test_target, verbose=2)
test_loss2, test_acc2 = model2.evaluate(test_data, test_target, verbose=2)
test_loss3, test_acc3 = model3.evaluate(test_data, test_target, verbose=2)
test_loss4, test_acc4 = model4.evaluate(test_data, test_target, verbose=2)
test_loss5, test_acc5 = model5.evaluate(test_data, test_target, verbose=2)

print('LSTM model')
print('Test accuracy:', test_acc2)
print('Test loss:', test_loss2)

print('LSTM only model')
print('Test accuracy:', test_acc4)
print('Test loss:', test_loss4)

print('LSTM sequences model')
print('Test accuracy:', test_acc5)
print('Test loss:', test_loss5)
```

В данном разделе мы подготовили скрипт на языке Python, в котором создается в общей сложности 5 моделей нейронных сетей:

- полносвязный перцептрон,
- сверточная модель,
- 3 модели рекуррентных нейронных сетей.

При выполнении скрипта мы проведем небольшое обучение всех пяти моделей на одном наборе исходных данных и сравним результаты работы обученных моделей на одном наборе тестовых данных. Это даст нам возможность сравнить работу различных архитектурных решений на реальных данных. Результаты тестов будут даны в следующей главе.

4.2.5 Сравнительное тестирование рекуррентных моделей

Вот мы с вами дошли и до тестирования работы рекуррентных моделей. Ранее мы уже проводили тестирование различных моделей полносвязного перцептрона и нескольких сверточных моделей. Можно заметить, что в обоих разделах, посвященных тестированию моделей, есть определенная последовательность действий. Так сказать, свой алгоритм проведения тестирования. В данном разделе мы тоже будем придерживаться этой последовательности.

Как и при тестировании предыдущих моделей, начнем мы с проверки корректности распределения градиента через наш рекуррентный слой, построенный средствами *MQL5*. Для этого мы построим скрипт *check_gradient_lstm.mq5*. Скрипт мы будем строить на базе ранее построенных аналогичных скриптов для проверки корректности работы предыдущих моделей. В принципе мы сделаем копию скрипта *check_gradient_conv.mq5* из раздела тестирования сверточных моделей и внесем в него изменения, соответствующие новой модели.

И первое, что мы изменим в скрипте, так это блок задания структуры модели для тестирования. Мы удалим из модели сверточный и подвыборочный слои. Вместо них в нашей модели появится один рекуррентный слой.

```
//--- рекуррентный слой
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      delete layers;
      return false;
     }
   descr.type = defNeuronLSTM;
   descr.count = BarsToLine;
   descr.window_out = 2;
   descr.activation = AF_NONE;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete layers;
      delete descr;
      return false;
     }
```

В остальном блок построения нейронной сети остается без изменения.

При тестировании других архитектурных решений нейронных слоев вышеприведенного изменения скрипта в части задания структуры нейронной сети было бы достаточно для проведения теста. Но рекуррентный блок *LSTM* имеет свои особенности. В первую очередь у него отсутствует в общепринятом смысле матрица весов — ее функционал возлагается на матрицы весов внутренних слоев. Организовать к ним доступ будет немного сложнее, но я не вижу в этом смысла. В качестве внутренних слоев мы используем уже проверенный класс полносвязного слоя, в корректной работе которого мы уверены. Следовательно, нам нет необходимости повторно проверять работу уже проверенного алгоритма распределения градиента ошибки до матрицы весов. В то же время у нас есть вопрос по корректности работы нового функционала по распределению градиента ошибки внутри LSTM блока. Думаю, для получения ответа на этот вопрос достаточно проверить передачу градиента ошибки до уровня исходных данных (входа нейронной сети). Поэтому блок проверки корректности градиента ошибки на уровне матрицы весов мы удаляем из скрипта.

Второй особенностью рекуррентного слоя является использование своих результатов в качестве исходных данных для новой итерации. Я согласен, что именно для этого все и затевалось. Мы хотели, чтобы нейронная сеть учитывала не только текущее состояние внешней среды, но и ее предыдущие состояния, которые мы передаем в качестве скрытого состояния на новую итерацию. Это дает положительный эффект для работы нейронной сети, но искажает данные для тестирования корректности распределения градиента ошибки. Дело в том, что весь наш алгоритм проверки корректности распределения градиента ошибки построен на принципе изменения только одного проверяемого параметра при прочих постоянных значениях состояния внешней среды. Но в случае с рекуррентным слоем даже при постоянстве всех параметров исходных данных мы можем получить другой результат за счет изменения скрытого состояния. Чтобы исключить это влияние, нам временно нужно добавить очистку буферов памяти и скрытого состояния в метод прямого прохода нашего класса рекуррентного *LSTM*-блока *CNeuronLSTM::FeedForward*. Я специально выделил эти строки в коде заливкой.

```
bool CNeuronLSTM::FeedForward(CNeuronBase *prevLayer)
  {
//--- Проверяем актуальность всех объектов
    ....
//--- Подготавливаем заготовки для новых буферов памяти и скрытого состояния
   CBufferDouble *memory = CreateBuffer(m_cMemorys);
   if(!memory)
      return false;
   CBufferDouble *hidden = CreateBuffer(m_cHiddenStates);
   if(!hidden)
     {
      delete memory;
      return false;
     }
//--- Только для проверки градиента
   memory.BufferInit(m_cOutputs.Total(), 0);
   hidden.BufferInit(m_cOutputs.Total(), 0);
//--- Далее следует код метода без изменений
```

Не забудьте удалить или закомментировать эти строки после проведения теста распространения градиента.

После внесения всех необходимых правок компилируем и запускаем выполнение теста с использованием технологии многопоточных вычислений *OpenCL* и без. Полученные результаты вполне удовлетворяют нашим требованиям и мы можем продолжить тестирование моделей дальше.

```
Use OpenCL true
Delta at input gradient between methods -1.94600e-11
Use OpenCL false
Delta at input gradient between methods -2.85605e-13
```

Тест корректности распределения градиента ошибки через LSTM блок

Мы получили подтверждения корректности работы построенного нами алгоритма распространения градиента ошибки через рекуррентный *LSTM*-блок и можем перейти к следующему этапу наших тестов. Но еще раз повторюсь, перед началом работ по проведению тестов нам необходимо убрать из кода метода прямого *CNeuronLSTM::FeedForward* выделенный выше код обнуления буферов памяти и скрытого состояния.

Скрипт для тестирования рекуррентных моделей

Для проведения тестового обучения рекуррентной модели создадим скрипт *lstm_test.mq5*. Данный скрипт создается полностью по шаблону скриптов аналогичного тестирования предыдущих моделей.

В начале скрипта объявляем внешние параметры для управления процессом создания и обучения модели нейронной сети. Практически все внешние параметры перекочевали из скрипта тестирования сверточных моделей без изменений.

```
//+------------------------------------------------------------------+
//| Внешние параметры для работы скрипта                             |
//+------------------------------------------------------------------+
// Имя файла с обучающей выборкой
input string   StudyFileName = "study_data.csv";
// Имя файла для записи динамики ошибки
input string   OutputFileName = "loss_study_lstm.csv";
// Количество исторических баров в одном паттерне
input int      BarsToLine     = 40;
// Количество нейронов входного слоя на 1 бар
input int      NeuronsToBar   = 4;
// Использовать OpenCL
input bool     UseOpenCL      = false;
// Размер пакета для обновления матрицы весов
input int      BatchSize      = 10000;
// Коэффициент обучения
input double   LearningRate   = 0.00003;
// Количество скрытых слоев
input int      HiddenLayers   =  3;
// Количество нейронов в одном скрытом слое
input int      HiddenLayer    =  40;
// Количество циклов обновления матрицы весов
input int      Epochs         =  1000;
```

В функции описания архитектуры модели *CreateLayersDesc* между слоем исходных данных и блоком скрытых слоев вставим один *LSTM*-блок. Размер буфера результатов данного рекуррентного блока будет равен количеству анализируемых нейронных слоев. Глубину анализируемой истории установим на пять итераций. Архитектурой *LSTM*-блока уже определены функции активации всех его составляющих, и сам блок не имеет верхнеуровневой функции активации. Соответственно, в описании архитектуры блока мы укажем отсутствие функции активации. Метод оптимизации параметров будем использовать Adam.

```
//--- рекуррентный слой
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   descr.type = defNeuronLSTM;
   descr.count = BarsToLine;
   descr.window_out = 5;
   descr.activation = AF_NONE;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
```

На этом процесс создания рекуррентной модели нейронной сети можно считать завершенным, так как остальной код функции остался без изменений.

На данном этапе выполнения скрипта у нас уже есть сформированная модель рекуррентной нейронной сети и загружена в память обучающая выборка. Можно сказать, что у нас все готово к обучению модели. И здесь у нас будет небольшое отступление от ранее используемого шаблона. Дело в том, что для обучения полносвязного перцептрона и сверточной нейронной сети мы использовали случайные паттерны из общей обучающей выборки. В то же время мы не раз уже упоминали, что рекуррентные нейронные сети требует строгого соблюдения хронологической последовательности подаваемых на вход исходных данных. Поэтому нам предстоит внести небольшие правки в функцию обучения модели *NetworkFit*.

Нам нужна строгая последовательность паттернов при обучении модели. Поэтому мы убираем генерацию случайного паттерна для каждой итерации. Вместо этого мы будем случайным образом определять начало очередного пакета данных из обучающей выборки.

```
bool NetworkFit(CNet &net, const CArrayObj &data, const CArrayObj &target,
                                                                VECTOR &loss_history)
  {
//--- обучение
   int patterns = data.Total();
//--- цикл по эпохам
   for(int epoch = 0; epoch < Epochs; epoch++)
     {
      ulong ticks = GetTickCount64();
      //--- обучаем батчами
      //--- выбор случайного паттерна
      int k = (int)((double)(MathRand() * MathRand()) / MathPow(32767.0, 2) *
                                                                (patterns - 10));
      k = fmax(k, 0);
```

Но и тут есть нюанс. При осуществлении прямого прохода рекуррентный блок учитывает результаты предыдущих итераций на глубину анализируемой истории. С целью сопоставимости данных мы должны заполнить буфер последовательными данными перед обучением модели.

Поэтому мы увеличиваем цикл каждого пакета перед обновлением параметров на число итераций для заполнения буфера глубины анализируемой истории. При этом до заполнения буфера мы не будем вызывать метод обратного прохода.

```
      for(int i = 0; (i < (BatchSize + 10) && (k + i) < patterns); i++)
        {
         //--- проверим на остановку обучения
         if(IsStopped())
           {
            Print("Network fitting stopped by user");
            return true;
           }
         if(!net.FeedForward(data.At(k + i)))
           {
            PrintFormat("Error in FeedForward: %d", GetLastError());
            return false;
           }
         if(i < 10)
            continue;
         if(!net.Backpropagation(target.At(k + i)))
           {
            PrintFormat("Error in Backpropagation: %d", GetLastError());
            return false;
           }
        }
      //--- перенастраиваем веса сети
      net.UpdateWeights(BatchSize);
      printf("Use OpenCL %s, epoch %d, time %.5f sec", (string)UseOpenCL, epoch,
                                                 (GetTickCount64() - ticks) / 1000.0);
      //--- сообщим о прошедшей эпохе
      TYPE loss = net.GetRecentAverageLoss();
      Comment(StringFormat("Epoch %d, error %.5f", epoch, loss));
      //--- запомним ошибку эпохи для сохранения в файл
      loss_history[epoch] = loss;
     }
   return true;
  }
```

В остальном код скрипта остался без изменений.

Надеюсь, с алгоритмом и принципом построения скрипта все понятно, и мы можем перейти к анализу полученных результатов.

Первое тестирование LSTM-модели

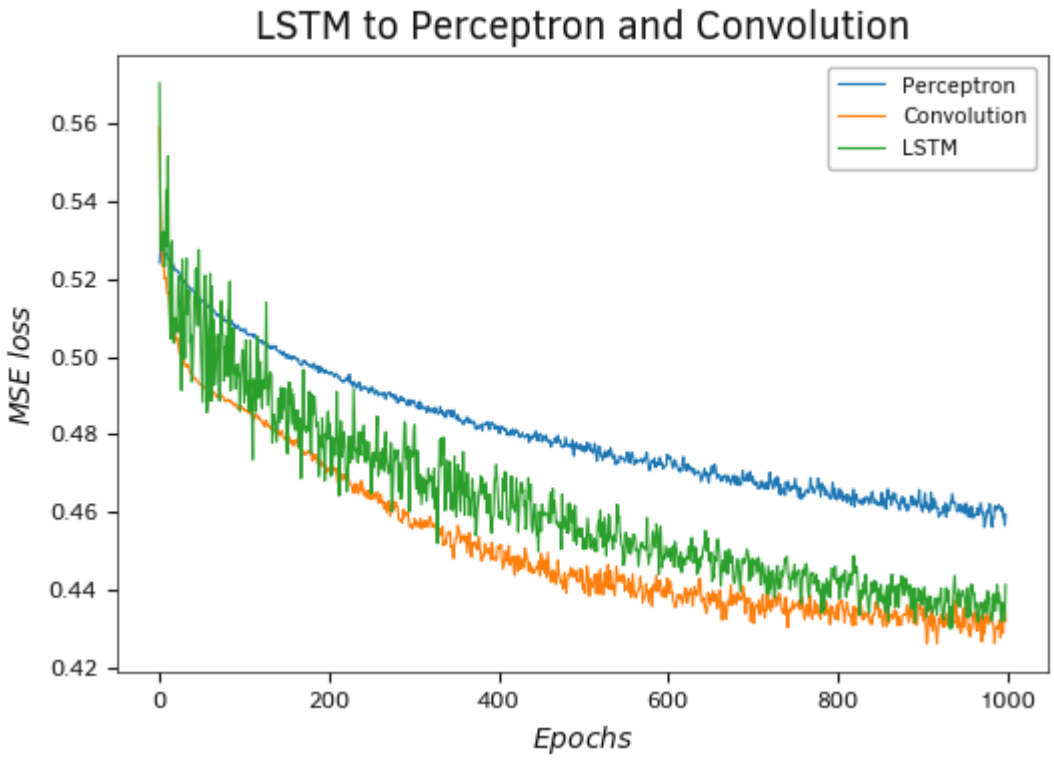
Для начала я создал модель по аналогии с тестируемыми сверточными моделями: один рекуррентный слой, три скрытых полносвязных слоя и один полносвязный слой для вывода результатов.

По результатам тестов можно заметить, что использование рекуррентного слоя наряду с использованием сверточного слоя для предварительной обработки данных значительно повышает показатели качества работы полносвязного перцептрона.

Напомню, что в модели перцептрона мы использовали три скрытых полносвязных слоя и один полносвязный слой результатов. В модели сверточной сети мы использовали один сверточный слой, один подвыборочный слой, три скрытых полносвязных слоя и один полносвязный слой результатов. В модели рекуррентной нейронной сети мы использовали один рекуррентный LSTM-блок, три скрытых полносвязных слоя и один полносвязный слой результатов.

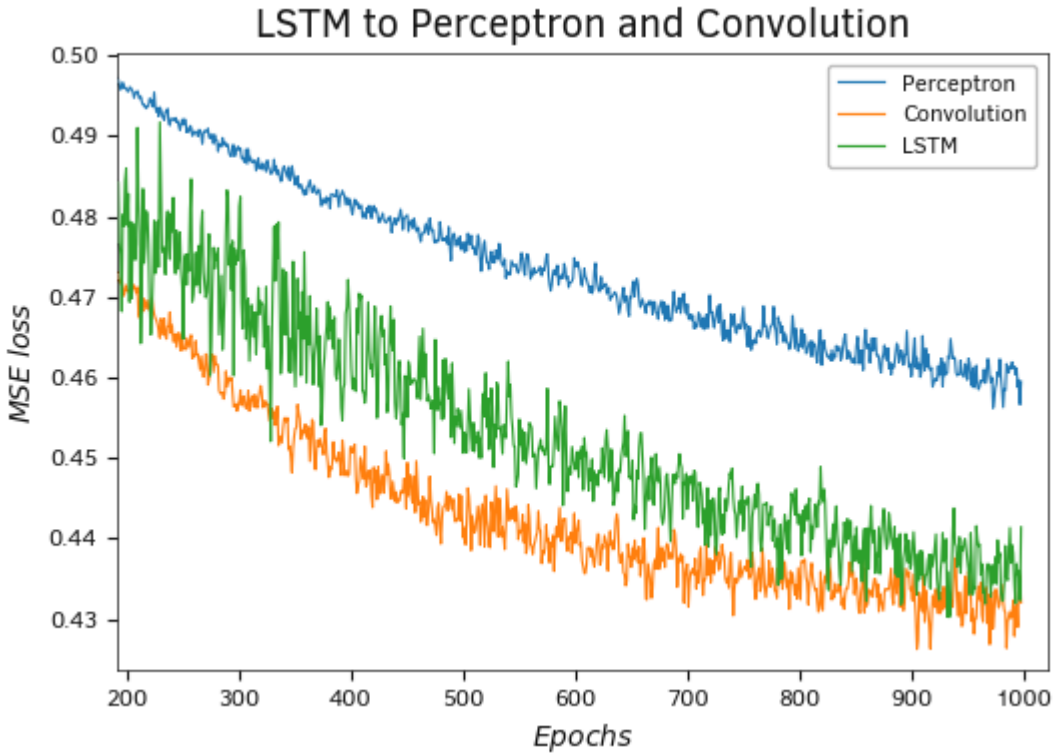
По существу, в сверточной и рекуррентной моделях мы добавили перед ранее протестированным перцептроном сверточный или рекуррентный блок для предварительной обработки данных. Тип используемого блока зависит от модели.

Как следствие, мы видим улучшение показателей работы нейронной за счет дополнительного слоя предварительной обработки исходных данных.



Тестирование модели рекуррентной нейронной сети

Сравнивая между собой сверточную и рекуррентную модели, можно заметить, что график ошибки рекуррентной модели обладает большими шумовыми колебаниями. Это может быть вызвано особенностями обучения моделей. Для обучения сверточной модели мы использовали паттерны, выбранные случайным образом из всей обучающей выборки. Такой вариант дает максимально репрезентативную выборку для каждого пакета накопления градиента ошибки перед обновлением весовых коэффициентов. В то же время для обучения рекуррентной модели мы брали паттерны в хронологическом порядке. А следовательно, обновление весовых коэффициентов и запись средней ошибки модели производилось на различных временных отрезках. Это не могло не отразиться на результате, ведь каждый локальный временной отрезок подвержен своим локальным трендам.



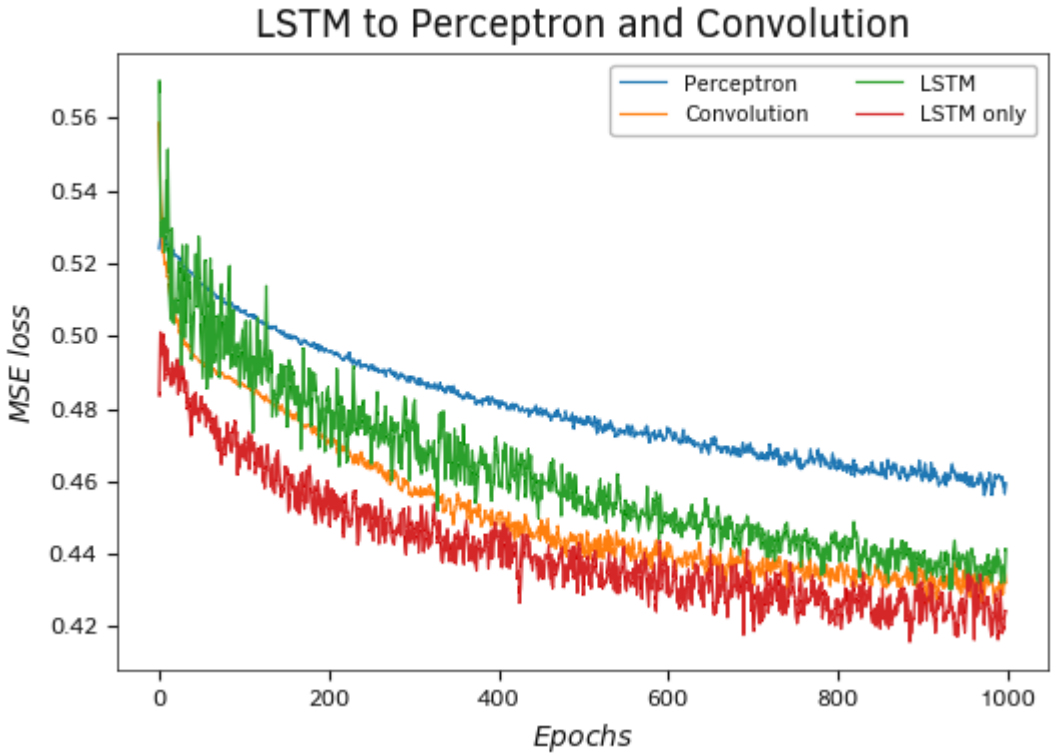
Тестирование модели рекуррентной нейронной сети

Несмотря на большой шум графика общая тенденция рекуррентной модели имеет большую тенденцию к снижению ошибки. Да, на протяжении всего процесса обучения значение ошибки модели немного лучше у сверточной модели. Но после 700 итераций обновления матрицы весов модели заметна тенденция к замедлению темпов ее снижения. Это может свидетельствовать о приближении к минимуму. При этом у рекуррентной модели нет такой тенденции. Рекуррентная модель обладает большим количеством параметров, и ей требуется больше времени на обучение. Потенциально она способна улучшить результаты при дальнейшем обучении.

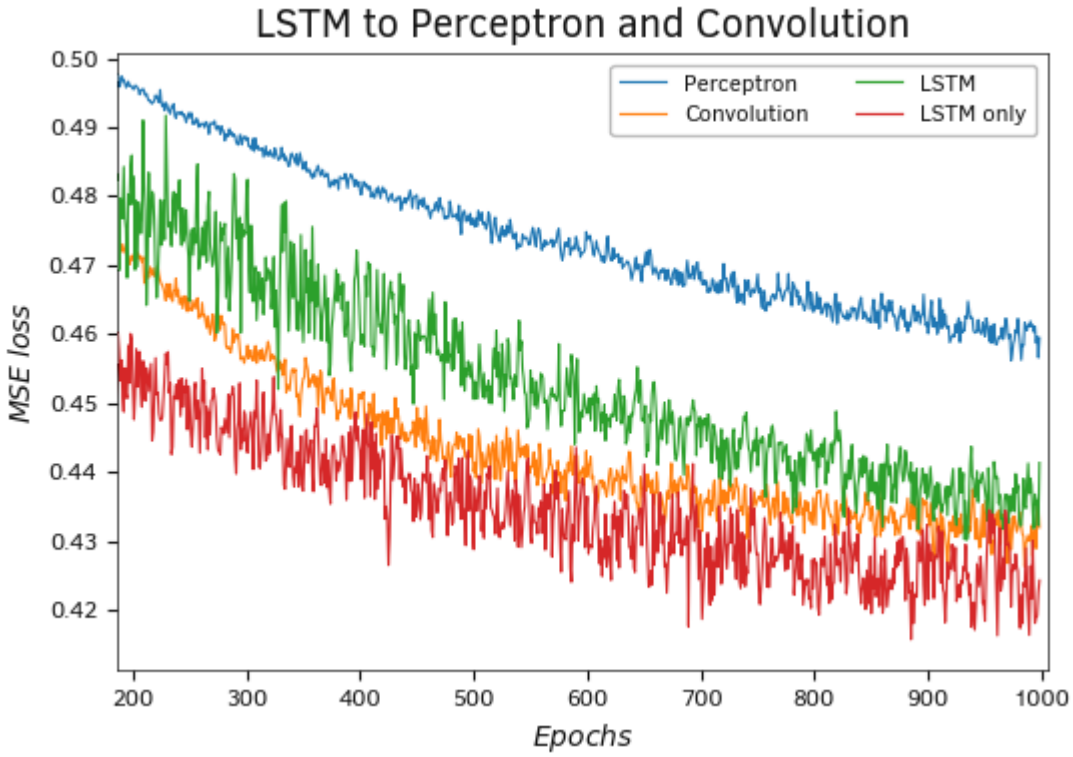
Второе тестирование LSTM-модели

В предыдущем тестировании рекуррентной модели мы использовали LSTM-слой для предварительной обработки исходных данных перед блоком полносвязных нейронных слоев. Но на практике существует возможность использовать рекуррентные слои без дополнительной обработки. Чтобы проверить влияние блока полносвязных слоев на качество работы рекуррентной нейронной сети мы провели второй эксперимент с тем же скриптом. Только теперь мы указали 0 в параметре количества скрытых слоев. Таким образом, мы хотим сравнить работу двух рекуррентных моделей и оценить необходимость использования блока полносвязных нейронных слоев для последующей обработки данных после рекуррентного слоя.

Полученные результаты тестирования демонстрируют весьма интересную тенденцию. В начале обучения рекуррентная модель без скрытых полносвязных слоев демонстрирует более резкое падение ошибки модели, которое превосходит все другие модели, приведенные на графике. При увеличении масштаба графика видно явное преимущество модели без блока срытых полносвязных слоев.



Тестирование модели рекуррентной нейронной сети



Тестирование модели рекуррентной нейронной сети

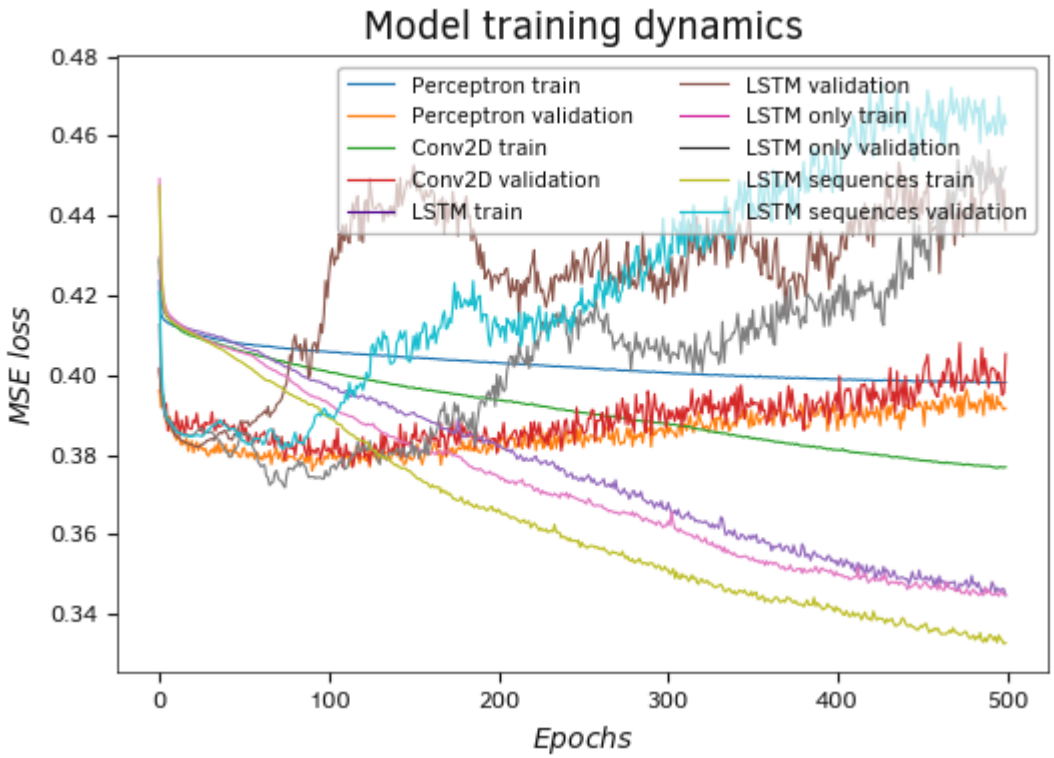
Результаты проведенных тестов показывают преимущество работы рекуррентных сетей над ранее рассмотренными моделями. При этом использование рекуррентных слоев дает результат даже без использования дополнительной обработки результатов полносвязными слоями.

Здесь надо сказать, что оценка моделей проводилась только для решения определенной задачи работы с временным рядами. При решении других задач возможно получение абсолютно противоположных результатов. Поэтому при решении ваших задач рекомендуется экспериментировать с различными архитектурными решениями нейронных сетей.

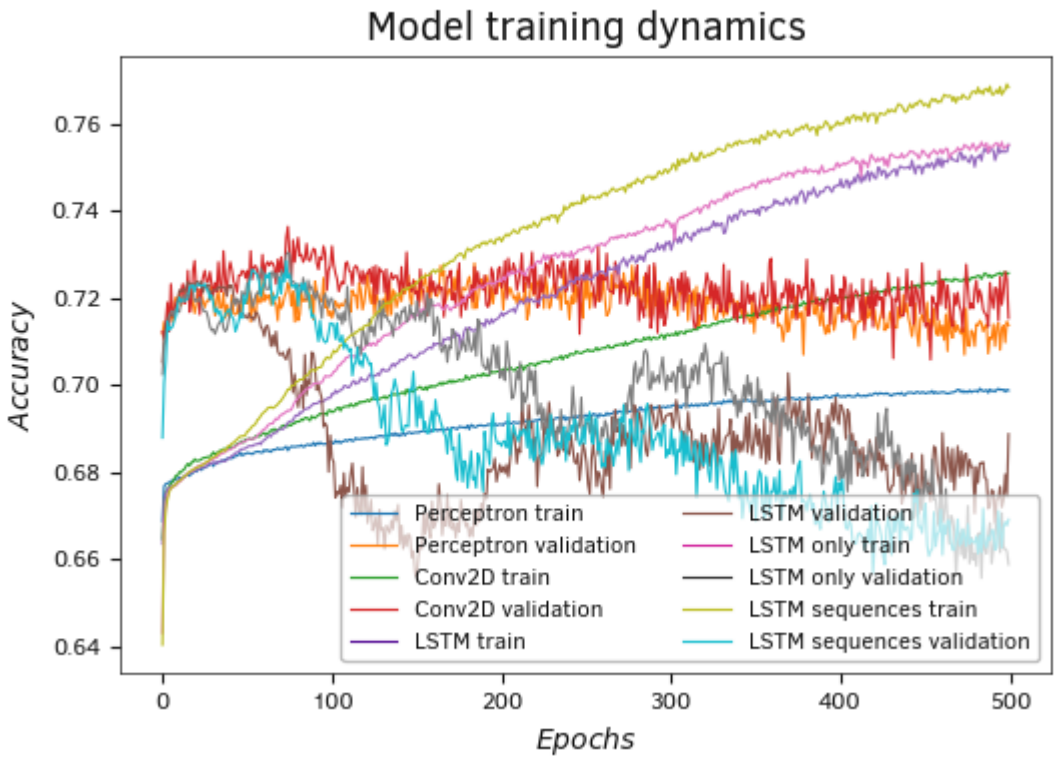
Результаты тестирования рекуррентных моделей в Python

Ранее мы рассмотрели реализацию скрипта с построением трех рекуррентных моделей на языке Python. Сейчас я предлагаю рассмотреть результаты тестового обучения построенных моделей.

Полученные результаты тестирования подтверждают сделанные нами ранее выводы на основании тестирования моделей, созданных средствами *MQL5*. Все три рекуррентные модели значительно превосходят другие модели по качеству работы нейронной сети. На графике изменения ошибки в процессе обучения нейронной сети мы видим, что рекуррентные модели уже после 50-ти эпох обучения демонстрируют ошибку меньше полносвязного перцептрона и сверточной модели. При дальнейшем обучении превосходство только растет. В то же время можно заметить и рост ошибки на валидационной выборке, что свидетельствует о склонности модели к переобучению.



Результаты тестового обучения моделей Python

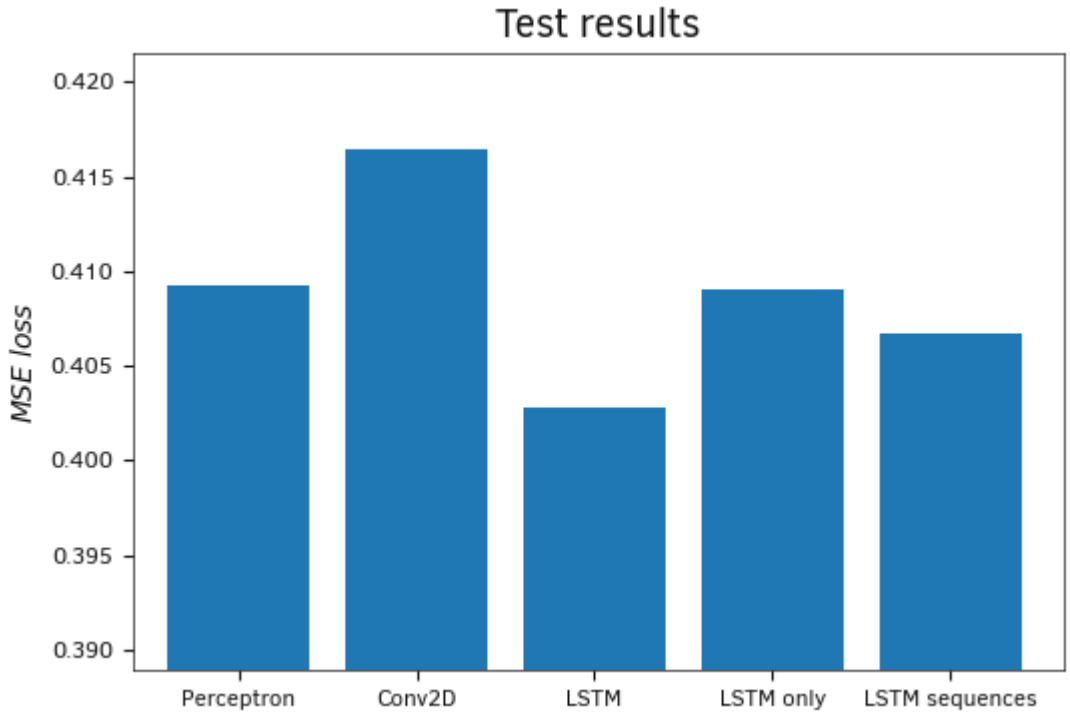


Результаты тестового обучения моделей Python

Сравнивая рекуррентные модели между собой, можно заметить, что рекуррентная модель в которой первый рекуррентный слой возвращает значения на каждом цикле больше склонна к переобучения. Она показывает меньшую из всех моделей ошибку на обучающей выборке и максимальную ошибку на валидационной выборке. В то же время пересечение графиков ошибки на тестовой и валидационной выборке для указанной модели наблюдается в районе 130 эпох при значении ошибки около 0,385. Пересечение графиков двух других моделей наблюдается с уровнем ошибки около 0,395.

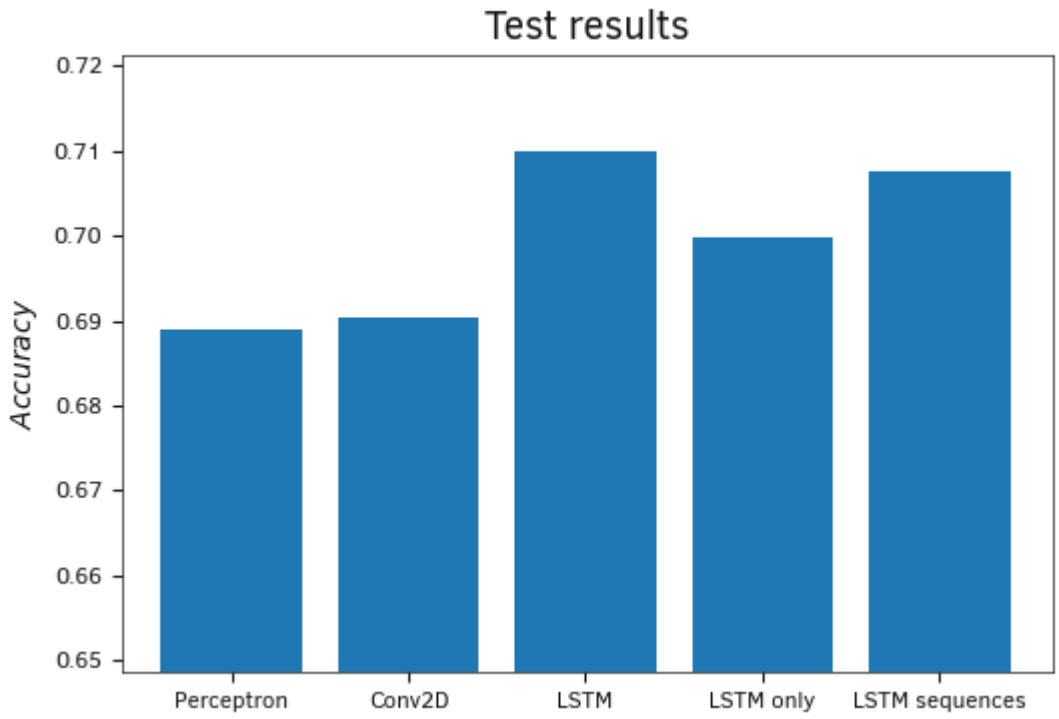
График динамики обучения по метрике Accuracy полностью подтверждают наши выводы, сделанные по графику ошибки.

На тестовой выборке все обученные модели показали довольно близкие результаты. Отклонение как по среднеквадратичной ошибке, так и по метрике accuracy минимально.



Тестирование обученных моделей Python на тестовой выборке

Если по значениям MSE картина довольно смешанная, то на графике метрики Accuracy наблюдается явное превосходство рекуррентных моделей.



Тестирование обученных моделей Python на тестовой выборке

По результатам проведенных тестов можно сказать, что при решении задач с временными рядами рекуррентные сети способны показать результаты лучше рассмотренных ранее архитектурных решений. При этом для решения подобных задач мы можем рассматривать различные архитектурные решения. Среди них могут быть нейронные сети состоящие только из рекуррентных слоев, а могут быть использованы и смешанные модели, состоящие из нейронных слоев различного типа.

Несмотря на то, что в результате нашего теста моделей на языке Python победила рекуррентная модель, содержащая только рекуррентные нейронные слои, я рекомендую при решении ваших практических задач всегда экспериментировать с различными моделями — часто наилучшие результаты дает самое необычное архитектурное решение.

5. Механизмы внимания

В предыдущих разделах книги мы уже изучили различные архитектуры организации нейронных сетей, в том числе сверточные сети, заимствованные из алгоритмов обработки изображений. Также мы познакомились с рекуррентными нейронными сетями, используемыми для работы с последовательностями, в которых важны как сами значения, так и их место в исходном наборе данных.

Полносвязные и сверточные нейронные сети имеют фиксированные размер входной последовательности. Рекуррентные нейронные сети позволяют немного расширить анализируемую последовательность за счет передачи скрытых состояний с предыдущих итераций. Но и их эффективность снижается с ростом последовательности.

Все рассмотренные модели затрачивают одинаковое количество ресурсов на анализ всей последовательности. Но попробуйте оценить свое поведение в какой-либо ситуации. К примеру, даже читая эту книгу вы бежите взглядом по буквам, словам строчкам, последовательно переворачиваете страницу за страницей. При этом вы концентрируете свое внимание на какой-то отдельной составляющей. Постепенно читая слова, написанные в книге, в своем сознании вы складываете мозаику логической цепочки, вложенной в написанные слова. И опять в вашем сознании всегда лишь некая часть общего содержимого книги.

Рассматривая фотографию своих родных людей, вы прежде всего концентрируете свое внимание на их портретах. Лишь потом можете перевести свой взгляд на фоновые элементы фотографии. При этом вы все равно концентрируете свое внимание на фотографии. А вся внешняя среда, окружающая вас, остается за пределами вашей мыслительной деятельности в этот момент.

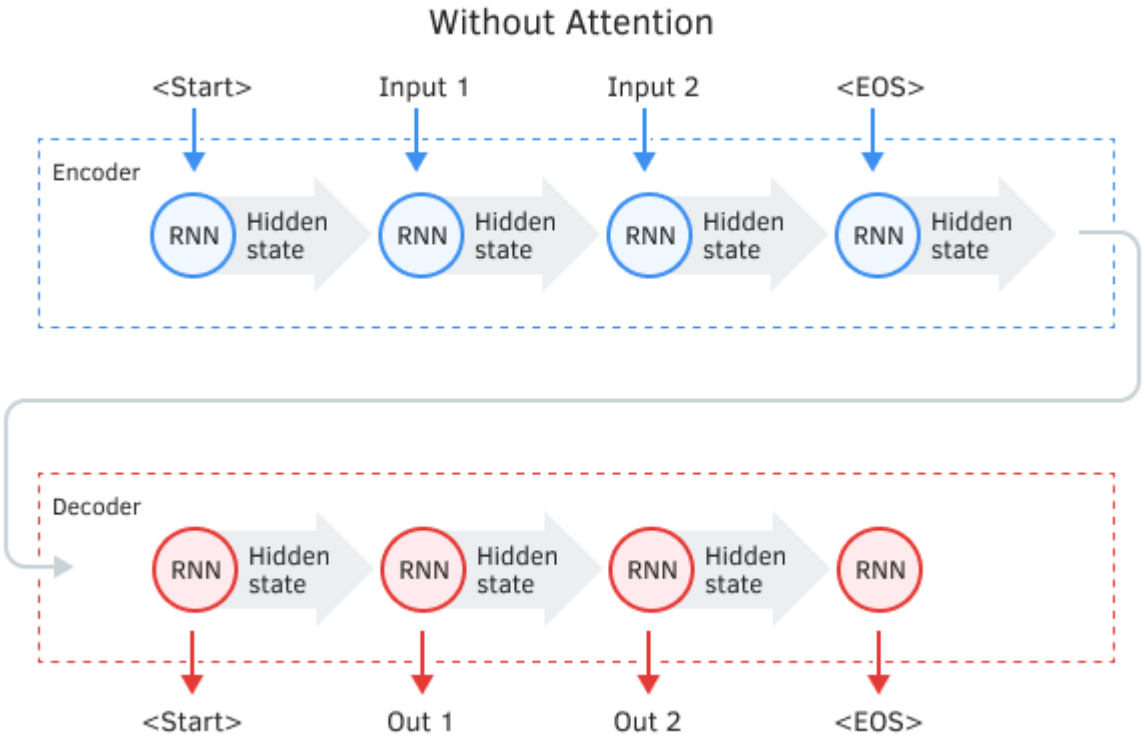
Я хочу вам показать, что человеческое сознание не оценивает всю окружающую среду целиком. Оно постоянно выхватывает из нее какие-то детали и переносит на них свое внимание. А вот рассмотренные нами модели нейронных сетей не обладают такой способностью.

Поэтому в 2014 году в области программного перевода было предложено использование первого механизма внимания, который был призван программным путем определять и выделять блоки исходного предложения (контекст), наиболее релевантные для целевого слова перевода. Такой интуитивно понятный людям подход, позволил значительно повысить качество перевода текстов нейронными сетями.

Анализируя свечной график движения инструмента, мы выделяем тренды и тенденции, определяем зоны проторговки. Т.е. из общей картины мы выделяем некоторые объекты, концентрируя свое внимание именно на них. Для нас интуитивно понятно, что объекты в разной степени влияют на будущее поведение цены. Для реализации именно такого подхода и был

предложен первый алгоритм, анализирующий и выделяющий зависимости между элементами входной и выходной последовательностей. Предложенный алгоритм называют обобщенным механизмом внимания. Изначально он был предложен для использования в моделях машинного перевода с использованием рекуррентных сетей для решения задач долгосрочной памяти в переводе длинных предложений. Такой подход значительно превысил результаты ранее рассмотренных рекуррентных нейронных сетей на основе *LSTM*-блоков.

Классическая модель машинного перевода с использованием рекуррентных сетей состоит из двух блоков, Encoder и Decoder. Первый кодирует входную последовательность на исходном языке в вектор контекста, а второй декодирует полученный контекст в последовательность слов на целевом языке. С увеличением длины входной последовательности влияние первых слов на итоговый контекст предложения снижается, и, как следствие, снижается качество перевода. Использование *LSTM*-блоков немного увеличивало возможности модели, но все равно они оставались ограниченными.



Encoder - Decoder без механизма внимания

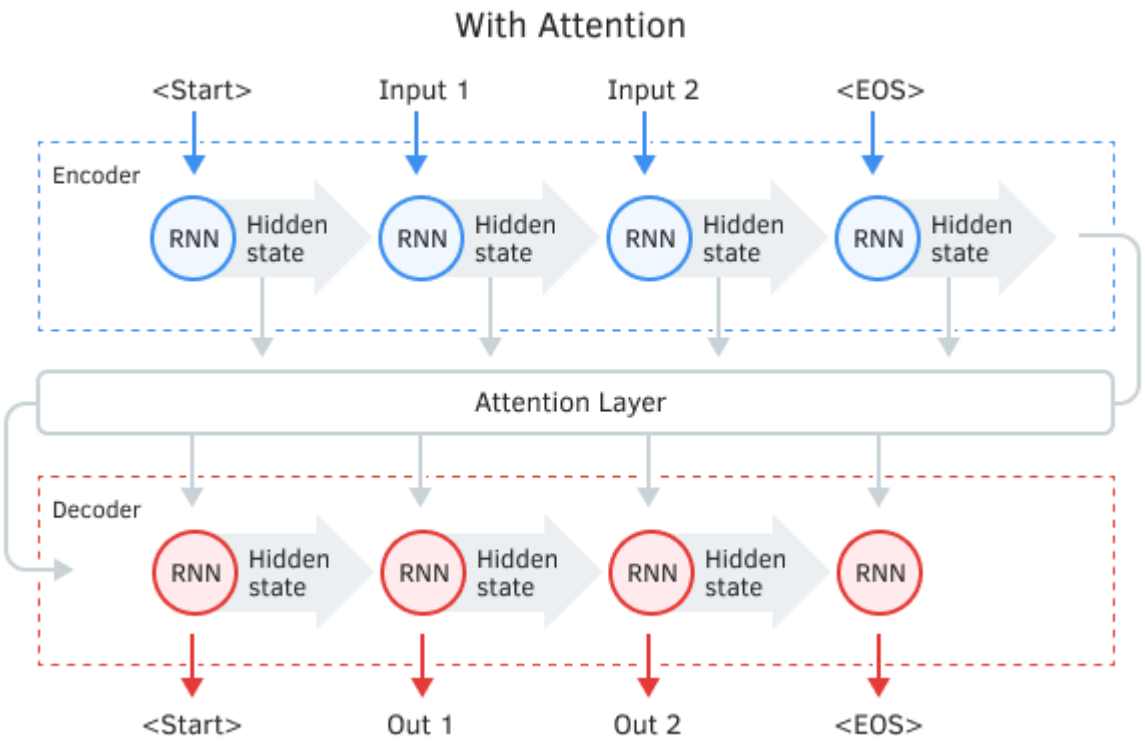
Авторы базового механизма внимания предложили использовать дополнительный слой, который бы аккумулировал скрытые состояния всех рекуррентных блоков входной последовательности и далее, в процессе декодирования последовательности, оценивал бы влияние каждого элемента входной последовательности на текущее слово выходной последовательности и предлагал декодеру наиболее релевантную часть контекста.

Алгоритм работы такого механизма включал следующие итерации:

1. Создание скрытых состояний *Encoder* и аккумулирование их в блоке внимания.
2. Оценка парных зависимостей между скрытыми состояниями каждого элемента *Encoder* и последнего срытого состояния *Decoder*.
3. Полученные оценки объединяются в единый вектор и нормализуются путем использования функции *Softmax*.

4. Вычисления вектора контекста путем умножения всех скрытых состояний *Encoder* на соответствующее им оценки выравнивания.
5. Декодирование вектора контекста и объединение полученного значения с предыдущим состоянием *Decoder*.

Все итерации повторяются до получения сигнала конца предложения.



Encoder - Decoder с механизмом внимания

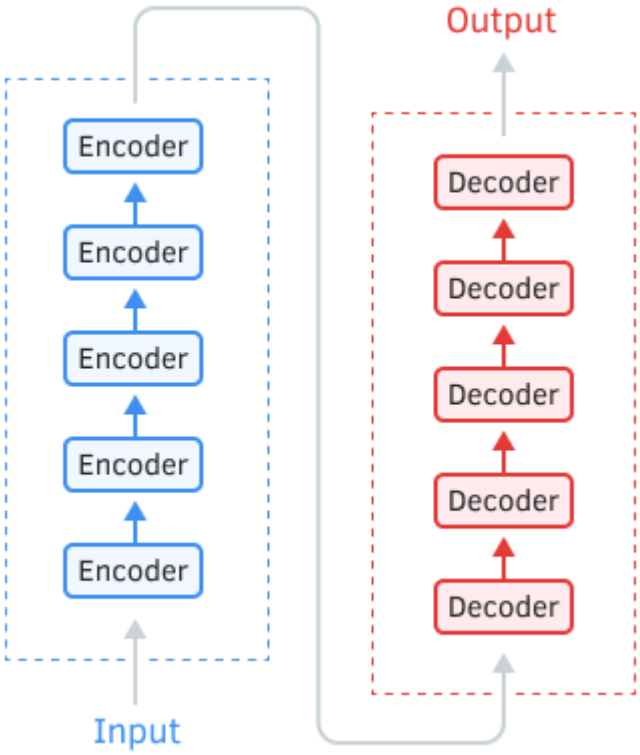
Предложенный механизм позволил решить проблему с ограничением длины входной последовательности и повысить качество машинного перевода с использованием рекуррентных нейронных сетей. Как следствие, он получил широкое распространение и различные вариации реализации. В частности, в августе 2015 года Минь-Тханг Луонг в статье Effective Approaches to Attention—based Neural Machine Translation предложил свою вариацию метода внимания. Основными отличиями нового подхода стали использование трех функций для вычисления степени зависимостей и точка использования механизма внимания в Decoder.

5.1 Self-Attention

Описанные выше модели используют рекуррентные блоки, обучение которых требует много затрат. В июне 2017 года в статье Attention Is All You Need была предложена новая архитектура нейронной сети — Трансформер — в которой отказались от использования рекуррентных блоков и предложили новый алгоритм внимания *Self-Attention*. В отличие от описанного выше алгоритм *Self-Attention* анализирует парные зависимости внутри одной последовательности. На тестах Трансформер показал лучшие результаты, и на сегодняшний день данная модель и ее производные используется во многих моделях, в том числе *GPT-2* и *GPT-3*. Рассмотрим алгоритм *Self-Attention* подробнее.

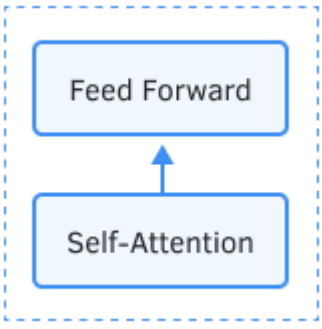
5.1.1 Описание архитектуры и принципов реализации

В основе архитектуры Трансформера лежат последовательные блоки *Encoder* и *Decoder* со схожей архитектурой. Каждый из блоков включает несколько одинаковых слоев с разными весовыми матрицами.



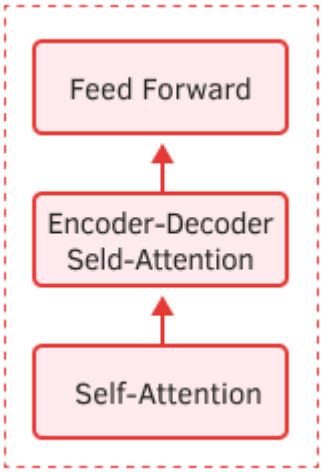
Архитектура Трансформера

Каждый слой Encoder содержит два внутренних слоя: *Self-Attention* и *Feed Forward*. Слой *Feed Forward* включает два полносвязных слоя нейронов с функцией активации *ReLU* на внутреннем слое. Каждый слой применяется для всех элементов последовательности с одинаковыми весовыми коэффициентами, что позволяет одновременно проводить независимые вычисления для всех элементов последовательности в параллельных потоках.



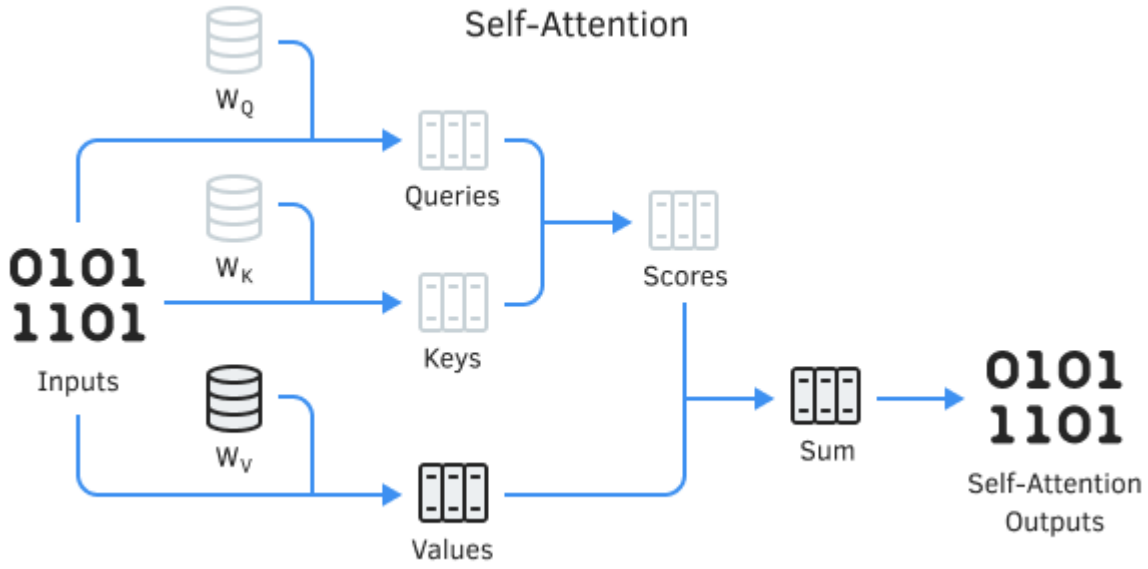
Encoder

Слой Decoder имеет схожую структуру, но добавляется еще один слой *Self-Attention*, анализирующий зависимости между входными и выходными последовательностями.



Decoder

Сам же механизм *Self-Attention* включается в себя несколько итерационных действий, применяемых для каждого элемента последовательности.



1. Вначале вычисляем векторы *Query* (запрос), *Key* (ключ) и *Value* (значение). Указанные векторы получаются путем умножения каждого элемента последовательности на соответствующую матрицу W_Q, W_K и W_V.
2. Далее определяем парные зависимости между элементами последовательности. Для этого перемножим вектор *Query* с векторами *Key* всех элементов последовательности. Данная итерация повторяется для вектора *Query* каждого элемента последовательности. В результате данной итерации получаем матрицу *Score* размером *N*N*, где *N* — размер последовательности.
3. Следующим этапом разделим полученные значение на квадратный корень из размерности вектора *Key* и нормализуем функцией *Softmax* в разрезе каждого *Query*. Таким образом, получаем коэффициенты попарной взаимозависимости между элементами последовательности.

4. Умножением каждого вектора *Value* на соответствующий коэффициент взаимозависимости получаем скорректированное значение элемента. Цель данной итерации — акцентировать внимание на релевантных элементах и снизить влияние не релевантных значений.
5. Далее суммируем все скорректированные вектора *Value* для каждого элемента. Результат данной операции и будет вектор выходных значений слоя *Self-Attention*.

Результаты итераций каждого слоя складываются с входной последовательностью и нормализуются.

$$\bar{a} = \frac{1}{h}\sum_{i=1}^{h} a_i$$

$$\hat{a} = \frac{a - \bar{a}}{\sqrt{\frac{1}{h}\sum_{i=1}^{h}(a - \bar{a})^2}}$$

Для нормализации данных мы сначала определяем среднее значение всей последовательности. Затем для каждого элемента рассчитываем частное от деления его отклонения от среднего на среднеквадратичное отклонение последовательности.

5.1.2 Построение Self-Attention средствами MQL5

Представленная архитектура *Self-Attention* после первого ознакомления может показаться довольно сложной для пониманию и тем более для реализации. Но не будем пессимистами. Давайте попытаемся разложить весь алгоритм на мелкие составляющие. Тогда, с реализацией каждого отдельного блока, мы соберем общую картину, и она уже не будет такой сложной для понимания. При этом вы сами удивитесь, как мы справимся с работой и выстроим рабочий механизм для нашей библиотеки.

Ну а сейчас приступим к работе. Для реализации нашего слоя *Self-Attention* мы создадим новый класс *CNeuronAttention*. Как всегда, наследоваться будем от нашего базового класса нейронного слоя *CNeuronBase*.

```
class CNeuronAttention    :  public CNeuronBase
  {
public:
                     CNeuronAttention(void);
                    ~CNeuronAttention(void);
   //---
   virtual bool      Init(const CLayerDescription *desc) override;
   virtual bool      SetOpenCL(CMyOpenCL *opencl) override;
   virtual bool      FeedForward(CNeuronBase *prevLayer) override;
   virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer) override;
   virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                   VECTOR &Beta, VECTOR &Lambda) override;
   //--- методы работы с файлами
   virtual bool      Save(const int file_handle) override;
   virtual bool      Load(const int file_handle) override;
   //--- метод идентификации объекта
   virtual int       Type(void) override  const { return(defNeuronAttention); }
  };
```

Рассмотрим первое действие алгоритма *Self-Attention* — вычисление векторов *Query*, *Key* и *Value*. На входе мы получаем тензор исходных данных, содержащий признаки по каждому бару анализируемой последовательности. Поочередно мы берем признаки одной свечи и, перемножая их с матрицей весов, получаем вектор. Затем берем признаки второй свечи и перемножаем их на ту же матрицу весов — получаем второй вектор, аналогичный первому. Не кажется ли вам это похожим на созданный ранее сверточный слой? Здесь длина вектора результатов равна количеству фильтров, используемых в сверточном слое. Следовательно, для организации указанного процесса объявим три вложенных сверточных слоя *CNeuronConv*. Будем использовать соответствующие имена слоев, чтобы код было легче читать.

```
class CNeuronAttention    :  public CNeuronBase
  {
protected:
   CNeuronConv       m_cQuerys;
   CNeuronConv       m_cKeys;
   CNeuronConv       m_cValues;
   .....
  };
```

По этому алгоритму на следующем этапе мы определяем матрицу *Score*, перемножая матрицы *Query* и *Key*. Для записи данных матрицы создадим буфер данных — объект класса *CBufferType*.

```
class CNeuronAttention    :  public CNeuronBase
  {
protected:
   .....
   CBufferType       m_cScores;
   .....
  };
```

После определения матрицы коэффициентов зависимостей *Score* нужно найти взвешенные значения. Для этого умножим векторы *Values* на соответствующие значения матрицы *Score*. После дополнительной обработки мы получим тензор, равный размеру исходных данных. О

причинах одинакового размера мы поговорим в процессе реализации. Сейчас же просто отметим для себя необходимость создания хранилища данных. Для сбора данных в хранилище нам нужно будет настроить новый процесс, поэтому нам нужен объект с легким доступом для записи данных. В последующем мы планируем передавать данные на вход внутреннего нейронного слоя. Значит, наиболее удобным для нас будет шаблон нейронного слоя исходных данных. Напомню, в качестве слоя исходных данных мы используем базовый нейронный слой с нулевым входным окном.

```
class CNeuronAttention    :  public CNeuronBase
  {
protected:
   .....
   CNeuronBase       m_cAttentionOut;
   .....
  };
```

Здесь следует обратить внимание на отличие выхода алгоритма *Self-Attention* от выхода всего нашего класса *CNeuronAttention*. Первый получаем после выполнения алгоритма *Self-Attention* корректировкой значений векторов *Value* и сохраняем в экземпляр объекта базового нейронного слоя *m_cAttentionOut*, а второй — после отработки в блоке *Feed Forward*, его сохраняем в буфере результатов нашего класса.

Поэтому дальше нам нужно организовать блок *Feed Forward*. Его мы создадим из двух последовательных сверточных слоев. Может показаться странным использование сверточного слоя, в то время как в описании архитектуры решения говорится о полносвязных слоях. Дело в том, что здесь ситуация аналогичная первому пункту алгоритма, когда мы определяли значение векторов *Query*, *Key* и *Value*. Рассматривая блок в рамках одного элемента последовательности, мы видим два полносвязных нейронных слоя. Но стоит посмотреть на всю таймсерию, как можно заметить применение одной матрицы весов поочередно к каждому элементу последовательности. При этом как последовательно идут исходные данные, в той же последовательности ложатся результаты. Разве это не напоминает работу сверточного слоя? Нам лишь надо взять сверточный слой и установить ширину окна исходных данных равной размеру вектора одного элемента последовательности. Шаг окна исходных данных устанавливаем равным ширине окна, а количество используемых фильтров определяется размером полносвязного слоя для одного элемента последовательности.

Таким образом, добавляем два сверточных слоя для организации блока *Feed Forward*.

```
class CNeuronAttention    :  public CNeuronBase
  {
protected:
   .....
   CNeuronConv       m_cFF1;
   CNeuronConv       m_cFF2;
   .....
  };
```

Мы определили объекты, необходимые нам для организации работы механизма *Self-Attention* в нашем классе. Для полноты картины добавим еще несколько переменных:

- *m_iWindow* — ширина окна исходных данных (размер вектора одного элемента последовательности);
- *m_iUnits* — количество элементов последовательности;

- *m_iKeysSize* — ширина размера вектора результатов для *Query* и *Key;*
- *m_dStd* — в процессе нормализации слоя мы будем делить значение на стандартное отклонение, значение будем сохранять для определения производной.

С учетом стандартного набора функций для переопределения структура класса будет иметь следующий вид.

```
class CNeuronAttention    :  public CNeuronBase
  {
protected:
   CNeuronConv       m_cQuerys;
   CNeuronConv       m_cKeys;
   CNeuronConv       m_cValues;
   CBufferType       m_cScores;
   int               m_cScoreGrad;
   int               m_cScoreTemp;
   CNeuronBase       m_cAttentionOut;
   CNeuronConv       m_cFF1;
   CNeuronConv       m_cFF2;
   //---
   int               m_iWindow;
   int               m_iUnits;
   int               m_iKeysSize;
   CBufferType       m_cStd;

public:
                     CNeuronAttention(void);
                    ~CNeuronAttention(void);
   //---
   virtual bool      Init(const CLayerDescription *desc) override;
   virtual bool      SetOpenCL(CMyOpenCL *opencl) override;
   virtual bool      FeedForward(CNeuronBase *prevLayer) override;
   virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer) override;
   virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                   VECTOR &Beta, VECTOR &Lambda) override;
   //--- методы работы с файлами
   virtual bool      Save(const int file_handle) override;
   virtual bool      Load(const int file_handle) override;
   //--- метод идентификации объекта
   virtual int       Type(void) override  const { return(defNeuronAttention); }
  };
```

В конструкторе класса мы лишь задаем начальные значения переменных.

Хочу обратить внимание, что в данном классе мы используем статические объекты, а не указатели на объекты, используемые нами ранее. Время жизни статических объектов, как и переменных, равно времени жизни содержащих их объекта. Это позволяет нам отказаться от необходимости создавать экземпляры объектов при инициализации класса и очищать память при завершении работы класса. Также нам нет необходимости каждый раз проверять действительность указателя на объект. Так мы можем сэкономить немного времени при выполнении каждого метода. Но это же лишает нас возможности подмены объектов копированием только указателя на объект — а это свойство мы активно используем в нашем

классе активации и в рекуррентных сетях (использование одних указателей на объекты при анализе всей глубины истории).

```
CNeuronAttention::CNeuronAttention(void) :   m_iWindow(1),
                                             m_iUnits(0),
                                             m_iKeysSize(1)
  {
   m_cStd.BufferInit(1, 2, 1);
  }
```

Использование статических объектов позволяет нам оставить пустым деструктор класса.

```
CNeuronAttention::~CNeuronAttention(void)
  {
  }
```

Метод инициализации класса

После создания конструктора и деструктора класса мы переходим к переопределению основных методов класса. Первым будем переопределять метод инициализации класса *CNeuronAttention::Init*. Основная задача данного метода — подготовка класса для выполнения своего функционала с заданными пользователем параметрами. Как и аналогичные методы других ранее рассмотренных классов, в параметрах метод получает экземпляр объекта *CLayerDescription,* в котором указаны параметры инициализируемого нейронного слоя. Поэтому, для исключения возможных ошибок в дальнейшей работе, мы организуем блок проверки исходных данных. В нем мы проверим наличие минимально необходимых параметров в полученных данных.

```
bool CNeuronAttention::Init(const CLayerDescription *desc)
  {
//--- проверяем исходные данные
   if(!desc || desc.type != Type() || desc.count <= 0 ||
       desc.window <= 0 || desc.window_out <= 0)
      return false;
```

После этого сохраним основные параметры в специально подготовленные переменные. Обратите внимание на соотношение параметров класса описания нейронного слоя и их функционального назначения:

- *CLayerDescription.window* — содержит размер окна исходных данных, вектор исходных данных одного элемента последовательности (в нашем случае описание одного бара);
- *CLayerDescription.count* — количество элементов в последовательности (количество анализируемых баров);
- *CLayerDescription.window_out* — размер вектора результатов для *Query* и *Key*.

```
   m_iWindow   = desc.window;
   m_iUnits    = desc.count;
   m_iKeysSize = desc.window_out;
```

Как и ранее, инициализацию объекта мы начинаем с вызова аналогичного метода инициализации родительского класса. Но здесь есть нюанс. Мы не можем просто передать полученное описание нейронного слоя. Создадим новый экземпляр объекта описания нейронного слоя и *CLayerDescription* и внесем в него скорректированные данные.

В поле *count* укажем общее количество на выходе из слоя, которое получим перемножением полей *count* и *window* данного объекта.

Обратите внимание, что для получения общего количества элементов на выходе нейронного слоя количество элементов в последовательности (количество анализируемых баров) мы умножаем на размер исходного окна (элементов, описывающих 1 бар), а не размер окна результатов. Дело в том, что размер окна результатов мы будем использовать только для тензоров *Query* и *Key*. Размер вектора результатов для тензоров *Value* и второго слоя блока *Feed Forward* будет равен размеру окна исходных данных. Это сделано для того, чтобы выровнять размерности исходных данных и результатов. Ведь алгоритмом предусмотрено сложение тензоров исходных данных с результатами работы блока *Self-Attention*, а потом еще и сложение тензоров результатов блока *Feed Forward* и *Self-Attention*. Таким образом, в результате сложения тензоров на выходе нашего нейронного слоя последовательность не может быть меньше исходных данных. Да и увеличивать ее нет никакого смысла. Следовательно, мы выравниваем размерности векторов.

Кроме изменения количества элементов, изменим и размер выходного окна, приравняв его к единице. Размер окна исходных данных укажем равным гулю. После этого вызовем метод инициализации родительского класса.

```
//--- вызываем метод инициализации родительского класса
   CLayerDescription *temp = new CLayerDescription();
   if(!temp)
      return false;
   temp.count = desc.count * desc.window;
   temp.window_out = 1;
   temp.window     = 0;
   temp.optimization = desc.optimization;
   temp.activation = desc.activation;
   temp.activation_params = desc.activation_params;
   temp.type = desc.type;
   if(!CNeuronBase::Init(temp))
     {
      delete temp;
      return false;
     }
```

Такая подмена параметров позволит запустить метод инициализации родительского класса в режиме нейронного слоя исходных данных. При этом не будут созданы дополнительные буфера для матрицы весов, а также соответствующие буферы метода оптимизации. Как и в случае LSTM-блока, данный нейронный слой не будет иметь отдельную матрицу весов. Все весовые коэффициенты будут храниться во внутренних нейронных слоях.

Аналогичную архитектуру укажем для внутреннего слоя сбора данных блока внимания *AttentionOut*. Мы лишь изменим тип нейронного слоя и явным образом отключим функцию активации.

```
//--- инициализируем AttentionOut
   temp.type = defNeuronBase;
   temp.activation=AF_NONE;
   if(!m_cAttentionOut.Init(temp))
     {
      delete temp;
      return false;
     }
```

Далее, чтобы инициализировать наши внутренние нейронные слои, нам необходимо создать для них описание. Заполним ранее созданный экземпляр класса *CLayerDescription* необходимыми данными. Почти все наши внутренние нейронные слои являются сверточными, в параметре *type* укажем *defNeuronConv*. Остальные параметры перенесем без изменений из полученного внешнего описания.

```
//--- создаем описание для внутренних нейронных слоев
   temp.type = defNeuronConv;
   temp.window = desc.window;
   temp.window_out = m_iKeysSize;
   temp.step = desc.window;
   temp.count = desc.count;
```

Далее мы приступаем к инициализации внутренних нейронных слоев. Первым мы инициализируем сверточный слой для определения векторов *Query* с помощью предварительно созданного описания. И конечно, не забываем проверить результат выполнения операций.

```
//--- инициализируем Querys
   if(!m_cQuerys.Init(temp) || !m_cQuerys.SetTransposedOutput(true))
     {
      delete temp;
      return false;
     }
```

Обратите внимание, что после инициализации сверточного нейронного слоя мы используем новый метод *CNeuronConv::SetTransposedOutput*. Причины его появления и функционал мы разберем чуть позже.

По аналогичному алгоритму инициализируем слой ключей *Keys*.

```
//--- инициализируем Keys
   if(!m_cKeys.Init(temp) || !m_cKeys.SetTransposedOutput(true))
     {
      delete temp;
      return false;
     }
```

Следом инициализируем слой *Values*. Мы также будем использовать приведенный выше алгоритм, но с небольшим дополнением. Как уже говорилось выше, при инициализации этого объекта используется окно результатов равное окну исходных данных. Поэтому внесем изменение в объект описания нейронного слоя и вызовем метод инициализации. Проверим результат выполнения операций.

```
//--- инициализируем Values
   temp.window_out = m_iWindow;
   if(!m_cValues.Init(temp) || !m_cValues.SetTransposedOutput(true))
     {
      delete temp;
      return false;
     }
```

Следующей мы инициализируем матрицу коэффициентов *Scores*. По алгоритму механизма *Self-Attention* это квадратная матрица со стороной равной количеству элементов в последовательности. Для нас это количество анализируемых баров.

В обсуждении данного алгоритма следует понимать разницу между *количеством элементов последовательности* и общим *количеством элементов на выходе нейронного слоя*. Если перевести это на анализ свечного графика изменения биржевого инструмента, то:

- Количество элементов последовательности — это количество анализируемых баров;
- Длина вектора одного элемента последовательности (окно входа / выхода) — количество элементов описывающих 1 бар;
- Общее количество элементов на входе / выходе нейронного слоя — произведение первых двух величин.

Вернемся к инициализации буфера матрицы коэффициентов. Для нее мы объявили буфер данных. Инициализируем его нулевыми значениями, задав размер буфера в виде квадратной матрицей.

```
//--- инициализируем Scores
   if(!m_cScores.BufferInit(temp.count, temp.count, 0))
     {
      delete temp;
      return false;
     }
```

Далее по алгоритму *Self-Attention* идет объект базового нейронного слоя для записи результатов внимания, который мы уже инициализировали выше.

Нам остается лишь инициализировать блок *Feed Forward*. Как уже было сказано, он будет состоять из двух сверточных нейронных слоев. Согласно предложенной авторами архитектуры, в первом нейронном слое тензор результатов в 4 раза превышает исходные данные. Кроме того, в первом нейронном слое авторы использовали функцию активации *ReLU*. Мы же заменим ее на *Swish*. Внесем указанные правки в описание нейронного слоя и проведем его инициализацию.

```
//--- инициализируем FF1
   temp.window_out *= 4;
   temp.activation = AF_SWISH;
   temp.activation_params[0] = 1;
   temp.activation_params[1] = 0;
   if(!m_cFF1.Init(temp) || !m_cFF1.SetTransposedOutput(true))
     {
      delete temp;
      return false;
     }
```

Для инициализации второго нейронного слоя блока *Feed Forward* нам наоборот нужно увеличить размер окна исходных данных и его шага. Размер окна результатов нужно вернуть в соответствие с размером тензора результатов блока внимания. Он же будет соответствовать размеру тензора предыдущего слоя.

Для второго нейронного слоя блока *Feed Forward* мы возьмем функцию активации, указанную пользователем при инициализации нашего класса.

После внесения изменений в объект описания нейронного слоя воспользуемся ранее рассмотренным алгоритмом и инициализируем последний внутренний нейронный слой.

```
//--- инициализируем FF2
   temp.window = temp.window_out;
   temp.window_out = temp.step;
   temp.step = temp.window;
   temp.activation = desc.activation;
   temp.activation_params = desc.activation_params;
   if(!m_cFF2.Init(temp) || !m_cFF2.SetTransposedOutput(true))
     {
      delete temp;
      return false;
     }
   delete temp;
```

После инициализации всех внутренних нейронных слоев удаляем временный объект описания нейронных слоев. Он нам больше не нужен.

Теперь применим небольшую хитрость. Согласно алгоритму, мы получаем результат работы в буфере результатов второго нейронного слоя блока *Feed Forward*. Для передачи данных на последующий нейронный слой нам нужно данные перенести в буфер результатов нашего класса. На операцию копирования данных нам потребуется дополнительные время и ресурсы на каждой итерации. Чтобы этого избежать, мы можем подменить указатели на объекты. Помните, что мы обсуждали об объектах и указателях на них?

Вначале мы удалим объект буфера результатов нашего класса, чтобы не оставлять неучтенные объекты в памяти. Затем в переменную для хранения указателя на объект буфера запишем указатель на аналогичный буфер второго нейронного слоя блока *Feed Forward*. Такую же операцию проведем и для буфера градиентов.

```
//--- для исключения копирования буферов осуществим их подмену
   if(m_cOutputs)
      delete m_cOutputs;
   m_cOutputs = m_cFF2.GetOutputs();
   if(m_cGradients)
      delete m_cGradients;
   m_cGradients = m_cFF2.GetGradients();
```

Благодаря такому несложному трюку нам удалось отказаться от постоянных копирований данных между буферами и сократить время на проведения операций внутри класса.

В заключение метода инициализации вызовем метод *SetOpenCL*, чтобы все наши внутренние объекты работали в едином контексте. Выйдем из метода с положительным результатом.

```
//--- передаем указатель на объект работы с OpenCL до всех внутренних объектов
   SetOpenCL(m_cOpenCL);
//---
   return true;
  }
```

Метод *SetOpenCL*, вызываемый в конце метода инициализации, призван распределить указатель на объект работы с контекстом *OpenCL* между всеми внутренними объектами. Это необходимо для обеспечения работы всех объектов в едином пространстве. Данный метод был создан виртуальным в базовом классе нейронного слоя. При необходимости он переопределяется в каждом новом классе.

Алгоритм метода довольно прост, и мы не раз уже рассматривали его во всех предыдущих классах. В параметрах метод получает указатель объект работы с контекстом *OpenCL* от внешней программы. Мы просто сначала вызываем метод родительского класса и передаем ему полученный указатель. Проверка корректности полученного указателя уже реализована в методе родительского класса, поэтому нет необходимости повторять ее здесь.

А потом передаем во все внутренние объекты указатель на контекст *OpenCL*, сохраненный в переменной нашего класса. Фокус в том, что метод родительского класса проверил полученный указатель и сохранил в переменную соответствующий указатель. Чтобы все объекты работали в одном контексте, мы распространяем уже обработанный указатель.

```
bool CNeuronAttention::SetOpenCL(CMyOpenCL *opencl)
  {
   CNeuronBase::SetOpenCL(opencl);
   m_cQuerys.SetOpenCL(m_cOpenCL);
   m_cKeys.SetOpenCL(m_cOpenCL);
   m_cValues.SetOpenCL(m_cOpenCL);
   m_cAttentionOut.SetOpenCL(m_cOpenCL);
   m_cFF1.SetOpenCL(m_cOpenCL);
   m_cFF2.SetOpenCL(m_cOpenCL);
   if(m_cOpenCL)
     {
      m_cScores.BufferCreate(m_cOpenCL);
      ulong size = sizeof(TYPE) * m_cScores.Total();
      m_cScoreGrad = m_cOpenCL.AddBuffer((uint)size, CL_MEM_READ_WRITE);
      m_cScoreTemp = m_cOpenCL.AddBuffer((uint)size, CL_MEM_READ_WRITE);
      m_cStd.BufferCreate(m_cOpenCL);
     }
   else
     {
      m_cScores.BufferFree();
      m_cStd.BufferFree();
     }
//---
   return(!!m_cOpenCL);
  }
```

Немного забегая вперед, хочу обратить внимание на создания буферов *m_cScoreGrad* и *m_cScoreTemp*. Они используются только в контексте *OpenCL* для временного хранения данных, поэтому мы не создавали под них зеркальные объекты в основной памяти. Также мы не будем их использовать для обмена данными между основной программой и контекстом *OpenCL*. В таком случае мы создаем буферы в контексте *OpenCL*, а на стороне основной программы используем только указатели работы с ними. При отключении технологии многопоточных вычислений мы сразу удаляем указанные буферы.

После завершения работы с методом инициализации класса мы можем перейти к переопределению функциональных методов нашего класса.

5.1.2.1 Метод прямого прохода Self-Attention

Мы с вами уже создали структуру класса организации для организации механизма внимания и даже создали метод инициализации объекта. В этом разделе мы будем организовывать процесс прямого прохода.

Как вы знаете, в базовом классе нейронной сети мы создали виртуальный метод *CNeuronBase::FeedForward*, который отвечает за организацию прямого прохода. В каждом новом классе мы переопределяем данный метод, чтобы организовать процесс прямого прохода в соответствии с алгоритмом работы реализуемого архитектурного решения. Тем самым мы как бы персонализируем метод для каждого класса. В то же время внешней программе не надо ничего знать об организации процесса в классе. Ей даже не нужно знать тип нейронного слоя. Она просто обращается к методу *FeedForward* очередного объекта и передает ему указатель на предыдущий слой нейронной сети. Таким образом, мы переложили функционал диспетчеризации и проверки требуемого типа объектов со своей программы на систему.

Вернемся к нашему методу *CNeuronAttention::FeedForward*. В параметрах, как и метод родительского класса, он получает указатель на объект предыдущего слоя. Это соответствует принципам наследования и переопределения методов. Так как мы получаем указатель на объект, то и в начале метода мы обычно организуем блок проверки действительности полученного указателя. Однако в данном случае мы его опустим. Дело в том, что использование статических внутренних объектов позволяет нам отказаться от проверки их указателей. Что касается указателя на предыдущий нейронный слой, то мы его будем использовать для прямого прохода внутренних сверточных нейронных слоев *m_cQuerys*, *m_cKeys* и *m_cValues*. В них уже реализованы аналогичные контроли — нам нет необходимости дублировать их.

В соответствии с алгоритмом работы *Self-Attention* нам необходимо определить векторы *Query*, *Key* и *Value* для каждого элемента последовательности. Как вы помните, именно для выполнения этого функционала мы создавали три первых сверточных слоя. Поэтому для решения этой задачи нам достаточно вызвать методы прямого прохода *FeedForward* для названных внутренних слоев. При каждом вызове в параметрах передадим указатель на предыдущий нейронный слой, полученный в параметрах нашего метода прямого прохода *CNeuronAttention::FeedForward*.

```
if(!m_cQuerys.FeedForward(prevLayer))
   return false;
if(!m_cKeys.FeedForward(prevLayer))
   return false;
if(!m_cValues.FeedForward(prevLayer))
   return false;
```

Далее по алгоритму *Self-Attention* нам надо определить коэффициенты зависимостей и заполнить матрицу *Score*. Тут наступает момент вспомнить о нашей парадигме создания классов, способных работать как на *CPU*, так и с использованием мощностей *GPU.* Каждый раз, выстраивая новый процесс, мы создаем разветвление алгоритма в зависимости от используемого вычислительного устройства. Этот метод не будет исключением, и мы продолжим работу в начатом ключе. Именно сейчас мы и создадим аналогичное разветвление процесса. Как всегда, сейчас мы рассмотрим создание процесса средствами *MQL5*. К ветке технологии *OpenCL* вернемся немного позже.

Для удобства работы мы скопируем матрицы результатов сверточных слоев *m_cQuerys* и *m_cKeys*.

```
//--- разветвление алгоритма по вычислительному устройству
   MATRIX out;
   if(!m_cOpenCL)
     {
      MATRIX querys = m_cQuerys.GetOutputs().m_mMatrix;
      MATRIX keys = m_cKeys.GetOutputs().m_mMatrix;
```

После выполнения подготовительной работы нам предстоит «закатить рукава» и выстроить новый процесс. Напомню, что алгоритмом метода *Self-Attention* предусмотрена построчная нормализация матрицы зависимостей с помощью функции *Softmax*.

$$SoftMax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Основная особенность такой нормализации заключается в получении ряда положительных значений, которые в сумме дают 1. Таким образом, перемножив нормализованные коэффициенты зависимостей на значения векторов *Value* соответствующих элементов последовательности и затем сложив полученные вектора в рамках одного запроса *Query,* мы ожидаем получить новые векторы в том же диапазоне значений.

Посмотрим на реализацию данного процесса. Вначале мы организуем процесс вычисления коэффициентов зависимости в матрицу *Score*. Согласно алгоритму *Self-Attention*, каждый элемент матрицы представляет произведение векторов *Query* и *Key*. При этом строка матрицы указывает на положение вектора в матрице *Querys*, а столбец — в матрице *Keys*.

Здесь следует внимательно отнестись к выбору перемножаемых элементов. Давайте вспомним, как мы организовывали вывод результатов в буфер сверточного слоя. Для возможности работы подвыборочного слоя в разрезе фильтров, мы организовали последовательный вывод фильтров. Сначала в первую строку матрицы буфер результатов мы выводим все элементы результата работы одного фильтра. Затем в следующую строку записываем элементы следующего фильтра и т.д. Такая организация буфера удобна для прозрачной работы подвыборочного слоя в рамках фильтров. В данном случае, в рамках вектора одного элемента последовательности нам нужно использовать по одному значению из каждого фильтра. Иными словами, нам нужна транспонированная матрица. Выстроить буфер таким образом, чтобы сначала были первые элементы всех фильтров, потом вторые элементы всех фильтров и т.д. Но реорганизация данных буфера в данном участке программы потребует дополнительных ресурсов на каждом прямом проходе. Куда проще было бы организовать удобную нам запись непосредственно в сверточном слое. Но это нарушит работу как подвыборочного слоя, так и последующих сверточных слоев при выстраивании сверточных моделей. Поэтому было принято решение добавить в работу сверточного слоя флаг выстраивания значений в буфере результатов. Возможно, вы уже догадались об этом, когда при описании метода инициализации я говорил о новом методе сверточного слоя *SetTransposedOutput*. Тогда я обещал вернуться к описанию функционала этого метода. Подобное решение помогло нам оставить прозрачной структуру метода прямого прохода и избежать дополнительных затрат времени и ресурсов на реорганизацию данных. Но давайте закончим работу с методом прямого прохода, а потом вернемся к изменениям в сверточном слое.

С учетом транспонирования результатов работы сверточного слоя для получения значений матрицы коэффициентов зависимостей нам необходимо умножить матрицу *Querys* на транспонированную матрицу *Keys*. Немного странно звучит, транспонировать работу метода сверточного слоя и потом транспонировать матрицу *Keys*. Но результатом транспонирования работы сверточного слоя мы воспользуемся еще не один раз. Конечно, с помощью введенного

флага мы могли транспонировать работу сверточного слоя *m_cQuerys*, а слой *m_cKeys* оставить без изменения. Но в таком случае не исключена путаница с размерностью матриц. Это усложнит читаемость и понимание кода. Поэтому было принято решение о единстве размерностей используемых матриц.

Сразу скажу, что параллельно с вычислением произведения векторов мы будем готовить данные для нормализации, согласно приведенной выше формуле *Softmax*. Именно для этого мы сразу разделим полученную матрицу на квадратный корень из размера вектора *Key* и возьмем экспоненту от полученного значения.

Затем мы возьмем построчную сумму значений матрицы и на полученный вектор разделим значения матрицы *Scores.* Матричные операции *MQL5* не позволяют разделить матрицу на вектор. Поэтому мы организуем цикл, в теле которого поочередно разделим каждую строку на сумму ее значений.

```
//--- определяем Scores
MATRIX scores = MathExp(querys.MatMul(keys.Transpose()) / sqrt(m_iKeysSize));
//--- нормализуем Scores
VECTOR summs = scores.Sum(1);
for(int r = 0; r < m_iUnits; r++)
   if(!scores.Row(scores.Row(r) / summs[r], r))
      return false;
m_cScores.m_mMatrix = scores;
```

После того, как мы нормализовали данные матрицы, содержащей коэффициенты зависимости элементов последовательности, перенесем полученные значения в наш буфер данных *m_cScores*.

На данном этапе мы получили посчитанные и нормализованные коэффициенты зависимостей между всеми элементами последовательности. Теперь, согласно алгоритму метода Self-Attention, нам необходимо посчитать взвешенную сумму значений векторов *Values* в разрезе каждого запроса *Query*. Для этого нам достаточно умножить матрицу коэффициентов зависимостей на матрицу результатов сверточного слоя *m_cValues*. Снова именно благодаря транспонированию работы сверточного слоя мы не транспонируем матрицу результатов слоя *m_cValues*.

```
//--- выход блока внимания
MATRIX values = m_cValues.GetOutputs().m_mMatrix;
out = scores.MatMul(values);
```

Произведение матриц даст нам результат механизма *Self-Attention*. Но мы пойдем немного дальше и выстроим полностью блок Encoder трансформера. Согласно его алгоритму результаты *Self-Attention* складываются с буфером исходных данных. Полученные значения нормализуются в рамках нейронного слоя. Для нормализации данных используются следующие формулы.

$$\bar{a} = \frac{1}{h}\sum_{i=1}^{h} a_i$$

$$\hat{a} = \frac{a - \bar{a}}{\sqrt{\frac{1}{h}\sum_{i=1}^{h}(a - \bar{a})^2}}$$

Для осуществления этой операции мы сначала приведем формат матрицы результатов блока *Self-Attention* в соответствие с форматом матрицы исходных данных и сложим две матрицы. Результат нормализуем в специально выделенном методе *NormlizeBuffer*.

```
      //--- суммируем с исходными данными и нормализуем
      if(!out.Reshape(prevLayer.Rows(), prevLayer.Cols()))
         return false;
      m_cAttentionOut.GetOutputs().m_mMatrix = out +
                                          prevLayer.GetOutputs().m_mMatrix;
      if(!NormlizeBuffer(m_cAttentionOut.GetOutputs(), GetPointer(m_cStd), 0))
         return false;
     }
```

На этом завершен первый блок операций, и вместе с ним мы завершаем блок разделения алгоритма по устройству выполнения математических операций. Для блока операций с использованием технологии OpenCL мы временно установим возврат ошибочного значения и вернемся к нему позже.

```
   else // Блок OpenCL
     {
      return false;
     }
```

Продолжим работу с алгоритмом энкодера и перейдем ко второму блоку операций. Здесь необходимо провести сигнал каждого элемента последовательности через два полносвязных слоя. Как вы помните, эту работу мы решили организовать через два сверточных слоя. На первый взгляд, в этом нет ничего сложного — мы просто последовательно вызываем методы прямого прохода для каждого сверточного слоя.

```
//--- вызываем методы прямого прохода слоев блока Feed Forward
   if(!m_cFF1.FeedForward(GetPointer(m_cAttentionOut)))
      return false;
   if(!m_cFF2.FeedForward(GetPointer(m_cFF1)))
      return false;
```

Здесь корректная работа возможна только благодаря транспонированию буфера результатов сверточных нейронных слоев. Только такой подход позволяет выравнять работу по каждому отдельному элементу последовательности.

После осуществления прямого прохода через два сверточных слоя, как и после определения результатов внимания, необходимо сложить полученные результаты с данными, поданными на вход первого сверточного слоя, и нормализовать полученные суммы. Мы уже рассматривали такую задачу выше. Здесь мы будем использовать тот же алгоритм, изменятся только буферы данных.

```
//--- суммируем с выходом внимания и нормализуем
   if(!m_cOutputs.SumArray(m_cAttentionOut.GetOutputs()))
      return false;
//--- нормализуем
   if(!NormlizeBuffer(m_cOutputs, GetPointer(m_cStd), 1))
      return false;
//---
   return true;
  }
```

И тут надо обратить внимание, что благодаря организованной в методе инициализации подмены буферов результаты работы второго сверточного слоя мы получаем из буфера результатов текущего слоя. В этот же буфер мы и сохраним результаты нормализации данных.

После завершения операций выходим из метода прямого прохода с положительным результатом.

А теперь посмотрим на изменения, внесенные в класс сверточного слоя. Прежде всего мы добавим переменную для хранения флага структуры вывода данных *m_bTransposedOutput*. Это будет логический флаг, указывающий на необходимость транспонировать матрицу результатов для вывода в буфер. По умолчанию установим значение *false*, что подразумевает работу в нормальном режиме.

```
class CNeuronConv    :  public CNeuronProof
  {
protected:
   bool              m_bTransposedOutput;

public:
   bool              SetTransposedOutput(const bool value);
   ....
  }
```

Для управления значением флага создадим метод *SetTransposedOutput*. Функционал метода довольно прост. Мы просто изменяем размеры матриц результатов и градиентов ошибки.

```
bool CNeuronConv::SetTransposedOutput(const bool value)
  {
   m_bTransposedOutput = value;
   if(value)
     {
      if(!m_cOutputs.BufferInit(m_iNeurons, m_iWindowOut, 0))
         return false;
      if(!m_cGradients.BufferInit(m_iNeurons, m_iWindowOut, 0))
         return false;
     }
   else
     {
      if(!m_cOutputs.BufferInit(m_iWindowOut, m_iNeurons, 0))
         return false;
      if(!m_cGradients.BufferInit(m_iWindowOut, m_iNeurons, 0))
         return false;
     }
//---
   return true;
  }
```

Но как вы понимаете, наличие флага и даже метода, который его изменяет, никак не повлияет на результаты вывода данных в буфер. Для этого мы должны внести некоторые правки в метод прямого прохода. Мы абсолютно не меняем алгоритм и логику расчета — наши изменения коснутся лишь перестановки матриц при умножении исходных данных на матрицу весов в зависимости от состояния флага *m_bTransposedOutput*.

```
bool CNeuronConv::FeedForward(CNeuronBase *prevLayer)
  {
//--- блок контролей
    ....
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
     ....
      //--- Вычисление взвешенной суммы элементов входного окна
      if(m_bTransposedOutput)
         m = m.MatMul(m_cWeights.m_mMatrix.Transpose());
      else
         m = m_cWeights.m_mMatrix.MatMul(m.Transpose());
      m_cOutputs.m_mMatrix = m;
     }
   else  // Блок OpenCL
     {
     ....
     }
//---
   if(!m_cActivation.Activation(m_cOutputs))
      return false;
//---
   return true;
  }
```

После внесения изменений в метод прямого прохода мы просто обязаны сделать аналогичные правки в методах обратного прохода, ведь градиент ошибки должен прийти именно в место возникновения ошибки. В противном случае результаты обучения нейронной сети будут непредсказуемыми. Сначала мы внесем правки в метод распределения градиента в скрытом слое *CNeuronConv::CalcHiddenGradient*.

```
bool CNeuronConv::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
//--- блок контролей
    ....
//--- корректировка градиентов ошибки на производную функции активации
    ....
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   CBufferType* input_gradient = prevLayer.GetGradients();
   if(!m_cOpenCL)
     {
      MATRIX g = m_cGradients.m_mMatrix;
      if(m_bTransposedOutput)
        {
         if(!g.Reshape(m_iNeurons, m_iWindowOut))
            return false;
        }
      else
        {
         if(!g.Reshape(m_iWindowOut, m_iNeurons))
            return false;
         g = g.Transpose();
        }
    ....
     }
   else  // Блок OpenCL
     {
    ....
     }
//---
   return true;
  }
```

А затем — и в метод распределение градиента до уровня матрицы весов *CNeuronConv::CalcDeltaWeights*.

```
bool CNeuronConv::CalcDeltaWeights(CNeuronBase *prevLayer)
  {
//--- блок контролей
    ....
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   CBufferType *input_data = prevLayer.GetOutputs();
   if(!m_cOpenCL)
     {
    ....
      //---
      MATRIX g = m_cGradients.m_mMatrix;
      if(m_bTransposedOutput)
        {
         if(!g.Reshape(m_iNeurons, m_iWindowOut))
            return false;
         g = g.Transpose();
        }
      else
        {
         if(!g.Reshape(m_iWindowOut, m_iNeurons))
            return false;
        }
      m_cDeltaWeights.m_mMatrix += g.MatMul(inp);
     }
   else  // Блок OpenCL
     {
    ....
     }
//---
   return true;
  }
```

Как видите, изменения не такие глобальные, но они дают нам больше свободы настроек.

5.1.2.2 Методы обратного прохода Self-Attention

В предыдущем разделе мы рассмотрели метод прямого прохода блока *Encoder* архитектурного решения Трансформер. Данный блок включает в себя механизм внимания *Self-Attention* (самовнимание) с последующей обработкой двумя полносвязными нейронными слоями. Особенность механизма *Self-Attention* заключается в определении зависимостей между элементами последовательности. При этом каждый элемент последовательности представлен в виде вектора свойств фиксированной длины. Каждый элемент последовательности в рамках одного нейронного слоя обрабатывается блоком *Encoder* с одним набором весовых коэффициентов. Это позволило нам использовать ранее разработанные сверточные слои для решения ряда задач. Организация прямого прохода очень важная часть алгоритма работы нейронных сетей. Мы используем его как при обучении наших моделей нейронных сетей, так и в процессе промышленной эксплуатации. Но обучение нейронной сети невозможно без обратного прохода. Поэтому сейчас мы посмотрим на организацию обратного прохода в нашем классе механизма внимания.

Напомню, что создавали мы свой класс наследником от базового класса нейронного слоя. За организацию обратного прохода в нем отвечает несколько методов:

- *CNeuronBase::CalcOutputGradient* — метод расчета градиента ошибки слоя результатов;
- *CNeuronBase::CalcHiddenGradient* — метод расчета градиента ошибки через скрытый слой;
- *CNeuronBase::CalcDeltaWeights* — метод расчета градиента ошибки до уровня матрицы весов;
- *CNeuronBase::UpdateWeights* — метод обновления весовых коэффициентов.

Все методы были созданы виртуальными для возможности переопределения в классах-наследниках. В нашем классе мы не будем переопределять только первый метод.

Работу над методами мы осуществим в соответствии с логикой метода обратного распространения градиента ошибки. Первым мы переопределим метод расчета градиента ошибки через скрытый слой *CNeuronAttention::CalcHiddenGradient*. Из трех переопределяемых методов этот, наверное, наиболее сложный в понимании и организации. Ведь именно в этом методе нам нужно будет повторить весь путь прямого прохода, но в обратном порядке. При этом нам предстоит найти производные от всех операций, используемых при прямом проходе.

В параметрах метода мы получаем указатель на объект предыдущего слоя, в буфер которого нам предстоит сохранить результат операций. Следом в теле метода организовываем блок проверок актуальности указателей на объекты. Но здесь я решил не останавливаться на проверке всех объектов, а сделал лишь проверку тех объектов, которые не проверяются при вызове методов внутренних классов. Такое решение было принято в попытке исключить повторные проверки актуальности объектов в процессе выполнения операций метода.

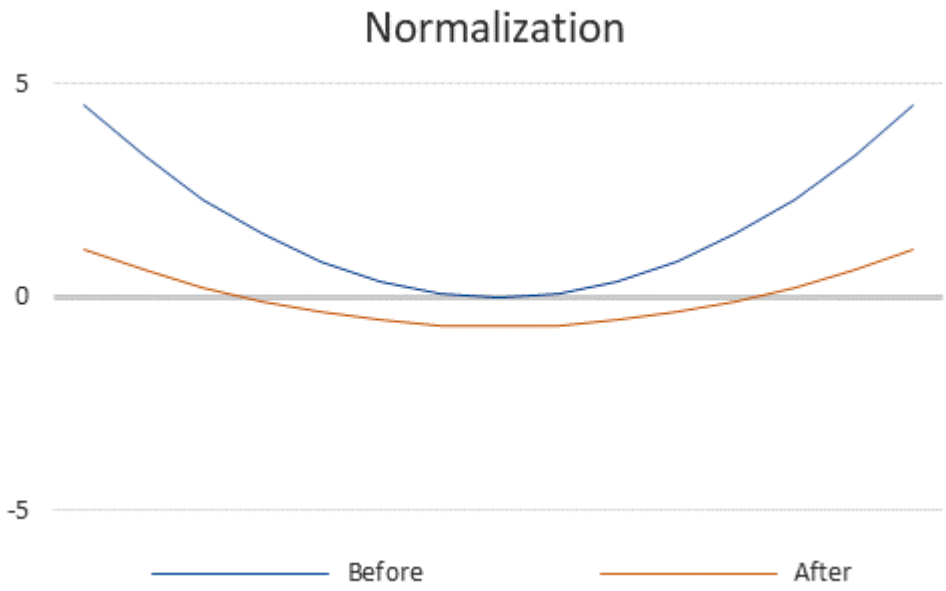
```
bool CNeuronAttention::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
//--- проверяем актуальность всех объектов
   if(!m_cOutputs || !m_cGradients ||
      m_cOutputs.Total() != m_cGradients.Total())
      return false;
```

Далее начинается самое интересное — распределение градиента ошибки в обратном порядке алгоритма прямого прохода. Посмотрим на алгоритм прямого прохода. Он завершается нормализацией результатов, которая осуществляется по формулам.

$$\bar{a} = \frac{1}{h}\sum_{i=1}^{h} a_i$$

$$\hat{a} = \frac{a - \bar{a}}{\sqrt{\frac{1}{h}\sum_{i=1}^{h}(a - \bar{a})^2}}$$

Что же такое процесс нормализации? Это процесс изменения статистических показателей выборки, приближение ее к каким-то заданным параметрам. Чаще всего это среднее значение и среднеквадратическое отклонение, как и в нашем случае. Среднее значение мы приравниваем к нулю, а среднеквадратическое отклонение приводим к единице. В результате такой операции график функции смещается и масштабируется, как показано на рисунке.



Влияние нормализации на график функции

По сути, в рамках алгоритма *Self-Attention* процесс нормализации данных используется в качестве функции активации нейронного слоя. Только, в отличие от последней, не изменяет структуру данных.

Но мы не будем сейчас вдаваться в подробности вывода производной сложной функции нормализации данных. Процесс корректировки градиента ошибки мы вынесли в отдельный метод.

```
//--- корректировка градиента на нормализацию
   if(!NormlizeBufferGradient(m_cOutputs, m_cGradients, GetPointer(m_cStd), 1))
      return false;
```

Далее мы можем воспользоваться методами наших внутренних слоев блока *FeedForward* и провести градиент ошибки до внутреннего слоя хранения результатов блока внимания.

```
//--- проводим градиент через слои блока Feed Forward
   if(!m_cFF2.CalcHiddenGradient(GetPointer(m_cFF1)))
      return false;
   if(!m_cFF1.CalcHiddenGradient(GetPointer(m_cAttentionOut)))
      return false;
```

На этом этапе нужно вспомнить, что в методе прямого прохода перед нормализацией слоя результатов мы складывали значение двух буферов (результатов блоков *FeedForward* и *Self-Attention*). Следовательно, и градиент ошибки нужно провести по обоим веткам алгоритма. Поэтому сложим два буфера градиента. Для облегчения доступа к буферу внутреннего слоя хранения результатов *Self-Attention* создадим локальный указатель на объекты.

```
   CBufferType *attention_grad = m_cAttentionOut.GetGradients();
   if(!attention_grad.SumArray(m_cGradients))
      return false;
```

Скорректируем градиент ошибки на величину среднеквадратического отклонения.

```
//--- корректировка градиента на нормализацию
   if(!NormlizeBufferGradient(m_cAttentionOut.GetOutputs(), attention_grad,
                                                   GetPointer(m_cStd), 0))
      return false;
```

После сложения двух тензоров градиента ошибки нам предстоит распределить градиент ошибки между внутренними слоями *m_cQuerys*, *m_cKeys* и *m_cValues.* Напомню, что при прямом проходе блок алгоритма потока данных от указанных нейронных слоев до буфера результатов Self-Attention мы воссоздавали полностью. Следовательно, нам предстоит создать и обратный процесс. И как всегда, здесь мы создадим разветвление алгоритма в зависимости от вычислительного устройства. Сейчас мы рассмотрим построение алгоритма стандартными средствами *MQL5*. К реализации механизма многопоточных вычислений средствами *OpenCL* мы вернемся чуть позже.

```
//--- разветвление алгоритма по вычислительному устройству
   if(!m_cOpenCL)
     {
      MATRIX values, gradients;
```

В начале блока MQL5 мы создадим две матрицы для хранения промежуточных данных *values* и *gradients*.

Первым мы перенесем градиент ошибки на нейронный слой значений *m_cValues*. Именно значения буфера результатов этого нейронного слоя мы умножали на коэффициенты зависимостей матрицы *Score* для определения результатов блока *Self-Attention* при прямом проходе. Сейчас мы выполняем обратную операцию. Как мы уже говорили, производная от операции умножения равна второму множителю. В нашем случае это коэффициенты матрицы *Score*.

Сразу напомню размерности тензоров данных:

- матрица Score квадратная со стороной равной количеству элементов в последовательности;
- буферы нейронных слоев *m_cValues* и *m_cAttentionOut* имеют количество строк равное количеству элементов последовательности и количество элементов в каждой строке равно размеру вектора описания одного элемента последовательности.

С целью предотвращения возможных несоответствий размеров матриц мы приведем матрицу градиентов ошибки к необходимому формату.

```
if(attention_grad.GetData(gradients, false) < (int)m_cOutputs.Total())
   return false;
if(!gradients.Reshape(m_iUnits, m_iWindow))
   return false;
```

Каждый элемент последовательности из *m_cValues* влияет на все элементы последовательности *m_cAttentionOut* с соответствующим коэффициентом из матрицы *m_cScores.*

Для организации процесса переноса градиента ошибки в буфер нейронного слоя *m_cValues* нам необходимо умножить транспонированную матрицу коэффициентов зависимости *m_cScores* на матрицу градиентов ошибки *gradients*.

```
//--- распределение градиента на Values
m_cValues.GetGradients().m_mMatrix =
                              m_cScores.m_mMatrix.Transpose().MatMul(gradients);
```

Далее мы распределим градиенты ошибки на *m_cQuerys* и *m_cKeys.* Оба нейронных слоя участвовали в создании матрицы коэффициентов зависимостей *m_cScores.* Следовательно, вначале нам надо определить градиент ошибки на матрице коэффициентов зависимости.

При прямом проходе для получения результата *Self-Attention* мы умножали матрицу *m_cScores* на тензор результатов нейронного слоя *m_cValues.* Мы уже определили градиент ошибки для нейронного слоя. Теперь нам надо провести градиент ошибки по второй ветке алгоритма и распределить его на значения матрицы коэффициентов зависимостей. Поэтому нам нужно будет умножить градиент ошибки на транспонированный буфер результатов нейронного слоя *m_cValues*.

```
gradients = gradients.MatMul(values.Transpose());
```

Напомню, что при прямом проходе значения матрицы были нормализованы функцией *Softmax* в рамках запросов *Query*. Сложность вычисления данной функции и ее производной заключается в необходимости вычислений сразу по всему массиву нормализации. В отличие от других функций, производной вектора значений будет матрица. Это связано с природой самой функции *Softmax*. Изменение одного элемента вектора исходный данных ведет к изменению всей последовательности нормализованного результата, ведь сумма всех элементов вектора результатов всегда равна единице. Следовательно, для корректного распределения градиента ошибки нам необходимо работать в разрезе запросов *Query*.

Математическая формула производной функции *Softmax* имеет вид:

$$\frac{df(x_i)}{dx_i} = \begin{cases} i = j & f(x_i) * (1 - f(x_i)) \\ i \neq j & -f(x_j) f(x_i) \end{cases}$$

Мы же воспользуемся ее матричным представлением:

$$\frac{df(X)}{dX} = f(X)^T (E - f(X))$$

где E — единичная квадратная матрица, размер которой равен количеству элементов последовательности.

Реализация данного подхода приведена ниже. В цикле мы определяем производную каждой отдельной строки матрицы коэффициентов зависимости. После умножения полученной матрицы на вектор градиентов соответствующей строки мы получаем вектор скорректированных градиентов ошибки. Не забываем, что перед нормализацией матрицы коэффициентов зависимостей *Score* мы делили ее значения на квадратный корень из размерности вектора описания одного элемента в тензоре *Key.* Соответственно, повторим эту процедуру и для градиента ошибки. Логика данной операции проста: деление на константу воспринимается как умножение на обратное константе число, а производная от операции умножения равна ее второму множителю.

$$\left(\frac{x}{const}\right)' = \left(\frac{1}{const}x\right)' = \frac{1}{const}$$

Результат вышеуказанных операций заменит анализируемую строку матрицы градиентов.

```
      for(int r = 0; r < m_iUnits; r++)
        {
         MATRIX ident = MATRIX::Identity(m_iUnits, m_iUnits);
         MATRIX ones = MATRIX::Ones(m_iUnits, 1);
         MATRIX result = MATRIX::Zeros(1, m_iUnits);
         if(!result.Row(m_cScores.m_mMatrix.Row(r), 0))
            return false;
         result = ones.MatMul(result);
         result = result.Transpose() * (ident - result);
         VECTOR temp = result.MatMul(gradients.Row(r));
         if(!gradients.Row(temp / sqrt(m_iKeysSize), r))
            return false;
        }
```

После получения скорректированного градиента ошибки для каждого отдельного коэффициента зависимости мы распределяем его на соответствующие векторы тензоров *Query* и *Key.* С этой целью умножим матрицу скорректированных градиентов коэффициентов зависимости на противоположную матрицу.

```
      m_cQuerys.GetGradients().m_mMatrix =
                                    gradients.MatMul(m_cKeys.GetOutputs().m_mMatrix
      m_cKeys.GetGradients().m_mMatrix =
                        gradients.Transpose().MatMul(m_cQuerys.GetOutputs().m_mMatrix
     }

    else // Блок OpenCL
     {
      return false;
     }
```

На этом завершается блок разделения алгоритма по вычислительному устройству. В блоке работы с технологией *OpenCL* мы пока оставим возврат отрицательного результата и вернемся к нему чуть позже. А сейчас идем дальше по нашему алгоритму обратного распространения ошибки. После получения градиента ошибки на выходе внутренних нейронных слоев, нам остается довести его до предыдущего слоя.

Как вы помните, при прямом проходе исходные данные используются в четырех ветках алгоритма:

- подаются на вход внутреннего слоя m_cQuerys;

- подаются на вход внутреннего слоя m_cKeys;
- подаются на вход внутреннего слоя m_cValues;
- суммируются с выходом блока *Self-Attention* перед нормализацией слоя.

Следовательно, в буфер градиентов ошибки предыдущего слоя мы должны собрать градиент ошибки со всех 4-х направлений. Алгоритм работы аналогичен построенному ранее процессу сложения буферов в рекуррентном *LSTM*-блоке. Только мы не будем создавать отдельный буфер для накопления данных, а воспользуемся уже имеющимся. Градиент ошибки на выходе блока *Self-Attention* у нас уже посчитан в буфере нейронного слоя *m_cAttentionOut.* В нем и будем накапливать промежуточные градиенты ошибки.

Мы будем поочередно вызывать метод передачи градиента на предыдущий слой *CalcHiddenGradient* для каждого внутреннего слоя с передачей ему указателя на предыдущий нейронный слой. После успешного выполнения метода сложим полученный результат с предварительно накопленным градиентом ошибки в буфере градиентов нейронного слоя *m_cAttentionOut.*

```
//--- перенос градиента ошибки на предыдущий слой
   if(!m_cValues.CalcHiddenGradient(prevLayer))
      return false;
   if(!attention_grad.SumArray(prevLayer.GetGradients()))
      return false;
   if(!m_cQuerys.CalcHiddenGradient(prevLayer))
      return false;
   if(!attention_grad.SumArray(prevLayer.GetGradients()))
      return false;
   if(!m_cKeys.CalcHiddenGradient(prevLayer))
      return false;
   if(!prevLayer.GetGradients().SumArray(attention_grad))
      return false;
//---
   return true;
  }
```

Обратите внимание, что в первых двух случаях мы записывали сумму двух буферов градиентов ошибки в буфер внутреннего нейронного слоя. А последний раз, наоборот, сохранили сумму двух буферов в буфер градиентов предыдущего нейронного слоя. Все дело в том, что метод *CalcHiddenGradient* внутреннего нейронного слоя перезаписывает значения в буфере градиентов нейронного слоя, указанного в параметрах. Поэтому нам потребовалось накапливать промежуточные градиенты в другом буфере. Но в конце метода нам надо передать градиент ошибки на предыдущий слой. Поэтому при последнем суммировании буферов мы сразу записываем сумму в буфер предыдущего нейронного слоя, тем самым избегая излишнее копирование данных.

Выше был анонсирован метод корректировки градиента ошибки на процесс нормализации данных *NormlizeBufferGradient*. Что же представляет из себя процесс нормализации и в чем сложность определения производной функции? На первый взгляд, мы от каждого элемента нормализуемого массива отнимаем значение средней арифметической, а полученную разницу делим на среднеквадратическое отклонение.

$$\hat{a} = \frac{a - \bar{a}}{\sqrt{\sigma^2}}$$

Если бы отнимали и делили на константы, не было бы никаких сложностей. При вычитании константы производная не меняется.

$$(f(a) - const)' = (f(a))' - 0 = (f(a))'$$

Производная из деления на константу равна отношению 1 к константе.

$$\left(\frac{f(a)}{const}\right)' = \left(\frac{1}{const} f(a)\right)' = \frac{1}{const}(f(a))'$$

Но проблема в том, что обе средние являются функциями. При изменении любого одного значения в тензоре исходных данных изменяется значение средних и, как следствие, все значения тензора на выходе блока нормализации. Это значительно усложняет вычисление производной всей функции. Мы не будем сейчас выводить их, а воспользуемся готовым результатом.

$$\frac{\partial f(a_i)}{\partial \sigma^2} = \sum_{i=1}^{m} \frac{\partial f(a_i)}{\partial \hat{a}_i} * (a_i - \bar{a}) * \frac{-1}{2}(\sigma^2 + \varepsilon)^{-\frac{3}{2}}$$

$$\frac{\partial f(a_i)}{\partial \bar{a}} = \left(\sum_{i=1}^{m} \frac{\partial f(a_i)}{\partial \hat{a}_i} * \frac{-1}{\sqrt{\sigma^2 + \varepsilon}}\right) + \frac{\partial f(a_i)}{\partial \sigma^2} * \frac{\sum_{i=1}^{m} -2(a_i - \bar{a})}{m}$$

$$\frac{\partial f(a_i)}{\partial a_i} = \frac{\partial f(a_i)}{\partial \hat{a}_i} * \frac{1}{\sqrt{\sigma^2 + \varepsilon}} + \frac{\partial f(a_i)}{\partial \sigma^2} * \frac{2(a_i - \bar{a})}{m} + \frac{\partial f(a_i)}{\partial \bar{a}} * \frac{1}{m}$$

Реализуем приведенные формулы в коде с помощью матричных операций *MQL5*. В параметрах метод получает указатели на 3 буфера данных:

- *output* — буфер результатов нормализации данных прямого прохода;
- *gradient* — буфер градиентов ошибки. Используется как для получения исходных данных, так и для записи результатов;
- *std* — буфер среднеквадратических отклонений, вычисленных при прямом проходе.

Как можно заметить, в параметрах нет буфера данных до нормализации и значения средней арифметической, вычисленной при прямом проходе. Мы просто заменили разницу ненормализованных данных и средней арифметической на произведение нормализованных данных и среднеквадратического отклонения.

$$\hat{a} = \frac{a - \bar{a}}{\sqrt{\sigma^2}} \Rightarrow \hat{a}\sqrt{\sigma^2} = a - \bar{a}$$

И, конечно, мы не ожидаем нулевое среднеквадратическое отклонение. Добавим проверку для предотвращения критической ошибки деления на ноль.

```
bool CNeuronAttention::NormlizeBufferGradient(CBufferType *output,
                                              CBufferType *gradient,
                                              CBufferType *std,
                                              uint std_shift)
  {
//---
   if(!m_cOpenCL)
     {
      if(std.At(std_shift) <= 0)
         return true;
      MATRIX ScG = gradient.m_mMatrix / std.m_mMatrix[0, std_shift];
      MATRIX ScOut = output.m_mMatrix * std.m_mMatrix[0, std_shift];
      TYPE dSTD = (gradient.m_mMatrix * output.m_mMatrix / (-2 * MathPow(std.m_mMatri
      TYPE dMean = -1 * ScG.Sum() - 2 * dSTD / (TYPE)output.Total() * ScOut.Sum();
      gradient.m_mMatrix = ScG + (ScOut * dSTD * 2 + dMean) / (TYPE)output.Total();
     }
    else // Блок OpenCL
     {
      return false;
     }
//---
   return true;
  }
```

Кроме метода распределения градиента через скрытый слой, алгоритм обратного распределения градиента ошибки во всех ранее рассмотренных нейронных слоя обычно представлен еще двумя методами:

- *CalcDeltaWeights* — метод расчета градиента ошибки до уровня матрицы весов;
- *UpdateWeights* — метод обновления весовых коэффициентов.

Рассматриваемый нами класс *CNeuronAttention* не будет исключением. В нем мы также переопределим два указанных метода. Алгоритм их прост и тривиален: мы просто будем вызывать поочередно одноименные методы всех внутренних нейронных слоев. И конечно, проверим результат выполнения операций.

```
bool CNeuronAttention::CalcDeltaWeights(CNeuronBase *prevLayer)
  {
   if(!m_cFF2.CalcDeltaWeights(GetPointer(m_cFF1)))
      return false;
   if(!m_cFF1.CalcDeltaWeights(GetPointer(m_cAttentionOut)))
      return false;
   if(!m_cQuerys.CalcDeltaWeights(prevLayer))
      return false;
   if(!m_cKeys.CalcDeltaWeights(prevLayer))
      return false;
   if(!m_cValues.CalcDeltaWeights(prevLayer))
      return false;
//---
   return true;
  }

bool CNeuronAttention::UpdateWeights(int batch_size, TYPE learningRate,
                                     VECTOR &Beta, VECTOR &Lambda)
  {
   if(!m_cQuerys.UpdateWeights(batch_size, learningRate, Beta, Lambda))
      return false;
   if(!m_cKeys.UpdateWeights(batch_size, learningRate, Beta, Lambda))
      return false;
   if(!m_cValues.UpdateWeights(batch_size, learningRate, Beta, Lambda))
      return false;
   if(!m_cFF1.UpdateWeights(batch_size, learningRate, Beta, Lambda))
      return false;
   if(!m_cFF2.UpdateWeights(batch_size, learningRate, Beta, Lambda))
      return false;
//---
   return true;
  }
```

Таким образом мы реализовали три метода, составляющие алгоритм обратного прохода нашего блока внимания.

5.1.2.3 Методы работы с файлами

Мы уже построили методы прямого и обратного проходов нашего слоя внимания. Можно добавить слой в свою модель и обучить ее, но мы же не будем каждый раз заново обучать модель перед ее использованием. Нам нужна возможность сохранить однажды обученную модель в файл и при необходимости загрузить из файла готовую к использованию нейронную сеть. За работу с файлами в нашем базовом нейронном слое отвечает два метода: *Save* и *Load.* Для корректного выполнения функции в нашем новом слое необходимо переопределить указанные методы.

Подобную итерацию мы осуществляем при создании каждого нового типа нейронного слоя. Сейчас мы пойдем уже проторенной дорогой: обратим свой взор на структуру нашего класса и определим, что нужно сохранить в файл, а какие переменные и объекты мы просто создадим и инициализируем начальными значениями.

В первую очередь, нужно сохранить внутренние нейронные слои, содержащие матрицы весовых коэффициентов *m_cQuerys*, *m_cKeys*, *m_cValues*, *m_cFF1* и *m_cFF2*. Кроме того, нам нужно сохранить значения переменных, определяющих архитектуру нейронного слоя: *m_iWindow*, *m_iUnits* и *m_iKeysSize*.

Нам нет необходимости сохранять в файл никакой информации из буфера *m_cScores*, так как он содержит только промежуточные данные, перезаписываемые при каждом прямом проходе. Его размер легко определить на основе количества элементов в последовательности, записанного в переменной *m_iUnits*.

Внутренний слой *m_cAttentionOut* не содержит матрицы весовых коэффициентов, и его данные, так же как и данные буфера *m_cScores*, перезаписываются на каждой итерации прямого и обратного прохода. Но давайте посмотрим на ситуацию с другой стороны. Вспомним процедуру инициализации нейронного слоя:

- создать объект описания нейронного слоя,
- заполнить объект описания нейронного слоя необходимой информацией,
- вызвать метод инициализации нейронного слоя с передачей описания,
- удалить объект описания нейронного слоя.

В то же время вызов метода сохранения для базового нейронного слоя без матрицы весов запишет в файл всего 3 целочисленных числа с общим размером в 12 байт. То есть пожертвовав 12 байт пространства на диске мы сокращаем свои трудозатраты на написание кода инициализации нейронного слоя в методе загрузки данных.

```
class CNeuronAttention    :  public CNeuronBase
  {
protected:
   CNeuronConv       m_cQuerys;
   CNeuronConv       m_cKeys;
   CNeuronConv       m_cValues;
   CBufferType       m_cScores;
   int               m_cScoreGrad;
   int               m_cScoreTemp;
   CNeuronBase       m_cAttentionOut;
   CNeuronConv       m_cFF1;
   CNeuronConv       m_cFF2;
   //---
   int               m_iWindow;
   int               m_iUnits;
   int               m_iKeysSize;
   CBufferType       m_cStd;
   //---
   virtual bool      NormlizeBuffer(CBufferType *buffer, CBufferType *std,
                                                          uint std_shift);
   virtual bool      NormlizeBufferGradient(CBufferType *output,
                       CBufferType *gradient, CBufferType *std, uint std_shift);

public:
                     CNeuronAttention(void);
                    ~CNeuronAttention(void);
   //---
   virtual bool      Init(const CLayerDescription *desc) override;
   virtual bool      SetOpenCL(CMyOpenCL *opencl) override;
   virtual bool      FeedForward(CNeuronBase *prevLayer) override;
   virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer) override;
   virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                   VECTOR &Beta, VECTOR &Lambda) override;
   //--- методы работы с файлами
   virtual bool      Save(const int file_handle) override;
   virtual bool      Load(const int file_handle) override;
   //--- метод идентификации объекта
   virtual int       Type(void) override  const { return(defNeuronAttention); }
  };
```

После того, как мы определились с объектами для записи данных в файл, можно приступать к работе над нашими методами. Начнем работу с метода записи данных в файл *Save*. В параметрах метод получает хендл файла для записи данных. Но мы не будем сразу проверять полученный хендл. Вместо этого мы вызовем аналогичный метод родительского класса, в котором уже реализованы все контрольные точки и сохранение унаследованных объектов. Результат выполнения метода родительского класса укажет на результат отработки блока контролей.

```
bool CNeuronAttention::Save(const int file_handle)
  {
   if(!CNeuronBase::Save(file_handle))
      return false;
```

После выполнения метода родительского класса мы поочередно вызываем метод сохранения внутренних объектов. При этом не забываем проверить результаты выполнения операций.

```
   if(!m_cQuerys.Save(file_handle))
      return false;
   if(!m_cKeys.Save(file_handle))
      return false;
   if(!m_cValues.Save(file_handle))
      return false;
   if(!m_cAttentionOut.Save(file_handle))
      return false;
   if(!m_cFF1.Save(file_handle))
      return false;
   if(!m_cFF2.Save(file_handle))
      return false;
```

После сохранения данных внутренних объектов сохраним значения переменных, определяющих архитектуру нейронного слоя. Конечно, мы проверяем результат выполнения операций.

```
   if(FileWriteInteger(file_handle, m_iUnits) <= 0)
      return false;
   if(FileWriteInteger(file_handle, m_iWindow) <= 0)
      return false;
   if(FileWriteInteger(file_handle, m_iKeysSize) <= 0)
      return false;
//---
   return true;
  }
```

После успешного сохранения всех необходимых данных завершаем метод с положительным результатом.

После создания метода записи данных переходим к работе над методом чтения данных *Load*. В параметрах метод получает хендл файла для чтения данных. Как и в случае с записью данных, мы не создаем новый блок контролей в своем методе, а вместо этого вызовем метод родительского класса, в котором уже реализованы все контроли и считывание унаследованных объектов и переменных. Проверка результата выполнения метода родительского класса сразу нам расскажет и о прохождении блока контролей, и о загрузке данных унаследованных объектов и переменных.

```
bool CNeuronAttention::Load(const int file_handle)
  {
   if(!CNeuronBase::Load(file_handle))
      return false;
```

После успешного выполнения метода загрузки данных родительского класса мы поочередно будем считывать данные внутренних объектов. Напомню, что считывание данных из файла осуществляется в строгом соответствии последовательности записи данных. При записи данных в файл первым мы сохранили информацию из внутреннего нейронного слоя *m_cQuerys*.

Следовательно, первым мы будем загружать данные именно в этот объект. Но не забываем про нюанс загрузки внутренних нейронных слоев: мы сначала проверяем тип загружаемого объекта и только потом вызываем метод загрузки соответствующего объекта.

```
if(FileReadInteger(file_handle) != defNeuronConv || !m_cQuerys.Load(file_handle))
   return false;
```

Аналогичный алгоритм повторяем для всех ранее сохраненных объектов.

```
if(FileReadInteger(file_handle) != defNeuronConv || !m_cKeys.Load(file_handle))
   return false;
if(FileReadInteger(file_handle) != defNeuronConv || !m_cValues.Load(file_handle))
   return false;
if(FileReadInteger(file_handle) != defNeuronBase ||
   !m_cAttentionOut.Load(file_handle))
   return false;
if(FileReadInteger(file_handle) != defNeuronConv || !m_cFF1.Load(file_handle))
   return false;
if(FileReadInteger(file_handle) != defNeuronConv || !m_cFF2.Load(file_handle))
   return false;
```

После загрузки данных объектов внутренних нейронных слоев мы считываем из файла значение переменных, определяющих архитектуру нашего нейронного слоя внимания.

```
m_iUnits = FileReadInteger(file_handle);
m_iWindow = FileReadInteger(file_handle);
m_iKeysSize = FileReadInteger(file_handle);
```

Дальше нам остается лишь инициализировать буфер коэффициентов зависимостей *m_cScores* нулевыми значениями. Предварительно мы не изменяем размер буфера, так как методом инициализации буфера предусмотрено изменение его размера до необходимого уровня.

```
if(!m_cScores.BufferInit(m_iUnits, m_iUnits, 0))
   return false;
```

Теперь мы загрузили все данные и инициализировали объекты. Остается вспомнить, что для исключения излишнего копирования данных мы сделали подмену указателей буферов результатов и градиентов внутреннего слоя *m_cFF2* и самого слоя внимания. И без этой подмены указателей вся работа нашего нейронного слоя будет некорректной. Но если по какой-либо причине мы заново создадим объект внутреннего слоя *m_cFF2*, то будут созданы и новые объекты буферов указанного внутреннего нейронного слоя. В таком случае нам нужно осуществить такую подмену указателей повторно. В то же время, если обе переменных содержат указатель на один объект, то, удалив объект по одному указателю, мы получим не действительный указатель и во второй переменной. Это узкий момент, с которым нужно быть аккуратней.

Мы, конечно, добавим подмену буферов, но предварительно проверим соответствие указателей.

```
   if(m_cFF2.GetOutputs() != m_cOutputs)
     {
      if(m_cOutputs)
         delete m_cOutputs;
      m_cOutputs = m_cFF2.GetOutputs();
     }

   if(m_cFF2.GetGradients() != m_cGradients)
     {
      if(m_cGradients)
         delete m_cGradients;
      m_cGradients = m_cFF2.GetGradients();
     }
//---
   SetOpenCL(m_cOpenCL);
//---
   return true;
  }
```

После успешного выполнения всех операций выходим из метода с положительным результатом.

На этом можно считать завершенной работу по созданию нейронного слоя внимания стандартными средствами языка *MQL5*. В таком варианте мы уже можем вставить нейронный слой внимания в свою модель и проверить его работоспособность. Но для наиболее эффективного использования созданного класса нам необходимо дополнить его методы средствами многопоточных вычислений.

5.1.3 Организация параллельных вычислений в блоке внимания

В предыдущих разделах мы построили рабочий алгоритм блока внимания стандартными средствами языка *MQL5.* Вы уже можете добавить блок внимания в сою модель и испытать качество работы механизма *Self-Attention.* Но посмотрите на структуру блока. В его работе мы использовали пять внутренних слоев, создали алгоритм передачи данных между ними в прямом и обратном направлении. И немаловажен тот факт, что каждый элемент последовательности, описанный вектором значений, обрабатывается с использованием единых матриц весовых коэффициентов, но независимо друг от друга. Это позволяет легко распределить операции по параллельным потокам, что даст нам возможность совершать полный набор операций в более короткие временные интервалы. И да, мы изначально определились, что будем создавать библиотеку с возможностью использования двух технологий. Тем самым мы даем пользователю возможность самостоятельно протестировать и выбрать наиболее подходящую технологию для применения в каждом конкретном случае.

Как и ранее, блок параллельных вычислений организуем с использованием технологии *OpenCL*. Для использования данной технологии нам потребуется выполнить два этапа работы:

- создание программы *OpenCL*,
- внесение изменений в основную программу.

Код программы *OpenCL* мы добавим в ранее созданный файл opencl_program.cl. Именно в этом файле мы собрали все кернелы программы *OpenCL,* используемые в работе предыдущих классов. Для организации работы нашего класса внимания потребуется создание шести кернелов. В этих кернелах нам предстоит организовать поток информации между используемыми внутренними нейронными слоями в прямом и обратном направлении

Первым мы создадим кернел прямого прохода *AttentionFeedForward*. Кратко напомню последовательность операций при прямом проходе блока *Self-Attention*:

1. Исходные данные подаются на вход трем внутренним сверточным нейронным слоям *m_cQuery*, *m_cKeys*, *m_cValues.*

$$Input * W_Q = Query$$
$$Input * W_K = Key$$
$$Input * W_V = Value$$

2. Тензоры результатов *m_cQuery* и *m_cKeys* перемножаются для получения матрицы зависимостей *m_cScores*.

$$Query * Key^T = S$$

3. Значения матрицы *m_cScores* делятся на квадратный корень от размера вектора описания одного элемента последовательности *m_cKeys* и нормализуются функцией *Softmax* в разрезе строк (запросов *m_cQuery*).

$$Softmax\left(\frac{S}{\sqrt{Key_{size}}}\right) = Score$$

4. Нормализованная матрица *m_cScores* перемножается с тензором результатов нейронного слоя *m_cValues* для получения результатов *Self-Attention.*

$$Score * Value = Out_{Self-Attention}$$

5. Результаты блока *Self-Attention* складываются с исходными данными и нормализуются.

$$Normalize(Input + Out_{Self-Attention}) = Output$$

6. Полученный тензор служит исходными данными для блока из двух сверточных слоев *m_cFF1* и *m_cFF2*.

Если пункты 1 и 6 покрываются использованием ранее рассмотренного класса сверточных слоев, в котором уже реализован блок многопоточных вычислений, то остальные пункты нам предстоит реализовать в новом кернеле.

Для организации указанных операций нам потребуется передать в кернел шесть буферов данных и два параметра. Чтобы код программы был более читабелен, наименования буферов и переменных будут созвучны с наименованием соответствующих матриц в описании алгоритма.

```
__kernel void AttentionFeedForward(__global TYPE *querys,
                                   __global TYPE *keys,
                                   __global TYPE *scores,
                                   __global TYPE *values,
                                   __global TYPE *outputs,
                                   int window,
                                   int key_size)
  {
```

Как можно было заметить из описания алгоритма *Self-Attention,* основной аналитической единицей в данном методе является элемент последовательности, описываемый вектором значений. Для языковых моделей это чаще всего слово. В случае анализа финансовых рынков это бар. Именно между этими элементами последовательности определяются коэффициенты взаимных зависимостей. С учетом этих коэффициентов корректируются значения векторов

описания элементов. Поэтому вполне логично разделить операции на потоки именно по элементам последовательности.

Следовательно, в теле кернела первым делом определим анализируемый элемент последовательности по идентификатору нашего потока. В то же время общее количество запущенных потоков подскажет нам количество элементов в последовательности. Здесь же мы сразу определим смещение в тензоре запросов и матрице коэффициентов зависимостей до первого анализируемого значения.

```
const int q = get_global_id(0);
const int units = get_global_size(0);
int shift_query = key_size * q;
int shift_scores = units * q;
TYPE summ = 0;
```

Напомню, что для нормализации данных функцией *Softmax* нам потребуется сумма экспонент всех нормализуемых значений. Для ее подсчета добавим переменную с начальным нулевым значением.

После проведения подготовительной работы определим значения одного вектора из матрицы коэффициентов зависимостей, относящегося к расчетам зависимостей анализируемого элемента последовательности. Для этого организуем цикл с числом итераций равным количеству элементов в последовательности. В теле цикла мы будем поочередно перемножать вектор *Query* анализируемого элемента последовательности со всеми векторами тензора *Key*. Для каждого результата умножения векторов возьмем экспоненциальное значение и запишем его в соответствующий элемент матрицы *Score.* И конечно прибавим к нашей накопительной сумме всех значений вектора для последующей нормализации значений.

```
for(int s = 0; s < units; s++)
  {
   TYPE score = 0;
   int shift_key = key_size * s;
   for(int k = 0; k < key_size; k ++)
      score += querys[shift_query + k] * keys[shift_key + k];
   score = exp(score / sqrt((TYPE)key_size));
   summ += score;
   scores[shift_scores + s] = score;
  }
```

После завершения работы цикла в нашей переменной *summ* будет накоплена сумма всех элементов нашего вектора из тензора коэффициентов зависимостей. Для завершения нормализации значений данного вектора нам остается лишь разделить значение каждого его элемента на общую сумму всех значений вектора.

```
for(int s = 0; s < units; s++)
   scores[shift_scores + s] /= summ;
```

Таким образом, в анализируемом векторе мы получили коэффициенты зависимостей анализируемого элемента последовательности от остальных ее элементов. Сумма всех коэффициентов будет равна единице.

Далее, согласно алгоритму, нам необходимо каждый вектор тензора значений *Value* умножить на соответствующий элемент полученного вектора коэффициентов зависимостей. Полученные

векторы нужно сложить. Итоговый вектор значений и будет результатом работы блока *Self-Attention*.

Перед передачей данных дальше нужно сложить полученные данные с тензором исходных данных и нормализовать их. В теле кернела я предлагаю остановиться на определении результатов работы блока *Self-Attention*. Сложение же матриц и нормализацию данных в пределах всего нейронного слоя будет эффективнее организовать отдельно.

Посмотрим на реализацию такого решения. Чтобы не пересчитывать на каждой итерации, сначала определим смещение в тензорах исходных данных и результатов. Тензоры имеют одинаковую размерность, поэтому и смещение будет одно для обоих случаев. Затем организуем систему из двух вложенных циклов: во внешнем будем перебирать элементы вектора анализируемого элемента последовательности, а во внутреннем будем осуществлять непосредственно вычисление значений каждого элемента вектора результатов. Для этого количество итераций внутреннего цикла будет равняться количеству элементов последовательности. В теле данного цикла будем умножать значения элементов тензора *Value* на соответствующие коэффициенты зависимостей из матрицы *Score.* Полученные произведения будем накапливать в локальной переменной *query*. После завершения итераций вложенного цикла результат запишем в соответствующий элемент тензора результатов.

```
  shift_query = window * q;
  for(int i = 0; i < window; i++)
    {
     TYPE query = 0;
     for(int v = 0; v < units; v++)
        query += values[window * v + i] * scores[shift_scores + v];
     outputs[shift_query + i] = query;
    }
}
```

На этом мы завершим работу над первым кернелом прямого прохода. Следующим этапом создадим кернел сложения двух тензоров. Подобную работу порой выгоднее сделать с использованием матричных операций на стороне основной программы. Операция несложная, и накладные расходы в виде передачи данных вряд ли будут оправданы. У нас сейчас обратная ситуация. Мы организовываем весь процесс на стороне контекста *OpenCL*. Вся информация уже находится в памяти контекста, и для реализации операции на стороне основной программы нам потребуется копирование данных. Для вычислений в контексте перенос данных не нужен Поэтому мы создали кернел *Sum*, в теле которого просто складываем элементы из двух буферов с одним индексом и сохраняем в элемент третьего буфера с тем же индексом.

```
__kernel void Sum(__global TYPE *inputs1,
                  __global TYPE *inputs2,
                  __global TYPE *outputs)
  {
   const int n = get_global_id(0);
//---
   outputs[n] = inputs1[n] + inputs2[n];
  }
```

Процесс нормализации данных имеет более сложную архитектуру. Как вы знаете, ее процесс выражается следующими математическими формулами:

$$\bar{a} = \frac{1}{h}\sum_{i=1}^{h} a_i$$

$$\hat{a} = \frac{a - \bar{a}}{\sqrt{\frac{1}{h}\sum_{i=1}^{h}(a - \bar{a})^2}}$$

Как можно заметить, для вычисления нормализованного значения каждого элемента последовательности необходимо среднее арифметическое и среднеквадратическое отклонение всей последовательности. Для их вычисления нам потребуется организация передачи данных между отдельными потоками. Данную задачу будем решать аналогично многопоточной реализации функции активации *Softmax* — через массив в локальной памяти. Только нам потребуется организовать два блока суммирования значений по всему вектору, ведь до определения средней арифметической мы не можем вычислить дисперсию. Кроме того, до определения дисперсии нельзя вычислить нормализованное значение.

Процесс нормализации организован в кернеле *LayerNormalize*. В параметрах кернел получает указатели на 3 буфера:

- буфер исходных данных,
- буфер результатов,
- буфер для записи параметров среднеквадратического отклонения.

Последний буфер среднеквадратических отклонений нам понадобился для сохранения и передачи данных в кернел обратного прохода.

Кроме того, мы передадим в кернел два параметра: общее количество элементов в нормализуемом буфере и смещение в буфере среднеквадратических отклонений. Напомню, в рамках одного нейронного слоя внимания нормализацию данных мы осуществляем два раза. Нормализуем результаты блоков *Self-Attention* и *FeedForward*.

```
__kernel void LayerNormalize(__global TYPE* inputs,
                             __global TYPE* outputs,
                             __global TYPE* stds,
                             const int total,
                             const int std_shift)
  {
```

В теле кернела определяем идентификаторы потоков и инициализируем локальный массив данных.

```
    uint i = (uint)get_global_id(0);
    uint l = (uint)get_local_id(0);
    uint ls = min((uint)get_local_size(0), (uint)LOCAL_SIZE);
    __local TYPE temp[LOCAL_SIZE];
```

Первым мы определим среднее арифметическое значение элементов буфера. Для этого организуем цикл, в котором каждый поток просуммирует свои значения и результат сохранит в своем элементе локального массива. Так как мы вычисляем среднее арифметическое всего буфера, то и полученное значение разделим на количество элементов в буфере.

```
    uint count = 0;
    do
      {
       uint shift = count * ls + l;
       temp[l] = (count > 0 ? temp[l] : 0) + (shift < total ? inputs[shift] : 0);
       count++;
      }
    while((count * ls + l) < total);
    temp[l] /= (TYPE)total;
    barrier(CLK_LOCAL_MEM_FENCE);
```

Синхронизировать работу потоков будем с помощью функции *barrier*. Так как вычисления потоков не пересекаются, нам достаточно одного барьера в конце блока.

Далее нам необходимо собрать части общей суммы в единое целое. Мы организуем еще один цикл, в теле которого соберем среднее арифметическое буфера в одном элементе локального массива с индексом 0. Результат сохраним в локальную переменную.

```
    count = ls;
    do
      {
       count = (count + 1) / 2;
       temp[l] += (l < count ? temp[l + count] : 0);
       barrier(CLK_LOCAL_MEM_FENCE);
      }
    while(count > 1);
  //---
    TYPE mean = (TYPE) temp[0];
```

Хочу еще раз обратить внимание на расстановку барьеров. Здесь нужно уделить особое внимание работе алгоритма, ведь до каждого барьера должны дойти все потоки. Причем последовательность их посещений также должна быть соблюдена.

После определения среднего арифметического повторим циклы и вычислим среднеквадратическое отклонение.

```
    count = 0;
    do
      {
       uint shift = count * ls + l;
       temp[l] = (count > 0 ? temp[l] : 0) + (shift < total ? (TYPE)pow(inputs[shift]
       count++;
      }
    while((count * ls + l) < total);
    temp[l] /= (TYPE)total;
    barrier(CLK_LOCAL_MEM_FENCE);

    count = ls;
    do
      {
       count = (count + 1) / 2;
       temp[l] += (l < count ? temp[l + count] : 0);
       barrier(CLK_LOCAL_MEM_FENCE);
      }
    while(count > 1);
//---
    TYPE std = (TYPE)sqrt(temp[0]);
    if(l == 0)
       stds[std_shift] = std;
```

Полученное среднеквадратическое отклонение сохраним в буфер. Чтобы исключить одновременную запись значения всеми потоками, сохранять будем только в одном потоке. Для этого сделаем проверку индекса потока перед операцией записи значения в буфер.

Теперь, когда мы вычислили значения средних, можно нормализовать исходные данные. При этом надо учесть, что ограничение размера work group может не позволить организовать отдельный поток для каждого элемента буфера исходных данных. Поэтому нормализацию данных тоже будем осуществлять в цикле.

```
   count = 0;
   while((count * ls + l) < total)
     {
      uint shift = count * ls + l;
      outputs[shift] = (inputs[shift] - mean) / (std + 1e-37f);
      count++;
     }
}
```

На этом мы завершаем работу с кернелами прямого прохода. Продолжая работу над внесением дополнений в программу *OpenCL*, мы переходим к построению обратного прохода. Его алгоритм полностью повторяет проделанный выше путь, но в обратном направлении. В нем нам предстоит распределить градиент ошибки от выхода блока *Self-Attention* до внутренних нейронных слоев *m_cQuery*, *m_cKeys*, *m_cValues.*

Наиболее простым представляется расчет градиента ошибки для внутреннего нейронного слоя *m_cValues.* Напомню, для получения результата блока *Self-Attention* мы перемножали матрицу коэффициентов зависимостей *m_cScores* на тензор результатов нейронного слоя *m_cValues*. Следовательно, для получения градиента ошибки на уровне выхода указанного нейронного слоя нам необходимо умножить полученный от предыдущих операций градиент ошибки на

производную последней операции. В данном случае нам предстоит умножить матрицу коэффициентов зависимостей на тензор градиентов ошибки от предыдущих операций.

$$G_{Value} = Score^T * G_{Self-Attention}$$

После определения градиента ошибки на внутреннем нейронном слое *m_cValues* нужно распределить градиент ошибки еще на два внутренних нейронных слоя, *m_cQuerys* и *m_cKeys*. Но для того, чтобы довести градиент ошибки до уровня указанных нейронных слоев, нужно провести его через матрицу коэффициентов зависимостей.

$$G_{Score} = G_{Self-Attention} * V^T$$

Но если при реализации средствами *MQL5* мы не создаем дополнительный буфер для градиентов ошибки на уровне матрицы коэффициентов зависимостей, то в *OpenCL* существует сложность с выделением динамического массива для записи промежуточных данных о значениях градиента ошибки на уровне матрицы коэффициентов зависимостей. Поэтому здесь создадим два временных буфера данных: один для градиента ошибки нормализованных данных, а второй для градиентов ошибки, скорректированных на производную функции *Softmax*.

Обратите внимание: когда мы пересчитываем градиент ошибки до уровня нейронных слоев *m_cQuerys* и *m_cKeys*, одни и те же элементы матрицы градиентов ошибки коэффициентов зависимостей используются в разных потоках операций. Поэтому весь алгоритм обратного распределения ошибки внутри слоя внимания мы разделим на два блока. В первом мы проведем градиент ошибки до уровня внутреннего нейронного слоя значений *m_cValues* и матрицы коэффициентов *m_cScores.* Во втором блоке доведем градиент ошибки до двух других нейронных слоев, *m_cQuerys* и *m_cKeys.*

Первый блок операций реализуем в кернеле *AttentionCalcScoreGradient*. В параметрах данного кернела передадим указатели на пять буферов данных и один параметр:

- *scores* — буфер матрицы коэффициентов зависимости;
- *scores_temp* — буфер градиентов ошибки на уровне нормализованной матрицы коэффициентов зависимости;
- *scores_grad* — буфер градиентов ошибки на уровне матрицы коэффициентов зависимости, скорректированный на производную функции нормализации;
- *values* — буфер тензора значений *Values* (буфер результатов нейронного слоя *m_cValues*);
- *values_grad* — буфер тензора градиентов ошибки на уровне результатов нейронного слоя *m_cValues*;
- *outputs_grad* — буфер градиентов ошибки на уровне выхода блока *Self-Attention*;
- *window* — размер вектора описания одного элемента последовательности в тензоре значений *Values*.

Обратите внимание, что у буферов *scores_temp* и *scores_grad* нет аналогов на стороне основной программы. Дело в том, что градиенты ошибки на уровне матрицы коэффициентов зависимости нам потребуется только для выполнения операций текущего обратного прохода. Но в *OpenCL* нет возможности создавать динамические массивы. Вместо них мы создали указанные буферы.

```
__kernel void AttentionCalcScoreGradient(__global TYPE *scores,
                                         __global TYPE *scores_grad,
                                         __global TYPE *values,
                                         __global TYPE *values_grad,
                                         __global TYPE *outputs_grad,
                                         __global TYPE *scores_temp,
                                         int window)
  {
```

Алгоритм прямого прохода предусматривает нормализацию матрицы коэффициентов зависимости *Score* функцией *Softmax* в разрезе запросов *Query*. Поэтому после определения градиентов ошибки на уровне матрицы коэффициентов нужно скорректировать полученные значения на производную операции нормализации данных. Следовательно, вполне логично будет разделить операции на потоки в таком же ключе. К тому же такое распределение операций на потоки будет вполне уместно и для распределения градиента ошибки на уровень значений внутреннего нейронного слоя.

В начале кернела мы, как всегда, проведем небольшую подготовительную работу. Определим порядковый номер анализируемых вектора значений и строки матрицы коэффициентов зависимостей по идентификационному номеру потока. Общее количество запущенных потоков подскажет нам размерность тензоров. Сразу же определим смещение в буферах данных до первого элемента анализируемых векторов значений.

```
  const int q = get_global_id(0);
  const int units = get_global_size(0);
  int shift_value = window * q;
  int shift_score = units * q;
```

Далее мы распределим градиент ошибки до уровня внутреннего нейронного слоя *m_cValues.* Как уже было сказано выше, для определения градиента ошибки нам необходимо умножить транспонированную матрицу коэффициентов зависимостей на тензор градиентов на выходе блока *Self-Attention.*

В рамках кернела мы определим градиент ошибки лишь для одного вектора описания элемента. Как вы знаете, при прямом проходе каждый элемент последовательности в тензоре значений *Value* оставляет свой след в формировании всех элементов последовательности результатов блока *Self-Attention.* Следовательно, и каждый элемент тензора *Value* должен получить свою долю градиента ошибки со всех элементов тензора результатов блока *Self-Attention.* Мерой влияния будет соответствующий коэффициент зависимости из матрицы *Score.* Таким образом, каждому элементу последовательности тензора значений *Value* соответствует один столбец в матрице коэффициентов зависимостей *Score.* Этим и объясняется использование транспонированной матрицы *Score* в формуле, приведенной выше.

Для организации этого процесса создадим систему из двух вложенных циклов. Число итераций в первом цикле равно размеру вектора описаний одного элемента последовательности в тензоре *Value.* Надо отметить, что тензор градиентов ошибки на выходе блока *Self-Attention* имеет такие же размеры. Во вложенном цикле с числом итераций равным количеству элементов в последовательности мы будем перебирать значения соответствующего столбца матрицы коэффициентов зависимости *Score* и вектора градиентов ошибки на уровне результатов блока *Self-Attention.* При этом будем перемножать соответствующие элементы, а полученные произведения суммировать в частную переменную. После завершения итераций внутреннего цикла скопируем накопленную сумму произведений в буфер градиентов ошибки внутреннего сверточного слоя *m_cValues.*

```
//--- Распределение градиента на Values
   for(int i = 0; i < window; i ++)
     {
      TYPE grad = 0;
      for(int g = 0; g < units; g++)
         grad += scores[units * g + q] * outputs_grad[window * g + i];
      values_grad[shift_value + i] = grad;
     }
```

После отработки системы циклов можно считать выполненной первую часть нашей задачи — передачу градиентов ошибки во внутренний нейронный слой *m_cValues.*

Вторая часть нашего кернела посвящена определению градиента ошибки на уровне матрицы коэффициентов зависимости.

При прямом проходе каждый элемент последовательности запросов *Query* перемножается со всеми элементами последовательности ключей *Key* для формирования одного вектора матрицы коэффициентов зависимостей *Score*. Каждый такой вектор нормализуется функцией *Softmax.* После этого умножаем его на тензор значений *Value*. В результате этих операций получаем скорректированный вектор описания одного элемента последовательности в тензоре результатов блока *Self-Attention.* Таким образом, один элемент последовательности запросов *Query* взаимодействует со всеми элементами тензоров *Key* и *Value* для формирования вектора описания одного элемента последовательности результатов. Следовательно, для распределения градиента ошибки до отдельно взятого вектора из тензора запросов *Query* нам необходимо взять один соответствующий вектор градиентов ошибки одного элемента последовательности на уровне результатов блока *Self-Attention* и сначала умножить его на транспонированный тензор значений *Value*. Тем самым мы получим вектор ошибки на уровне матрицы коэффициентов зависимости *Score*. Далее нам необходимо скорректировать полученный вектор на производную функции *Softmax.* Именно эту часть распределения градиента ошибки мы реализуем в данном кернеле. Для дальнейшего распределения градиента ошибки до уровня внутренних нейронных слоев *m_cQuerys* и *m_cKeys* немного позже мы создадим еще один кернел.

Описанный выше алгоритм распределения градиента ошибки в матричном виде можно представить следующим образом:

1. Градиент ошибки на уровне матрицы Score.

$$G_{Score} = G_{Self-Attention} * V^T$$

2. Скорректируем градиент ошибки на производную функции *Softmax*.

$$G'_{Score} = S^T(E - S) \circ G_{Score}$$

Обобщим весь расчет в одну формулу:

$$G'_{Score} = S^T(E - S) \circ (G_{Self-Attention}V^T)$$

Вначале перенесем градиент ошибки на уровень матрицы коэффициентов зависимости *Score*. Так как благодаря разделению операций на параллельные потоки в рамках кернела мы будем определять градиент ошибки только одной строки, то и для определения этого вектора градиентов ошибки нам необходимо взять вектор градиентов ошибки для одного элемента последовательности на уровне результатов бока *Self-Attention* и умножить его на транспонированный тензор результатов внутреннего слоя *m_cValues*. Практически же мы будем использовать алгоритм, описанный выше при вычислении градиентов ошибки на слой *m_cValues*. Мы создадим систему из двух вложенных циклов. Но на этот раз количество итераций

внешнего цикла будет равно количеству элементов в последовательности. Вложенный цикл повторит свои операции по числу элементов в векторе описания одного элемента последовательности. Такая разница объясняется величиной вектора результатов и подтверждается логикой производимых операций. Вспомните, ведь при прямом проходе каждый элемент в строке матрицы коэффициентов зависимостей соответствует одному вектору описания элемента последовательности в тензоре значений *Values.*

```
//--- Распределение градиента на Score
   for(int k = 0; k < units; k++)
     {
      TYPE grad = 0;
      for(int i = 0; i < window; i++)
         grad += outputs_grad[shift_value + i] * values[window * k + i];
      scores_temp[shift_score + k] = grad;
     }
```

После переноса градиента ошибки на уровень матрицы коэффициентов зависимостей нам остается скорректировать полученные значения на производную функцию нормализации *Softmax.* Как и при прямом проходе для получения одного нормализованного значения требовалась обработка всего вектора нормализуемых значений, так и для вычисления одного скорректированного значения нам необходимо использование всех элементов обоих векторов (градиентов ошибок на уровне матрицы коэффициентов зависимости и самого нормализованного вектора коэффициентов).

Матричное выражение процесса корректировки на производную функции *Softmax* приведено выше. А для практической реализации мы создадим систему из двух вложенных циклов. Оба цикла имеют одинаковое число итераций, которое равно размеру нормализуемого вектора. В данном случае он равен количеству элементов в последовательности. При проведении операций потребуется накапливать сумму градиентов ошибки с каждого элемента нормализованного вектора. Для этого в теле внешнего цикла мы создадим частную переменную *grad.* Кроме того, чтобы сократить количество обращений к глобальной памяти, сохраним повторяющийся элемент в частную переменную *score.* Напомню, что обращение к глобальной памяти более затратное по времени. Сокращая количество обращений к буферам глобальной памяти, мы сокращаем общие затраты времени на проведение операций. В теле вложенного цикла будем производить операции умножения элементов и складывать полученные произведения в предварительно созданную частную переменную *grad*.

Обратите внимание, что единичную матрицу мы заменили выражением (*int)(i==k).* Логическое выражение даст нам истинное значение только на диагонали матрицы. Перевод логического значение в целочисленное подставит 1 вместо истинных значений и 0 для ложных. Таким образом, столь краткая запись позволяет нам получить значения единичной матрицы прямо в потоке операций без необходимости предварительного ее формирования и сохранения.

```
//--- Корректируем на производную Softmax
   for(int k = 0; k < units; k++)
     {
      TYPE grad = 0;
      TYPE score = scores[shift_score + k];
      for(int i = 0; i < units; i++)
         grad += scores[shift_score + i] *
                     ((int)(i == k) - score) * scores_temp[shift_score + i];
      scores_grad[shift_score + k] = grad;
     }
  }
```

После завершения итераций системы циклов на выходе мы получим градиенты ошибки на уровне матрицы коэффициентов зависимости скорректированные на производную функции *Softmax*.

На этом мы завершаем первый кернел обратного прохода и переходим к созданию второго кернела *AttentionCalcHiddenGradient*, в котором доведем градиент ошибки до внутренних нейронных слоев *m_cQuerys* и *m_cKeys*. Для этого нам потребуется передать в параметрах кернела указатели на пять буферов данных и одну константу:

- *querys* — буфер результатов внутреннего нейронного слоя *m_cQuerys*;
- *querys_grad* — буфер градиентов ошибки внутреннего нейронного слоя *m_cQuerys*;
- *keys* — буфер результатов внутреннего нейронного слоя *m_cKeys*;
- *keys_grad* — буфер градиентов ошибки внутреннего нейронного слоя *m_cKeys*;
- *scores_grad* — буфер градиентов ошибки матрицы коэффициентов зависимостей *m_cScores*;
- *key_size* — размер вектора ключа одного элемента.

```
__kernel void AttentionCalcHiddenGradient(__global TYPE *querys,
                                          __global TYPE *querys_grad,
                                          __global TYPE *keys,
                                          __global TYPE *keys_grad,
                                          __global TYPE *scores_grad,
                                          int key_size)
  {
```

По аналогии со всеми выше рассмотренными кернелами, распределение на протоки операций будем осуществлять в разрезе одного элемента последовательности. В начале кернела проведем подготовительную работу и определим смещение в буферах данных до первого элемента вектора анализируемого элемента.

```
   const int q = get_global_id(0);
   const int units = get_global_size(0);
   int shift_query = key_size * q;
   int shift_score = units * q;
```

В рассмотренном выше кернеле *AttentionCalcScoreGradient* мы уже скорректировали градиент ошибки матрицы коэффициентов зависимости на производную функции нормализации *Softmax*. Но во время прямого прохода перед нормализацией матрицы мы разделили все ее элементы на квадратный корень из размерности вектора ключа. Сейчас нам предстоит корректировать градиент ошибки на производную указанной операции. По аналогии с операцией умножения

нужно будет разделить все значения буфера градиентов ошибки матрицы коэффициентов зависимости на туже константу.

$$\left(\frac{x}{Constant}\right)' = \left(\frac{1}{Constant}x\right)' = \frac{1}{Constant}$$

Определим значение константы и сохраним в частную переменную.

```
//--- Распределение градиента на Querys и Keys
   const TYPE k = 1 / sqrt((TYPE)key_size);
```

На этом подготовительная работа завершена. Можно приступить непосредственно к пересчету градиентов ошибки. Для получения коэффициентов зависимостей мы перемножали два тензора (запросов *Query* и ключей *Key*). С производными операций умножения мы уже не раз сталкивались. Для получения градиентов ошибки на одном из тензоров нам необходимо умножить тензор градиентов ошибки на уровне матрицы коэффициентов зависимостей на второй тензор. А благодаря тому, что тензоры *Query* и *Key* имеют одинаковые размерности, мы можем посчитать градиенты ошибки для обоих тензоров в одной системе циклов.

Создадим систему из двух вложенных циклов. Внешний цикл имеет количество итераций равное размеру вектора ключа одного элемента последовательности. А во вложенном цикле мы будем перебирать векторы противоположного тензора и соответствующие градиенты ошибки матрицы коэффициентов зависимости. Следовательно, число его итераций будет равняться количеству элементов в анализируемой последовательности.

Как вы понимаете, в теле вложенного цикла мы будем умножать соответствующие элементы. Результаты этих произведений нужно будет суммировать. Для накопления этой суммы мы создадим две частные переменные *grad_q* и *grad_k* перед объявлением вложенного цикла.

И еще один момент. Для сокращения количества операций вычисления мы не будем добавлять в произведения вложенного цикла наш ранее посчитанный коэффициент для корректировки градиента ошибки. Воспользуемся математическими свойствами функций и вынесем постоянный множитель за скобки.

$$ka + kb + kc = k(a + b + c)$$

Таким образом, нет необходимости умножать значение каждый раз на поправочный коэффициент в теле вложенного цикла. Вместо этого достаточно один раз умножить на поправочный коэффициент итоговую сумму перед записью в буфер данных.

```
   for(int i = 0; i < key_size; i++)
     {
      TYPE grad_q = 0;
      TYPE grad_k = 0;
      for(int s = 0; s < units; s++)
        {
         grad_q += keys[key_size * s + i] * scores_grad[shift_score + s];
         grad_k += querys[key_size * s + i] * scores_grad[units * s + q];
        }
      querys_grad[shift_query + i] = grad_q * k;
      keys_grad[shift_query + i] = grad_k * k;
     }
  }
```

На выходе из системы циклов мы получаем градиенты ошибок для двух вложенных внутренних нейронных слоев *m_cQuerys* и *m_cKeys*. То есть задача данного кернела решена. А с учетом выше рассмотренного кернела *AttentionCalcScoreGradient*, мы распределили градиент ошибки на все внутренние нейронные слои и дальнейшее распределение градиента ошибки до предыдущего слоя будет осуществляться уже протестированными методами внутренних нейронных слоев, как это реализовано стандартными средствами *MQL5*.

Рассмотренные выше кернелы обратного прохода обошли процессы сложения буферов результатов и нормализации данных, которые мы осуществляли при прямом проходе. И если производная двух функций будет равна сумме производных этих функций, то для операции сложения градиентов мы можем воспользоваться аналогичным кернелом прямого прохода. Надо лишь указать правильные буферы данных.

$$(f(x)+g(x))' = (f(x))' + (g(x))'$$

В случае же корректировки градиента ошибки на функцию нормализации данных нам предстоит создать дополнительный кернел. Напомню формулы корректировки градиента ошибки.

$$\frac{\partial f(a_i)}{\partial \sigma^2} = \sum_{i=1}^{m} \frac{\partial f(a_i)}{\partial \hat{a}_i} * (a_i - \bar{a}) * \frac{-1}{2}(\sigma^2 + \varepsilon)^{-\frac{3}{2}}$$

$$\frac{\partial f(a_i)}{\partial \bar{a}} = \left(\sum_{i=1}^{m} \frac{\partial f(a_i)}{\partial \hat{a}_i} * \frac{-1}{\sqrt{\sigma^2 + \varepsilon}}\right) + \left(\frac{\partial f(a_i)}{\partial \sigma^2} * \frac{-2\sum_{i=1}^{m}(a_i - \bar{a})}{m}\right)$$

$$\frac{\partial f(a_i)}{\partial a_i} = \frac{\partial f(a_i)}{\partial \hat{a}_i} * \frac{1}{\sqrt{\sigma^2 + \varepsilon}} + \frac{\partial f(a_i)}{\partial \sigma^2} * \frac{2(a_i - \bar{a})}{m} + \frac{\partial f(a_i)}{\partial \bar{a}} * \frac{1}{m}$$

Как можно заметить, в приведенных выше формулах при вычислении производных средних используется сумма значений по всему буферу значений. Но в отличие от прямого прохода, у нас есть возможность все три суммы вычислять параллельно.

В параметрах кернела мы передаем указатели на четыре буфера данных:

- *outputs* — буфер результатов нормализации прямого прохода;
- *out_gradient* — буфер градиентов на выходе блока нормализации;

- *inp_gradient* — буфер для записи скорректированных градиентов;
- *stds* — буфер среднеквадратических отклонений, вычисленных при прямом проходе.

Также в параметрах укажем размер буферов и смещение в буфере среднеквадратических отклонений.

```
__kernel void LayerNormalizeGradient(__global TYPE* outputs,
                                     __global TYPE* out_gradient,
                                     __global TYPE* inp_gradient,
                                     __global TYPE* stds,
                                     const int total,
                                     const int std_shift)
{
 uint i = (uint)get_global_id(0);
 uint l = (uint)get_local_id(0);
```

В теле кернела определяем идентификаторы потока и сразу объявляем локальные массивы данных. Их будет три. В одном будем собирать производную среднеквадратического отклонения, два других предназначены для слагаемых формулы производной среднего арифметического.

```
 uint ls = min((uint)get_local_size(0), (uint)LOCAL_SIZE);
 __local TYPE dSTD[LOCAL_SIZE];
 __local TYPE dMean1[LOCAL_SIZE];
 __local TYPE dMean2[LOCAL_SIZE];
```

Как и при прямом проходе, сначала каждый поток соберет свою часть от общей суммы.

```
 uint count = 0;
 do
   {
    uint shift = count * ls + l;
    dSTD[l] = (count > 0 ? dSTD[l] : 0) -
              (shift < total ? out_gradient[shift] * outputs[shift] /
              (2 * (pow(stds[std_shift], (TYPE)2) + 1e-37f)) : 0);
    dMean1[l] = (count > 0 ? dMean1[l] : 0) -
              (shift < total ? out_gradient[shift] /
              (stds[std_shift] + 1e-37f) : 0);
    dMean2[l] = (count > 0 ? dMean2[l] : 0) -
                (shift < total ? 2 * outputs[shift] * stds[std_shift] /
                (TYPE)total : 0);
    count++;
   }
 while((count * ls + l) < total);
 barrier(CLK_LOCAL_MEM_FENCE);
```

В следующем цикле соберем сумму в первых элементах массива.

```
    count = ls;
    do
      {
       count = (count + 1) / 2;
       dSTD[l] += (l < count ? dSTD[l + count] : 0);
       dMean1[l] += (l < count ? dMean1[l + count] : 0);
       dMean2[l] += (l < count ? dMean2[l + count] : 0);
       barrier(CLK_LOCAL_MEM_FENCE);
      }
    while(count > 1);
  //---
    TYPE dstd = dSTD[0];
    TYPE dmean = dMean1[0] + dstd * dMean2[0];
```

Полученные значения перенесем в частные переменные. При вычислении производной среднеарифметического отклонения значение правого слагаемого умножим на производную среднеквадратического отклонения и сложим с левым слагаемым.

На данном этапе у нас достаточно данных для корректировки градиента ошибки по каждому элементу буфера. Организуем еще один цикл, в теле которого и будет выполнена эта работа.

```
  //---
    count = 0;
    while((count * ls + l) < total)
      {
       uint shift = count * ls + l;
       inp_gradient[shift] = out_gradient[shift] / (stds[std_shift] + 1e-32f) +
                 (2 * dstd * outputs[shift] * stds[std_shift]  + dmean) / total;
       count++;
      }
   }
```

На этом заканчиваем работу с программой *OpenCL*. Теперь нам предстоит сделать вторую часть и организовать подготовительную работу для запуска многопоточных вычислений на стороне основной программы.

Для начала добавим константы для работы с кернелами в файл *defines.mqh*. Нам нужно добавить константы идентификации самих кернелов и их переменных. Для именования констант используем ранее оговоренные правила, которые применяются ко всем константам в рамках нашего проекта:

- все константы начинаются с префикса *def*;
- кернелы начинаются с префикса *def_k*;
- константы параметров после префикса *def* содержат указатель на кернел.

```
#define def_k_AttentionFeedForward     28
#define def_k_AttentionScoreGradients  29
#define def_k_AttentionHiddenGradients 30
#define def_k_Sum                      31
#define def_k_LayerNormalize           32
#define def_k_LayerNormalizeGradient   33

//--- прямой проход блока внимания
#define def_attff_querys               0
#define def_attff_keys                 1
#define def_attff_scores               2
#define def_attff_values               3
#define def_attff_outputs              4
#define def_attff_window               5
#define def_attff_key_size             6

//--- определение градиента на матрице коэффициентов зависимости блока внимания
#define def_attscr_scores              0
#define def_attscr_scores_grad         1
#define def_attscr_values              2
#define def_attscr_values_grad         3
#define def_attscr_outputs_grad        4
#define def_attscr_scores_temp         5
#define def_attscr_window              6

//--- распределение градиента через блок внимания
#define def_atthgr_querys              0
#define def_atthgr_querys_grad         1
#define def_atthgr_keys                2
#define def_atthgr_keys_grad           3
#define def_atthgr_scores_grad         4
#define def_atthgr_key_size            5

//--- сумма векторов
#define def_sum_inputs1                0
#define def_sum_inputs2                1
#define def_sum_outputs                2
```

```
//--- нормализация вектора
#define def_layernorm_inputs          0
#define def_layernorm_outputs         1
#define def_layernorm_std             2
#define def_layernorm_vector_size     3
#define def_layernorm_std_shift       4

//--- градиент нормализации вектора
#define def_layernormgr_outputs       0
#define def_layernormgr_out_grad      1
#define def_layernormgr_inp_grad      2
#define def_layernormgr_std           3
#define def_layernormgr_vector_size   4
#define def_layernormgr_std_shift     5
```

После этого нам предстоит добавить объявление новых кернелов в код основного класса-диспетчера нейронной сети. Как и всех ранее созданных кернелов, объявление новых кернелов мы добавим в метод *CNet::InitOpenCL*. В нем мы сначала изменим общее количество используемых в программе кернелов.

```
if(!m_cOpenCL.SetKernelsCount(34))
  {
   m_cOpenCL.Shutdown();
   delete m_cOpenCL;
   return false;
  }
```

После этого объявим и сами кернелы.

```
   if(!m_cOpenCL.KernelCreate(def_k_AttentionFeedForward,
                                      "AttentionFeedForward"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }

   if(!m_cOpenCL.KernelCreate(def_k_AttentionScoreGradients,
                                      "AttentionCalcScoreGradient"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }

   if(!m_cOpenCL.KernelCreate(def_k_AttentionHiddenGradients,
                                      "AttentionCalcHiddenGradient"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }

   if(!m_cOpenCL.KernelCreate(def_k_Sum, "Sum"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }

   if(!m_cOpenCL.KernelCreate(def_k_LayerNormalize, "LayerNormalize"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }

   if(!m_cOpenCL.KernelCreate(def_k_LayerNormalizeGradient
                                         "LayerNormalizeGradient"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
```

Затем перейдем к классу механизма внимания *CNeuronAttention* и внесем изменения в его методы в части работы с технологией *OpenCL*.

Первым дополним метод прямого прохода *CNeuronAttention::FeedForward*. В этом методе нам предстоит организовать процедуру вызова кернела прямого прохода *AttentionFeedForward*. Мы уже не раз создавали аналогичные процессы. Но все же напомню алгоритм:

1. Проверка наличия буферов данных в контексте OpenCL;

2. Передача параметров кернелу, в том числе указатели на буферы данных;
3. Постановка кернела в очередь выполнения операций.

При этом не забываем контролировать выполнение операций, чтобы исключить возможные критические ошибки в процессе дальнейшего выполнения программы.

```
bool CNeuronAttention::FeedForward(CNeuronBase *prevLayer)
  {
//--- вычисление векторов Query, Key, Value
   .....
//--- Разветвление алгоритма по вычислительному устройству
   MATRIX out;
   if(!m_cOpenCL)
     {
   // Блок MQL5
   .....
     }
   else // Блок OpenCL
     {
      //--- проверка буферов данных
      if(m_cQuerys.GetOutputs().GetIndex() < 0)
         return false;
      if(m_cKeys.GetOutputs().GetIndex() < 0)
         return false;
      if(m_cValues.GetOutputs().GetIndex() < 0)
         return false;
      if(m_cScores.GetIndex() < 0)
         return false;
      if(m_cAttentionOut.GetOutputs().GetIndex() < 0)
         return false;
```

При наличии всех необходимых буферов в контексте *OpenCL* мы организуем передачу указателей на них в параметры кернела.

```
   //--- передача параметров кернелу
   if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward, def_attff_keys,
                                                   m_cKeys.GetOutputs().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward, def_attff_outputs,
                                          m_cAttentionOut.GetOutputs().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward, def_attff_querys,
                                                  m_cQuerys.GetOutputs().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward, def_attff_scores,
                                                        m_cScores.GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward, def_attff_values,
                                                  m_cValues.GetOutputs().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgument(def_k_AttentionFeedForward, def_attff_key_size,
                                                              m_iKeysSize))
      return false;
   if(!m_cOpenCL.SetArgument(def_k_AttentionFeedForward, def_attff_window,
                                                                m_iWindow))
      return false;
```

Далее идет процедура постановки кернела в очередь выполнения. Для начала укажем количество необходимых потоков для запуска и смещение. Только потом вызовем функцию запуска кернела с передачей в нее информации о количестве запускаемых копий.

```
   //--- постановка кернела в очередь выполнения
   int off_set[] = {0};
   int NDRange[] = {m_iUnits};
   if(!m_cOpenCL.Execute(def_k_AttentionFeedForward, 1, off_set, NDRange))
      return false;
  }
```

На этом мы выполнили алгоритм запуска кернела блока *Self-Attention*. Но нам еще предстоит сложить содержимое двух буферов и нормализовать данные буфера результатов. Следуя алгоритму, сначала найдем сумму двух векторов (исходных данных и результатов *Self-Attention*). Эта операция довольно общая и может широко использоваться за пределами нашего класса нейронного слоя внимания *CNeuronAttention*. Поэтому было принято решение добавить ее отдельным методом в класс буфера данных *CBufferType::SumArray*.

В параметрах методу *SumArray* будем передавать указатель на добавляемый буфер. Сразу в теле метода проверим полученный указатель и размер полученного буфера. Для успешного выполнения операции размер текущего буфера, который будет выступать первым слагаемым, и полученного буфера (второго слагаемого) должны быть равны.

```
bool CBufferType::SumArray(CBufferType *src)
  {
//--- проверка массива исходных данных
   if(!src || src.Total() != Total())
      return false;
```

Как и все ранее рассмотренные методы, алгоритм данного метода разделяется на два потока в зависимости от устройства выполнения операций. В блоке выполнения операций средствами *MQL5* мы сначала приведем в соответствие форматы матриц обоих буферов. Затем выполним операцию матричного сложения. Результат операции сохраним в матрице текущего буфера.

```
   if(!m_cOpenCL)
     {
      //--- изменение размера матрицы
      MATRIX temp = src.m_mMatrix;
      if(!temp.Reshape(Rows(), Cols()))
         return false;
      //--- сложения матриц
      m_mMatrix += temp;
     }
```

Алгоритм блока многопоточных операций аналогичен рассмотренному выше. Мы сначала проверяем наличие данных в контексте *OpenCL* и при необходимости загружаем данные полученного буфера. Обратите внимание, что мы проверяем только полученный буфер. Выше, при разделение алгоритма в зависимости от вычислительного устройства, мы уже проверили указатель на контекст *OpenCL* текущего буфера. Следовательно, считаем данные текущего буфера уже переданными в контекст *OpenCL*.

За блоком контролей следует передача параметров кернелу и постановка его в очередь выполнения.

```
   else
     {
      if(src.GetIndex() < 0 && !BufferCreate(m_cOpenCL))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_Sum, def_sum_inputs1, m_myIndex))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_Sum, def_sum_inputs2, src.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_Sum, def_sum_outputs, m_myIndex))
         return false;
      uint off_set[] = {0};
      uint NDRange[] = {(uint)Total()};
      if(!m_cOpenCL.Execute(def_k_Sum, 1, off_set, NDRange))
         return false;
     }
//---
   return true;
  }
```

Процесс нормализации данных организован в методе *CNeuronAttention::NormlizeBuffer*. Но при соблюдении общих правил построения алгоритма в данном методе есть два исключения. Во-первых, мы исключили блок проверки наличия буферов в контексте *OpenCL*. В данном случае

риск использование незагруженных буферов минимален. Дело в том, что до вызова данного метода используемые буферы были уже проверены не один раз, и повторная проверка будет лишней.

```
bool CNeuronAttention::NormlizeBuffer(CBufferType *buffer,
                                      CBufferType *std,
                                      uint std_shift)
  {
   if(!m_cOpenCL)
     {
    // Блок MQL5
   .....
     }
   else
     {
      if(!m_cOpenCL.SetArgumentBuffer(def_k_LayerNormalize,
                                      def_layernorm_inputs, buffer.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_LayerNormalize,
                                     def_layernorm_outputs, buffer.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_LayerNormalize,
                                           def_layernorm_std, std.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_LayerNormalize,
                                def_layernorm_vector_size, (int)buffer.Total()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_LayerNormalize,
                                          def_layernorm_std_shift, std_shift))
         return false;
```

Второй момент связан с использованием локального массива данных и синхронизацией потоков. Дело в том, что синхронизация потоков доступна только в пределах work group. Нам нужно явно указать ее размер. Алгоритм кернела нормализации построен таким образом, что размер рабочей группы не может быть больше размера локального массива. Напомню, размер локального массива определяется константой *LOCAL_SIZE*. В то же время количество потоков не может быть больше размера нормализуемого буфера. Следовательно, в массиве указания размерности пространства задач мы укажем меньшее из двух величин. Так как мы нормализуем значения всего буфера одним пакетом, то и размерность глобального и локального пространства задач будет одна.

Определившись с размерностью задач, мы ставим кернел в очередь выполнения.

```
      int NDRange[] = {(int)MathMin(buffer.Total(), LOCAL_SIZE)};
      int off_set[] = {0};
      if(!m_cOpenCL.Execute(def_k_LayerNormalize, 1, off_set, NDRange, NDRange))
         return false;
     }
//---
   return true;
  }
```

На этом завершается блок использования технологии *OpenCL* в методе прямого прохода нашего класса механизма внимания, и мы завершаем работу над этим методом. Далее его код остается

без изменений. Полностью код приведен в разделе описания построения метода стандартными средствами *MQL5*.

Мы переходим к работе над одним из методов обратного прохода — методом распределения градиента ошибки через скрытый слой *CNeuronAttention::CalcHiddenGradient*. Алгоритм наших действий остается прежний. Сделаем лишь поправку на использование двух кернелов последовательно.

Напомню, что при создании кернелов обратного прохода мы определили необходимость использования двух дополнительных буферов для записи промежуточных значений градиентов ошибки матрицы коэффициентов зависимости. Поэтому сделаем шаг назад и объявим дополнительные буферы: *m_cScoreGrad* и *m_cScoreTemp*.

```
class CNeuronAttention    :  public CNeuronBase
  {
protected:
   .....
   int               m_cScoreGrad;
   int               m_cScoreTemp;
   .....
  };
```

Только в данном случае мы не будем объявлять экземпляры объектов буфера в основной памяти. Мы не будем использовать указанные буферы для обмена данными между контекстом *OpenCL* и основной программой. Они нужны лишь для временного хранения данных их передачи между кернелами. А значит, нам достаточно их наличие в памяти контекста *OpenCL.* В основной программе мы лишь объявим переменные для хранения указателей на буферы.

Но вернемся к работе над методом *CNeuronAttention::CalcHiddenGradient*. Вначале проверяем наличие и при необходимости создаем новые буферы данных в контексте *OpenCL,* используемые в первом кернеле. Мы намеренно не создаем сразу буферы данных для второго кернела, чтобы использование памяти было более рационально. Это позволит нам использовать буферы данных большего размера при ограниченных ресурсах памяти контекста *OpenCL.*

```
bool CNeuronAttention::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
   .....
//--- разветвление алгоритма по вычислительному устройству
   if(!m_cOpenCL)
     {
   // Блок MQL5
   .....
     }
   else // блок OpenCL
     {
      //--- проверка буферов данных
      if(m_cValues.GetOutputs().GetIndex() < 0)
         return false;
      if(m_cValues.GetGradients().GetIndex() < 0)
         return false;
      if(m_cScores.GetIndex() < 0)
         return false;
      if(m_cAttentionOut.GetGradients().GetIndex() < 0)
         return false;
      if(m_cScoreGrad < 0)
         return false;
      if(m_cScoreTemp < 0)
         return false;
```

После проверки всех необходимых буферов передадим указатели на них в кернел.

```
      //--- передача параметров кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
              def_attscr_outputs_grad, m_cAttentionOut.GetGradients().GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
                                      def_attscr_scores, m_cScores.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
                                        def_attscr_scores_grad, m_cScoreGrad))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
                                        def_attscr_scores_temp, m_cScoreTemp))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
                           def_attscr_values, m_cValues.GetOutputs().GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
                    def_attscr_values_grad, m_cValues.GetGradients().GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AttentionScoreGradients,
                                                def_attscr_window, m_iWindow))
         return false;
```

В дополнение к указателям на буферы данных передадим в кернел размер вектора описания одного элемента последовательности.

После передачи всех параметров, укажем количество необходимых параллельных потоков и вызовем функцию постановки кернела в очередь.

```
//--- Постановка кернела в очередь выполнения
int off_set[] = {0};
int NDRange[] = {m_iUnits};
if(!m_cOpenCL.Execute(def_k_AttentionScoreGradients, 1, off_set, NDRange))
   return false;
```

Переходим к работе над следующим кернелом. Проверим наличие буферов, необходимых для нового кернела.

```
if(m_cQuerys.GetOutputs().GetIndex() < 0)
   return false;
if(m_cQuerys.GetGradients().GetIndex() < 0)
   return false;
if(m_cKeys.GetOutputs().GetIndex() < 0)
   return false;
if(m_cKeys.GetGradients().GetIndex() < 0)
   return false;
```

После проверки всех необходимых буферов данных передадим указатели на них в кернел.

```
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                           def_atthgr_keys, m_cKeys.GetOutputs().GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                     def_atthgr_keys_grad, m_cKeys.GetGradients().GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                        def_atthgr_querys, m_cQuerys.GetOutputs().GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                  def_atthgr_querys_grad, m_cQuerys.GetGradients().GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                                       def_atthgr_scores_grad, m_cScoreGrad))
   return false;
if(!m_cOpenCL.SetArgument(def_k_AttentionHiddenGradients,
                                           def_atthgr_key_size, m_iKeysSize))
   return false;
```

В дополнение к указателям на буферы данных в параметры кернела передадим размер вектора ключа одного элемента последовательности.

После завершения передачи всех необходимых данных в кернел мы инициализируем постановку его в очередь исполнения. Массивы с указанием смещения и количества необходимых копий кернела для запуска уже готовы после запуска предыдущего кернела и нам нет необходимости их задавать повторно. Поэтому мы просто вызываем функцию постановки кернела в очередь.

```
   if(!m_cOpenCL.Execute(def_k_AttentionHiddenGradients, 1, off_set, NDRange))
     return false;
```

На этом мы заканчиваем работу над построением методов нашего класса внимания и можем перейти к тестированию его работы.

5.1.4 Тестирование механизма внимания

В отличие от ранее рассмотренного рекуррентного блока *LSTM*, блок внимания работает только с текущими данными. Поэтому для создания более репрезентативной выборки между обновлением весовых коэффициентов в процессе обучения нейронной сети мы будем использовать случайные паттерны из генеральной обучающей выборки. Именно такой подход мы использовали при тестировании полносвязного перцептрона и сверточной модели. В такой ситуации будет вполне логично взять скрипт тестирования сверточной модели convolution_test.mq5, пересохранить его с новым именем *attention_test.mq5* и внести изменения в части описания создаваемой модели.

Для создания тестового скрипта потребовалось сделать не так уж много изменений. Мы удалили из скрипта блоки описания сверточного и подвыборочного слоев. Вместо них сразу после исходных данных мы добавим описание нашего блока внимания. Для этого, как и при описании любого другого нейронного слоя, сначала мы создадим новый экземпляр класса описания нейронного слоя *CLayerDescription*. Сразу же проверим результат выполнения операции по полученному указателю на объект. Затем нам предстоит дать описания создаваемого нейронного слоя.

В поле *type* мы передадим константу *defNeuronAttention*, которая соответствует создаваемому блоку внимания.

В поле *count* мы должны указать количество анализируемых элементов последовательности. Его мы запрашиваем у пользователя при запуске скрипта и сохраняем в переменную *BarsToLine*. Следовательно, в описание нейронного слоя мы можем передать значение переменной.

Следующий далее параметр *window* при описании сверточного слоя мы использовали для указания размера окна исходных данных. Здесь же мы его будем использовать для указания размера вектора описания одного элемента последовательности исходных данных. Согласитесь, за немного разной формулировкой кроется схожий функционал. Но в отличие от сверточного слоя, мы не будем указывать шаг окна, так как в данном случае оно будет равно самому окну. Количество нейронов для описания одной свечи мы так же запрашиваем у пользователя в параметрах скрипта. Это значение сохраняется в переменной *NeuronsToBar*. Как и в случае с предыдущим полем, просто передаем в указанное поле значение из переменной.

Алгоритм *Self-Attention* не предусматривает изменение размера данных. На выходе из блока мы получаем тензор того же размера, что и исходных данных. Получается, поле *window_out* в описании нейронного слоя останется невостребованным. Но мы воспользуемся им, чтобы указать размер вектора ключа одного элемента в тензоре *Key*. На практике размер ключа не всегда отличается от размера вектора описания одного элемента. Понижение размерности используется при большом размере вектора описания одного элемента для экономии вычислительных ресурсов при расчете матрицы коэффициентов зависимости. В нашем случае, когда вектор описания одной свечи составляет всего четыре элемента, мы не будем понижать размерность и передадим в поле *window_out* значение переменной *NeuronsToBar.*

Дополнительно укажем метод оптимизации и его параметры. В тестовом примере я использовал метод *Adam*, как и во всех предыдущих тестах.

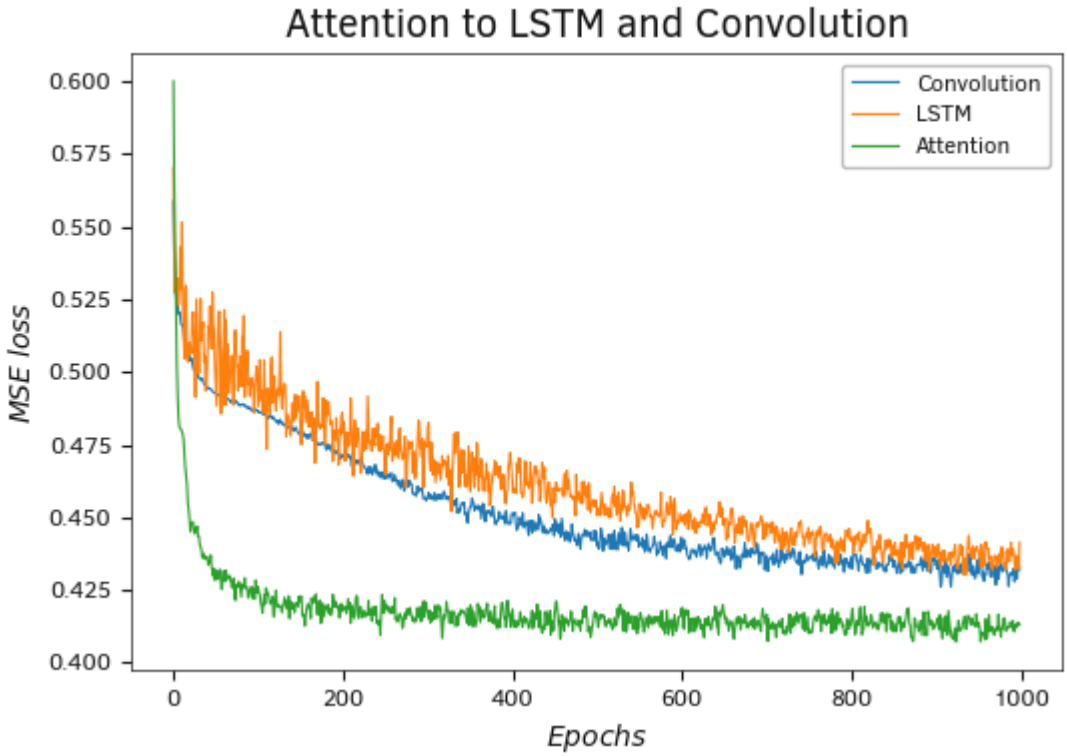
```
bool CreateLayersDesc(CArrayObj &layers)
  {
   CLayerDescription *descr;
//--- создаем слой исходных данных
   .....
//--- слой внимания
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   descr.type = defNeuronAttention;
   descr.count = BarsToLine;
   descr.window = NeuronsToBar;
   descr.window_out = NeuronsToBar;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
//--- скрытый слой
   .....
//---  Слой результатов
   .....
//---
   return true;
  }
```

После указания всех параметров мы добавляем объект в динамический массив описаний нейронных слоев. И, конечно, проверяем результат проведения операций. Далее код скрипта остался без изменения.

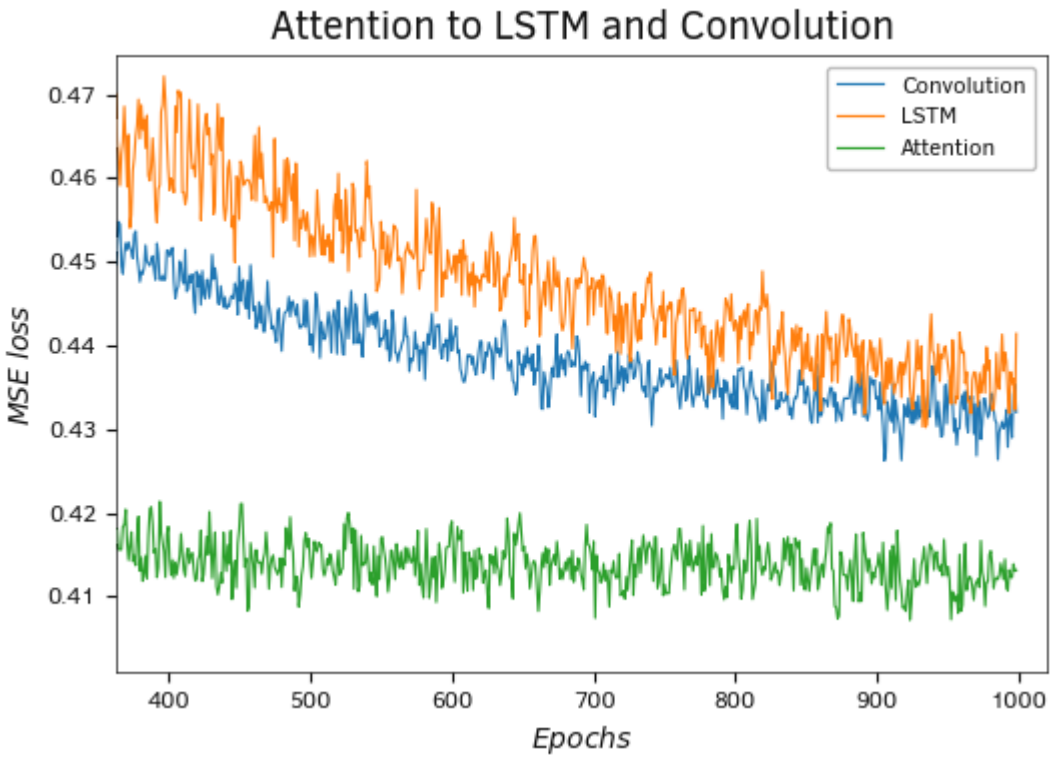
Как видите, при использовании нашей библиотеки изменение конфигурации модели — не очень сложная процедура. Таким образом, вы всегда сможете настроить и провести тестирование различных архитектурных решений для решения конкретной задачи без внесения изменений в логику основной программы.

Тестирование новой модели с использованием блока внимания мы проводили с сохранением всех прочих условий проверки предыдущих моделей. Такой подход позволяет максимально точно оценить влияние изменения архитектуры модели на результат обучения.

Первое же тестирование показало превосходство модели с механизмом внимания над ранее рассмотренными моделями. На графике процесса обучения модель с использованием одного слоя внимания демонстрирует более быструю сходимость по сравнению с моделями, использующими сверточный слой и рекуррентный блок *LSTM*.



Тестирование модели с использованием блока внимания



Тестирование модели с использованием блока внимания

При увеличении масштаба графика обучения можно заметить, что модель с использованием метода внимания демонстрирует меньшую ошибку на протяжении всего процесса обучения.

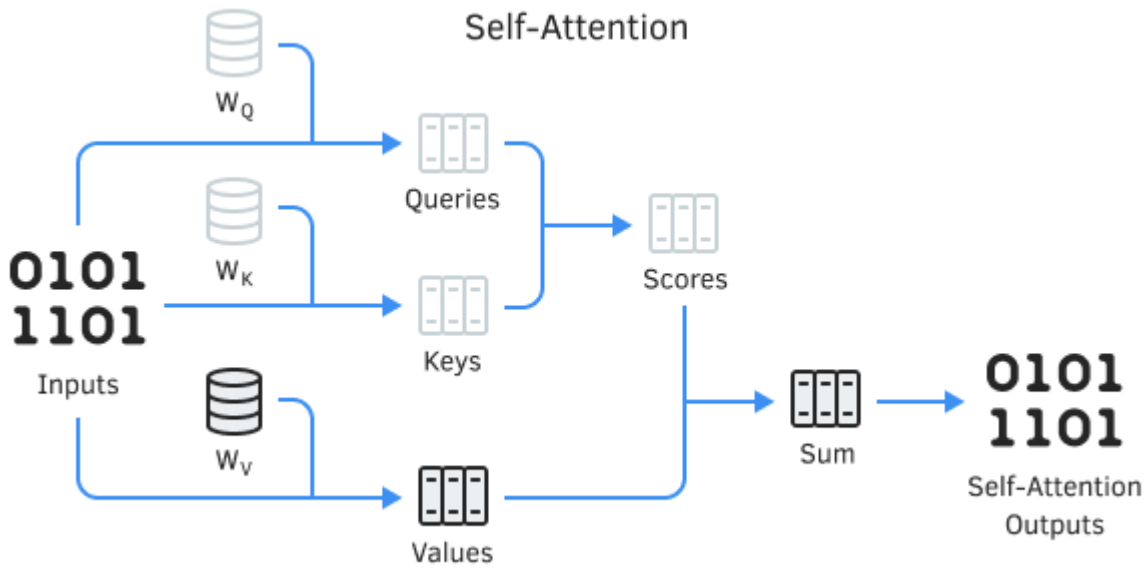
В то же время надо сказать, что использование блока внимания в подобном виде на практике практически не встречается. Наиболее широкое распространение получила архитектура многоголового внимания, которую мы рассмотрим в следующем разделе.

5.2 Многоголовое внимание

В предыдущем разделе мы познакомились с механизмом *Self-Attention,* который был представлен в июне 2017 года в статье Attention Is All You Need. Особенностью данного механизма является способность выявлять зависимости между отдельными элементами последовательности. Мы даже реализовали его и успели протестировать на реальных данных. Модель показала свою работоспособность.

Напомню, что в алгоритме *Self-Attention* используется три обучаемых матрицы весовых коэффициентов (W_Q, W_K и W_V). Данные матрицы используются для получения 3трех сущностей: *Query* (Запрос), *Key* (Ключ) и *Value* (Значение). Первые две определяют попарную взаимосвязь между элементами последовательности, а последняя — контекст анализируемого элемента.

Не секрет, что далеко не всегда ситуации бывают однозначны. Наверное, даже чаще одну и ту же ситуацию можно трактовать с различных точек зрения. При разных точках зрения выводы могут быть абсолютно противоположными. В таких ситуациях важно рассмотреть все возможные варианты, и только после тщательного анализа сделать вывод. Поэтому в той же статье авторы метода для решения таких задач предложили использовать многоголовое внимание *Multi-Head Attention*. Это запуск нескольких параллельных потоков Self-Attention с различными весовыми коэффициентами. Здесь каждая «голова» имеет свое мнение, а решение принимается взвешенным голосованием. Подобное решение должно лучше выявлять связи между различными элементами последовательности.



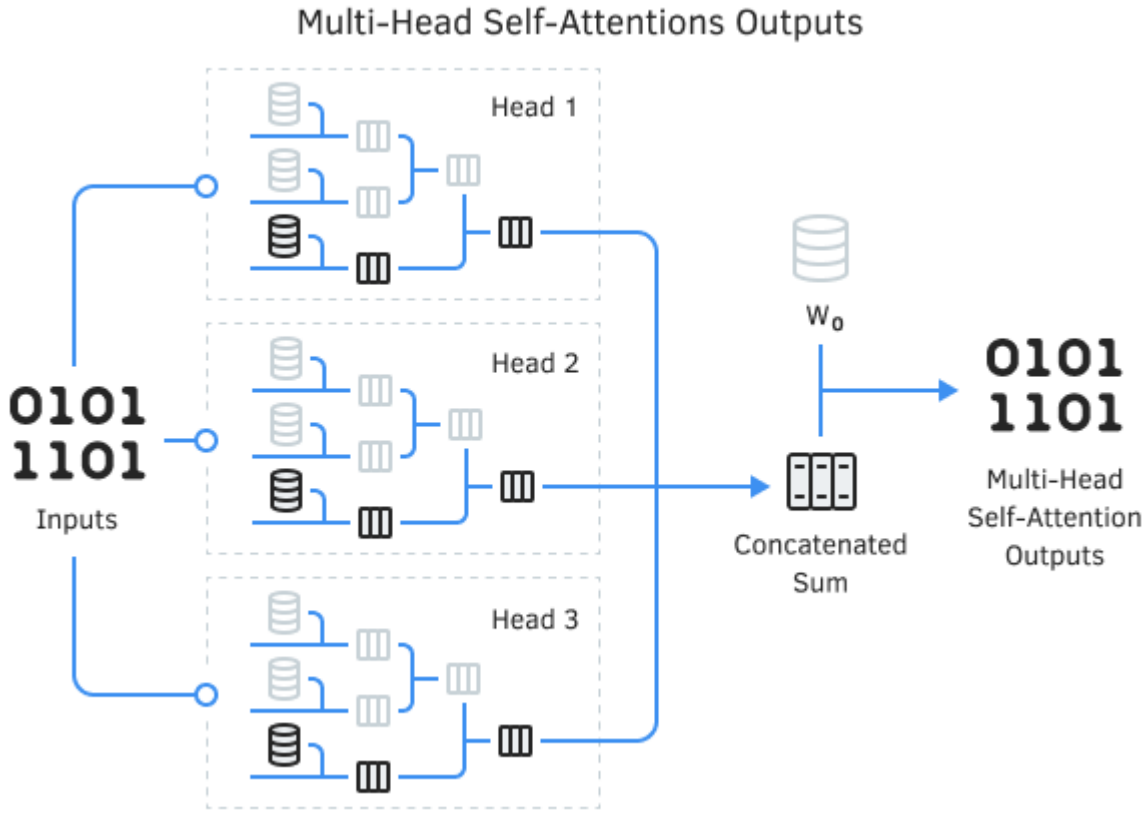


Схема архитектуры многоголового внимания

В архитектуре *Multi-Head Attention* параллельно используется несколько потоков *Self-Attention* с различными весовыми коэффициентами, что имитирует разносторонний разбор ситуации. Результаты работы потоков конкатенируются в единый тензор. Итоговый результат алгоритма определяется умножением тензора на матрицу W_0, параметры которой подбираются в процессе обучения нейронной сети. Вся эта архитектура подменяет блок *Self-Attention* в энкодере и декодере архитектуры «Трансформер».

Именно архитектура *Multi-Head Attention* наиболее часто используется для решения практических задач.

5.2.1 Описание архитектуры *Multi-Head Self-Attention*

Рассмотренная ранее технология *Self-Attention* выявляет зависимости между объектами последовательности в некоем контексте и потом ранжирует их с использованием функции *Softmax*. Но в решении практических задач далеко не всегда можно однозначно дать такую оценку. Как правило, коэффициенты зависимости между объектами очень сильно меняются при изменении точки зрения или контекста анализируемого элемента. Окончательное решение о зависимости элементов — это всегда компромисс. Именно использование *Multi-Head Self-Attention* призвано помочь находить зависимости между элементами со всесторонним рассмотрением исходных данных. А вводимая дополнительная обучаемая матрица весовых коэффициентов поможет модели научиться находить этот компромисс.

Наверное, самым простым вариантом решения подобной задачи будет взять наш класс внимания *CNeuronAttention* и дополнить его массивом блоков *Self-Attention.* Такой подход возможен, но он иррациональный. Он ведет к увеличению числа объектов пропорционально увеличению количества голов внимания. Кроме того, последовательный вызов операций каждой головы внимания не дает нам возможности для организации одновременного параллельного вычисления внимания всех голов. Кроме того, последующая операция конкатенации результатов работы голов внимания тоже потребует затрат ресурсов и времени.

Но выход все же есть, и он лежит в области математики матричных операций. Надо сказать, что именно знание и понимание математики матричных операций очень сильно помогает в осознании математики нейронных сетей и дает четкую картину о возможности разделения операций на параллельные потоки.

Давайте пройдем по алгоритму *Self-Attention* и подумаем о трансформации операций для реализации многоголового внимания *Multi-Head Self-Attention.*

1. Вначале вычисляем векторы *Query* (запрос), *Key* (ключ) и *Value* (значение). Указанные векторы получаются путем умножения каждого элемента исходной последовательности на соответствующую матрицу W_Q, W_K и W_V.

$$Q = IW_Q,\ K = IW_K,\ V = IW_V$$

Для организации *Multi-Head Self-Attention* нам необходимо повторить данную операцию по количеству голов внимания. Чтобы сильно не усложнять, возьмем, к примеру, три головы внимания.

$$Q_1 = IW_{Q1},\ K_1 = IW_{K1},\ V_1 = IW_{V1}$$
$$Q_2 = IW_{Q2},\ K_2 = IW_{K2},\ V_2 = IW_{V2}$$
$$Q_3 = IW_{Q3},\ K_3 = IW_{K3},\ V_3 = IW_{V3}$$

Думаю, здесь все понятно и не вызывает вопросов.

Теперь посмотрим на размерности тензоров. Напомню, архитектурой модели предусмотрено одинаковое число элементов последовательности на всех этапах. Каждый элемент последовательности описывается неким вектором значений. А так как механизм *Self-Attention* одинаково применяется к каждому элементу последовательности, то для примера мы можем разобрать операции только с вектором описания одного элемента. При этом размер этого вектора одинаковый для тензоров исходных данных и значений. Но он может отличаться от размерности вектора описания одного элемента последовательности в тензорах запросов и ключей. Давайте обозначим через n_I размер вектора исходных данных и через n_K размер вектора ключей. Тогда тензоры будут иметь нижеследующие размеры.

Тензор	I	Q	W_Q	K	W_K	V	W_V
Размер	n_I	n_K	$n_I * n_K$	n_K	$n_I * n_K$	n_I	$n_I * n_I$

Указанные размеры тензоров справедливы для всех голов внимания. А давайте попробуем объединить соответствующие матрицы весовых коэффициентов в одну большую.

$$Concatenate(W_{Q1}, W_{Q2}, W_{Q3}) = W_{QC}$$
$$Concatenate(W_{K1}, W_{K2}, W_{K3}) = W_{KC}$$
$$Concatenate(W_{V1}, W_{V2}, W_{V3}) = W_{VC}$$

Такие матрицы весовых коэффициентов будут иметь размер $n_I * 3n_K$ для матриц запросов W_{QC} и W_{KC}. Матрица W_{VC} получит размер $n_I * 3n_I$, где 3 — количество голов внимания.

Подставим конкатенированные матрицы в формулы определения векторов.

$$Q_C = IW_{QC},\ K_C = IW_{KC},\ V_C = IW_{VC}$$

По правилам умножения матриц получим следующие размеры тензоров.

Тензор	I	Q_C	W_{QC}	K_C	W_{KC}	V_C	W_{VC}
Размер	n_I	$3n_K$	$n_I * 3n_K$	$3n_K$	$n_I * 3n_K$	$3n_I$	$n_I * 3n_I$

Сравните размеры тензоров в двух таблицах — они очень похожи. Единственное их отличие в умножении на количество голов внимания. Какую это имеет для нас практическую ценность? Тут все очень просто. Вместо создания нескольких экземпляров объектов для каждой головы внимания, мы можем создать всего по одному объекту для расчета каждой сущности. Как и при организации аналогичного процесса в механизме *Self-Attention*, мы можем воспользоваться нашими сверточными слоями, только размер окна результатов нужно будет увеличить пропорционально количеству голов внимания.

2. Далее определяем парные зависимости между элементами последовательности. Для этого перемножим вектор *Query* с векторами *Key* всех элементов последовательности. Данная итерация повторяется для вектора *Query* каждого элемента последовательности. В результате данной итерации получаем матрицу *Score* размером *N*N*, где *N* — размер последовательности.

$$S = QK^T$$

В результате этой операции мы ожидаем получить по одному коэффициенту зависимости между парой элементов последовательности для каждой головы внимания. Но операция умножения двух конкатенированных векторов вернет нам только одно значение. Как и в случае одноголового *Self-Attention.*

Мы можем изменить размерность векторов и привести их к двумерным матрицам. В этом есть здравый смысл, так как мы можем выделить данные каждой головы в отдельную строку-вектор. Но поставив их в формулу выше, на выходе мы получим квадратную матрицу со стороной равной количеству голов, хотя ожидали получить вектор с размером равным количеству голов.

Выход все же есть. Давайте вспомним правило умножения матриц.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{22}b_{12} + a_{22}b_{22} \end{bmatrix}$$

Подставим сюда наши двумерные матрицы многоголового внимания. И не забудем, что вторая матрица транспонируется перед умножением.

$$\begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \\ Q_{31} & Q_{32} \end{bmatrix} * \begin{bmatrix} K_{11} & K_{21} & K_{31} \\ K_{12} & K_{22} & K_{32} \end{bmatrix} =$$

$$= \begin{bmatrix} Q_{11}K_{11} + Q_{12}K_{12} & Q_{11}K_{21} + Q_{12}K_{12} & Q_{13}K_{31} + Q_{12}K_{32} \\ Q_{21}K_{11} + Q_{22}K_{12} & Q_{21}K_{21} + Q_{22}K_{12} & Q_{21}K_{31} + Q_{22}K_{32} \\ Q_{31}K_{11} + Q_{32}K_{12} & Q_{31}K_{21} + Q_{32}K_{12} & Q_{31}K_{31} + Q_{32}K_{32} \end{bmatrix}$$

Как можно заметить, вектор, который мы ожидали получить, образовывает диагональ матрицы результатов. А все остальные операции для нас только потеря ресурсов. Но мы можем разложить данную процедуру на операции. К примеру, не будем транспонировать матрицу ключей и воспользуемся адамарным произведением матриц (поэлементное умножение матриц).

$$\begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \\ Q_{31} & Q_{32} \end{bmatrix} \circ \begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \\ K_{31} & K_{32} \end{bmatrix} = \begin{bmatrix} Q_{11}K_{11} & Q_{12}K_{12} \\ Q_{21}K_{21} & Q_{22}K_{22} \\ Q_{31}K_{31} & Q_{32}K_{32} \end{bmatrix}$$

После этого для получения ожидаемого результата нам достаточно лишь построчно сложить элементы матрицы.

$$\begin{bmatrix} Q_{11}K_{11} & Q_{12}K_{12} \\ Q_{21}K_{21} & Q_{22}K_{22} \\ Q_{31}K_{31} & Q_{32}K_{32} \end{bmatrix} = \begin{bmatrix} Q_{11}K_{11} + Q_{12}K_{12} \\ Q_{21}K_{21} + Q_{22}K_{22} \\ Q_{31}K_{31} + Q_{32}K_{32} \end{bmatrix} = \begin{bmatrix} S_1 \\ S_2 \\ S_3 \end{bmatrix}$$

Да, в итоге мы получили результат за две операции вместо одной. Но нужно отметить два момента:

- В формуле *Self-Attention* используется транспонированная матрица, а это тоже операция над матрицей, хотя и не выделена отдельно. И на ее выполнение тоже требуются ресурсы. При разложении на операции мы отказались от данной процедуры.
- Определение вектора коэффициентов осуществляется в две операции независимо от количества голов внимания.

3. Следующим этапом разделим полученные значение на квадратный корень из размерности вектора Key и нормализуем функцией *Softmax* в разрезе каждого *Query*. Таким образом, получаем коэффициенты попарной взаимозависимости между элементами последовательности.

В этом пункте мы не будем что-либо усложнять или упрощать. Деление матрицы на константу всегда выполняется поэлементно независимо от размера матрицы, а нормализовать данные нам придется в разрезе голов внимания.

4. Умножаем каждый вектор *Value* на соответствующий коэффициент взаимозависимости и получаем скорректированное значение элемента. Цель данной итерации — акцентировать внимание на релевантных элементах и снизить влияние не релевантных значений.

Для решения этой задачи воспользуемся приемами, применяемыми в пункте 2. Сначала изменим размерность вектора значений и приведем его к двумерной матрице. В ней строки будут соответствовать каждой отдельной голове внимания.

$$[V_{11} \quad V_{12} \quad V_{21} \quad V_{22} \quad V_{31} \quad V_{32}] = \begin{bmatrix} V_{11} & V_{12} \\ V_{21} & V_{22} \\ V_{31} & V_{32} \end{bmatrix}$$

После этого мы можем воспользоваться поэлементным умножением вектора коэффициентов зависимости на матрицу значений.

$$\begin{bmatrix} S_1 \\ S_2 \\ S_3 \end{bmatrix} \circ \begin{bmatrix} V_{11} & V_{12} \\ V_{21} & V_{22} \\ V_{31} & V_{32} \end{bmatrix} = \begin{bmatrix} S_1 V_{11} & S_1 V_{12} \\ S_2 V_{21} & S_2 V_{22} \\ S_3 V_{31} & S_3 V_{32} \end{bmatrix}$$

5. Далее суммируем все скорректированные векторы *Value* для каждого элемента. Результатом данной операции и будет вектор выходных значений слоя *Self-Attention*.

В последнем пункте нам также нечего добавить. Суммировать значение элементов векторов мы будем отдельно в разрезе запросов Query и голов внимания. Мы легко можем распараллелить выполнение этой задачи, создав отдельный поток для нахождения каждого отдельного вектора.

После выполнения всех пунктов механизма *Self-Attention* в режиме нескольких голов внимания мы получаем по вектору результатов для каждой головы внимания. Соответственно, общий размер тензора результатов будет превышать размер тензора исходных данных пропорционально количеству голов. Для понижения размерности алгоритмом *Multi-Head Self-Attention* предусматривается умножение конкатенированного тензора результатов на дополнительную матрицу весов W_0. Как вы понимаете, данная процедура очень напоминает операцию полносвязного нейронного слоя без функции активации. Аналогичные операции мы выполняли в пункте 1 для определения векторов *Query, Key, Values*. А значит, мы можем воспользоваться тем же решением и использовать ранее созданные сверточные слои.

Здесь можно отметить еще один момент. При описании работы блока *Self-Attention* мы обращали внимание на момент равенства размера векторов описания одного элемента последовательности тензоров значений *Value* и исходных данных. Такое требование исходило из необходимости последующего сложения тензоров результатов *Self-Attention* и исходных данных. В случае же многоголового внимания мы в любом случае получаем конкатенированный тензор результатов больше тензора исходных данных. Для приведения их в соответствие используется умножение тензора результатов на матрицу W_0. Следовательно, с целью экономии ресурсов мы можем понизить размерность вектора описания одного элемента последовательности в тензоре значений *Value* без риска получения ошибки при последующей обработке данных.

Дальнейший алгоритм работы энкодера трансформера остается без изменений, и мы можем воспользоваться наработками из предыдущего раздела.

Теперь, когда у нас есть полное представление о принципах реализации алгоритма, мы можем перейти к его реализации.

5.2.2 Построение Multi-Head Self-Attention средствами MQL5

Приступая к реализации блока многоголового внимания Multi-Head Self-Attention мы можем отметить его сильное сходство с ранее рассмотренным блоком *Self-Attention*. Это и не удивительно, ведь технология *Multi-Head Self-Attention* является логическим развитием технологии *Self-Attention*. Поэтому вполне логично будет при создании нового класса наследоваться не от базового класса нейронного слоя *CNeuronBase*, а от класса блока внимания *CNeuronAttention*.

При таком варианте наследования мы наследуем от родительского класса помимо методов и объектов базового класса еще и объекты класса *CNeuronAttention.* В том числе:

- *m_cQuerys* — сверточный слой формирования тензора запросов *Query*;
- *m_cKeys* — сверточный слой формирования тензора ключей *Key*;
- *m_cValues* — сверточный слой формирования тензора значений *Value*;

- *m_cScores* — буфер матрицы коэффициентов зависимости;
- *m_cAttentionOut* — базовый слой исходных данных для записи результатов работы блока *Self-Attention*;
- *m_cFF1* и *m_cFF2* — сверточные слои блока *Feed Forward*.

Как мы определились в разделе описания архитектурного решения, все объекты будут использоваться по своему прямому назначению. Мы только увеличим их размер пропорционально количеству голов внимания. Таким образом, для реализации алгоритма *Multi-Head Self-Attention* нам остается добавить внутренний слой матрицы W_0 и переменную для записи количества голов внимания.

```
class CNeuronMHAttention    :  public CNeuronAttention
  {
protected:
   CNeuronConv       m_cW0;

   int               m_iHeads;

public:
                     CNeuronMHAttention(void);
                    ~CNeuronMHAttention(void);
   //---
   virtual bool      Init(const CLayerDescription *desc) override;
   virtual bool      SetOpenCL(CMyOpenCL *opencl) override;
   virtual bool      FeedForward(CNeuronBase *prevLayer) override;
   virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer, bool read) override;
   virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                   VECTOR &Beta, VECTOR &Lambda) override;
   //--- методы работы с файлами
   virtual bool      Save(const int file_handle) override;
   virtual bool      Load(const int file_handle) override;
   //--- метод идентификации объекта
   virtual int       Type(void) override const { return(defNeuronMHAttention);  }
  };
```

Касательно методов класса мы будем переопределять ставший уже стандартным набор методов:

- *Init* — метод инициализации класса;
- *SetOpenCL* — метод указания хендла используемого контекста *OpenCL*;
- *FeedForward* — метод прямого прохода;
- *CalcHiddenGradient* — метод распределения градиента ошибка через скрытый слой;
- *CalcDeltaWeights* — метод распределения градиента ошибки до уровня матрицы весовых коэффициентов текущего нейронного слоя;
- *UpdateWeights* — метод обновления матрицы весов коэффициентов текущего нейронного слоя;
- *Save* — метод сохранения данных нейронного слоя в файл;
- *Load* — метод загрузки данных нейронного слоя из файла;
- *Type* — метод идентификация типа нейронного слоя.

Ну а начнем мы работу с конструктора класса. В нем мы создаем экземпляры объектов, необходимых для полноценного функционирования класса, и инициализируем внутренние переменные значениями по умолчанию. Выше мы определили только один новый объект — сверточный слой *m_cW0*. Мы будем использовать статические объекты, как и в родительском классе. А значит, в конструкторе класса нам остается только указать начальное значение для количества голов внимания. Деструктор класса остается пустым.

```
CNeuronMHAttention::CNeuronMHAttention(void) :  m_iHeads(8)
  {
  }
```

На следующем этапе мы займемся методом инициализации класса. Несмотря на то, что большинство объектов было унаследовано от родительского класса, мы не можем использовать его метод инициализации, так как для использования их в алгоритме *Multi-Head Self-Attention* потребуются другие размеры тензоров. Следовательно, мы вынуждены будем переписать метод инициализации полностью. В то же время для построения метода инициализации мы будем использовать алгоритм, аналогичный соответствующему методу родительского класса.

Как и аналогичные методы всех рассмотренных ранее классов, в параметрах метода получаем указатель на объект описания конфигурации создаваемого нейронного слоя. Сразу организовываем блок проверки полученных данных. В первую очередь, мы проверяем действительность полученного указателя. Только после подтверждения действительности его актуальности мы проверяем его содержимое:

- Тип создаваемого нейронного слоя в описании конфигурации должен соответствовать типу класса (параметр *type*).
- В создаваемом слое должен быть хотя бы один элемент анализируемой последовательности (параметр *count*).
- Размер вектора описания одного элемента исходных данных должен быть больше нуля (параметр *window*).
- Размер вектора ключа одного элемента последовательности должен быть больше нуля (параметр *window_out*).
- Должна быть как минимум одна голова внимания (параметр *step*).

```
bool CNeuronMHAttention::Init(const CLayerDescription *desc)
  {
//--- проверяем исходные данные
   if(!desc || desc.type != Type() ||
      desc.count <= 0 || desc.window <= 0 || desc.window_out <= 0 ||
      desc.step <= 0)
      return false;
```

Наверное странно выглядит использование параметра *step* для указания количества голов внимания. Но, как вы помните, в рамках реализации механизмов внимания размер шага окна исходных данных всегда равен размеру самого окна. Поэтому данный параметр у нас остается свободным. А чтобы не наращивать бесконтрольно размер объекта описания нейронного слоя, было принято решение максимально эффективно использовать существующие переменные класса. Но если для вас читабельность кода имеет больший приоритет, то вы всегда можете определить необходимое вам количество переменных для описания архитектуры создаваемого нейронного слоя и их наименование.

После успешного прохождения блока контролей мы сохраним ключевые параметры описания создаваемого нейронного слоя в локальные переменные.

```
//--- сохраняем константы
   m_iWindow = desc.window;
   m_iUnits = desc.count;
   m_iKeysSize = desc.window_out;
   m_iHeads = desc.step;
```

Как и в аналогичных методах всех ранее рассмотренных классов, следующим шагом мы вызовем метод базового нейронного слоя, в котором будут инициализированы унаследованные объекты. Мы не можем вызвать метод родительского класса, так как будут созданы объекты других размеров и нам придется изменять объекты. А мы не хотим выполнять одну работу дважды. Поэтому «прыгаем через голову» и обращаемся напрямую к методу базового класса.

Но и тут есть нюанс. Перед вызовом метода базового класса нам нужно немного подправить описание архитектуры создаваемого нейронного слоя. При этом мы не знаем, какие у пользователя планы на объект описания архитектуры слоя, полученного в параметрах. Помним, что мы говорили об объектах и указателях на них. В параметрах мы получили указатель на объект. Когда мы вносим изменения в объект, они получат свое отражение и на стороне основной программы у пользователя. А если пользователь использует один объект для описания нескольких нейронных слоев, то с большой долей вероятности при создании последующих нейронных слоев он получит ошибку. Также слои могут быть созданы с некорректной архитектурой. Поэтому мы создадим новый объект описания архитектуры нейронного слоя и заполним его нужными нам параметрами.

В родительском классе мы отработали технологию с подменой указателей на объект буферов результатов и градиентов ошибки. Поэтому нам не важно, какими будут созданы эти объекты в методе базового класса — в параметрах размера слоя и окна результатов можно указать любые значения. Чтобы не выполнять лишних операций, укажем их минимально больше нуля.

Для исключения создания ненужных объектов укажем размер окна исходных данных равным нулю и отключим функцию активации.

Тип нейронного слоя оставляем тот, что получили в описании от пользователя.

Далее вызываем метод базового нейронного слоя, передав ему правильное описание.

```
//--- вызываем метод инициализации родительского класса
   CLayerDescription* temp = new CLayerDescription();
   if(!temp)
      return false;
   temp.type = desc.type;
   temp.optimization = desc.optimization;
   temp.activation = AF_NONE;
   temp.count = desc.count;
   temp.window_out = 1;
   temp.window = 0;
   if(!CNeuronBase::Init(temp))
     {
      delete temp;
      return false;
     }
```

В созданном выше описании архитектуры нейронного слоя изменим тип создаваемого объекта его размер. Этого достаточно для создания объекта конкатенированных результатов работы голов внимания.

```
//--- инициализируем AttentionOut
   temp.type = defNeuronBase;
   temp.count = (int)(m_iUnits * m_iKeysSize * m_iHeads);
   if(!m_cAttentionOut.Init(temp))
     {
      delete temp;
      return false;
     }
   if(!m_cAttentionOut.GetOutputs().m_mMatrix.Reshape(m_iUnits, m_iKeysSize * m_iHead
      !m_cAttentionOut.GetGradients().m_mMatrix.Reshape(m_iUnits, m_iKeysSize * m_iHe
      return false;
```

После инициализации объекта мы немного изменим формат буферов результатов и градиентов ошибки.

Далее нам предстоит создать внутренние сверточные нейронные слои. Первыми мы будем создавать внутренние нейронные слои для формирования тензоров *Query, Key* и *Value*. Все они на вход получают последовательность исходных данных. Поэтому в параметрах *window* и *step* мы укажем размер вектора описания одного элемента последовательности исходных данных.

Количество фильтров используемого сверточного слоя, указываемое в параметре *window_out*, по нашей логике должно соответствовать размеру вектора ключа одного элемента последовательности. Но, как вы помните, при рассмотрении архитектурного решения данного класса мы определили использование конкатенированных тензоров. Поэтому количество фильтров мы увеличим пропорционально количеству создаваемых голов внимания.

Количество элементов в последовательности на всех этапах у нас остается постоянным. Следовательно, мы можем записать в параметр *count* количество элементов исходной последовательности, полученное от внешней программы.

Функция активации для создаваемых нейронных слоев архитектурой *Multi-Head Self-Attention* не предусмотрена. Поэтому в параметре *activation* оставим константу *AF_NONE*.

Метод оптимизации параметров всех нейронных слоев один, данный параметр оставляем без изменений.

```
//--- создаем описание для внутренних нейронных слоев
   if(!temp)
      return false;
   temp.type = defNeuronConv;
   temp.window = m_iWindow;
   temp.window_out = (int)(m_iKeysSize * m_iHeads);
   temp.step = m_iWindow;
   temp.count = m_iUnits;
```

Первым мы инициализируем внутренний слой для создания тензора запросов *Query.* Проверяем результат выполнения операции, чтобы исключить возможные критические ошибки при дальнейшем выполнении кода метода.

```
//--- инициализируем Querys
   if(!m_cQuerys.Init(temp))
     {
      delete temp;
      return false;
     }
   m_cQuerys.SetTransposedOutput(true);
```

После успешной инициализации сверточного нейронного слоя мы устанавливаем флаг транспонирования тензора результатов. Напомню, что этот флаг мы ввели для возможности получения тензора результатов, в строках которого содержались не элементы одного фильтра, а элементы всех фильтров для одного элемента последовательности.

Аналогично инициализируем объекты сверточных нейронных слоев для создания тензоров *Key* и *Value.*

```
//--- инициализируем Keys
   if(!m_cKeys.Init(temp))
     {
      delete temp;
      return false;
     }
   m_cKeys.SetTransposedOutput(true);
```

Обратите внимание, при инициализации объекта сверточного нейронного слоя для формирования тензора *Value* мы не выравниваем количество используемых фильтров с размером окна исходных данных, как это было в классе с одной головой внимания *CNeuronAttention*. Ведь использование матрицы W_0 позволяет нам отойти от этого правила. А снижение размерности вектора поможет сократить ресурсы и время выполнения операций. В свою очередь, после воссоздания полного алгоритма метода *Multi-Head Self-Attention* вы сможете на практических примерах оценить плюсы и минусы такой реализации.

```
//--- инициализируем Values
   if(!m_cValues.Init(temp))
     {
      delete temp;
      return false;
     }
   m_cValues.SetTransposedOutput(true);
```

После инициализации первой группы внутренних сверточных слоев, следуя по алгоритму механизма *Multi-Head Self-Attention*, мы инициализируем буфер матрицы коэффициентов зависимости *m_cScores*. Заполняем его нулевыми значениями, указав требуемый размер буфера. И опять проведем параллель с классом *CNeuronAttention*. Если ранее мы создавали квадратную матрицу со стороной равной количеству элементов последовательности, то теперь нам надо столько таких матриц, сколько голов внимания. В то же время мы с вами договорились использовать конкатенированную матрицу. Поэтому увеличим объем буфера пропорционально количеству используемых голов внимания. К сожалению, *MQL5* не поддерживает трехмерные матрицы. В рамках двумерной матрицы мы будем использовать строки для распределения буфера по головам внимания.

```
//--- инициализируем Scores
   if(!m_cScores.BufferInit(m_iHeads, m_iUnits * m_iUnits))
     {
      delete temp;
      return false;
     }
```

И теперь настал момент инициализации дополнительного сверточного слоя, выполняющего функционал матрицы W_0 в алгоритме *Multi-Head Self-Attention*. Скорректируем описание архитектуры создаваемого нейронного слоя.

Тип создаваемого нейронного слоя уже указан, следовательно, нам нет необходимости повторно его указывать.

Размер окна исходных данных определяем как произведение размера вектора описания одного элемента последовательности в тензоре Values на количество голов внимания. Напомню, что в данной реализации мы изменили размер указанного вектора до аналогичного в тензоре ключей Key. Таким образом, размер окна исходных данных определяется как произведение размера вектора ключей одного элемента последовательности на количество голов внимания (*m_iKeysSize * m_iHeads*).

Размер шага окна исходных данных мы приравняем к размеру самого окна.

Согласно алгоритму *Multi-Head Self-Attention*, матрица W_0 используется для выравнивания размеров тензора результатов блока многоголового внимания с тензором исходных данных. Поэтому количество фильтров в данном сверточном слое мы укажем равным размеру вектора описания одного элемента последовательности исходных данных, подаваемых на вход блока *Multi-Head Self-Attention*.

Алгоритмом *Multi-Head Self-Attention* функция активации для данной матрицы не предусмотрена. Поэтому, в соответствующем поле оставим константу *AF_NONE*.

Метод оптимизации матрицы весов для всех слоев нейронной сети, в том числе и внутренних слоев отдельных блоков, используется один. Следовательно, параметры указывающие на используемый метод оптимизации мы оставляем без изменений.

```
//--- инициализируем W0
   temp.window = (int)(m_iKeysSize * m_iHeads);
   temp.step = temp.window;
   temp.window_out = m_iWindow;
   if(!m_cW0.Init(temp))
     {
      delete temp;
      return false;
     }
   m_cW0.SetTransposedOutput(true);
```

После указания всех необходимых параметров описания создаваемого нейронного слоя мы вызываем метод инициализации нашего сверточного нейронного слоя *m_cW0.Init* и проверяем результат выполнения операций.

В заключение блока инициализации сверточного слоя *m_cW0* установим флаг транспонирования тензора результатов.

На этом завершается работа по инициализации объектов блока *Multi-Head Self-Attention*. Далее переходим к работе над блоком *Feed Forward*. Функционал и архитектура данного блока полностью перенесены из класса *CNeuronAttention*. Но так как нам пришлось полностью переопределять метод инициализации класса, то повторим действия по инициализации внутренних слоев *m_cFF1* и *m_cFF2.*

Алгоритм инициализации нейронного слоя остается прежним. Подготавливаем описание создаваемого нейронного слоя и вызываем метод его инициализации. Для описания сверточного нейронного слоя *m_cFF1* мы будем использовать объект описания сверточного нейронного слоя, который уже не раз использовали в этом методе. Поэтому укажем лишь изменяемые параметры, а остальные уже содержатся в объекте описания нейронного слоя.

- Размер окна исходных данных (*window*) — равен размеру вектора описания одного элемента последовательности тензора исходных данных, подаваемого на вход нашего блока *Multi-Head Self-Attention*. Данный параметр мы получаем от внешней программы и сохраняем в переменной *m_iWindow*. Следовательно, можем передать в параметр значение указанной переменной.
- Размер шага окна исходных данных (*step*) мы приравняем к размеру самого окна исходных данных.
- Количество используемых фильтров (*window_out*) — согласно архитектуре трансформера, предложенной авторами, размер выхода первого слоя блока *Feed Forward* в четыре раза превышает размер исходных данных. Воспользуемся этим коэффициентом. Но при реализации своих практических задач вы всегда можете изменить данный коэффициент или даже добавить его в описание конфигурации создаваемого нейронного слоя и провести практические тесты для выбора наиболее подходящего коэффициента для решения ваших конкретных задач.
- Функция активации (*activation*) — для данного слоя авторы предлагают использовать *ReLU* в качестве функции активации. Мы же заменили ее на близкую функцию *Swish*. График данной функции очень близок к графику функции, предложенной авторами. Но при этом он не содержит изломов и дифференцируем на всем протяжении значений.

- Параметры оптимизации матрицы весов остаются без изменений.

```
//--- инициализируем FF1
   temp.window = m_iWindow;
   temp.step = temp.window;
   temp.window_out = temp.window * 4;
   temp.activation = AF_SWISH;
   temp.activation_params[0] = 1;
   temp.activation_params[1] = 0;
   if(!m_cFF1.Init(temp))
     {
      delete temp;
      return false;
     }
   m_cFF1.SetTransposedOutput(true);
```

После того, как мы указали все параметры описания конфигурации создаваемого сверточного нейронного слоя, вызовем его метод инициализации. И, конечно, проверим результат выполнения операций.

И только при успешной инициализации объекта сверточного нейронного слоя установим флаг транспонирования тензора результатов.

Теперь мы можем перейти к инициализации последнего используемого в классе объекта — второго сверточного слоя блока *Feed Forward m_cFF2.* В результате работы данного нейронного слоя мы вновь возвращаемся к размерности тензора исходных данных. Поэтому в объекте описания структуры создаваемого нейронного слоя нам предстоит поменять местами значения окна исходных данных и количество используемых фильтров. Обычно для такой операции требуется локальная переменная для временного хранения одного из значений. Но в нашем случае параметры размера окна исходных данных и его шаг равны. Поэтому мы сначала в параметр размера окна исходных данных запишем количество фильтров предыдущего слоя. Затем в параметр количества фильтров укажем значение шага окна предыдущего сверточного слоя. И в заключение приравняем размер шага окна исходных данных его размеру.

Архитектура трансформера не предусматривает функцию активации для данного слоя. Но мы представим возможность пользователю поэкспериментировать. Для этого перенесем функцию активации и ее параметры из описания архитектуры, предоставленной пользователем в параметрах данного метода.

```
//--- инициализируем FF2
   temp.window = temp.window_out;
   temp.window_out = temp.step;
   temp.step = temp.window;
   temp.activation = desc.activation;
   temp.activation_params = desc.activation_params;
   if(!m_cFF2.Init(temp))
     {
      delete temp;
      return false;
     }
   m_cFF2.SetTransposedOutput(true);
   delete temp;
```

Когда указаны все необходимые параметры описания структуры создаваемого нейронного слоя, вызываем его метод инициализации и устанавливаем флаг транспонирования тензора результатов. При этом не забываем проверить результаты выполнения операций.

Теперь, когда все необходимые объекты инициализированы, мы можем без всякого риска на ошибку удалить локальный объект описания нейронного слоя.

Далее воспользуемся отработанным в классе *CNeuronAttention* приемом и подменим указатели на буферы результатов и градиентов ошибки нашего класса многоголового внимания на аналогичные буферы внутреннего сверточного нейронного слоя *m_cFF2*. Это позволит нам исключить излишние затраты на копирование данных между буферами. Также нам не потребуется дополнительная память для хранения дубликатов данных. Для этого мы сначала проверяем указатели и при необходимости удаляем не нужные ранее созданные объекты. А потом в переменные передаем указатели на объекты сверточного слоя *m_cFF2*.

```
//--- для исключения копирования буферов осуществим их подмену
   if(!SetOutputs(m_cFF2.GetOutputs()))
      return false;
   if(m_cGradients)
      delete m_cGradients;
   m_cGradients = m_cFF2.GetGradients();
//---
   SetOpenCL(m_cOpenCL);
//---
   return true;
  }
```

В заключение метода передадим всем объектам указатель на используемый контекст *OpenCL*. После этого выходим из метода с положительным результатом.

На этом мы завершаем работу над методом инициализации класса. Но надо отметить один незакрытый вопрос. В конце метода инициализации мы вызывали метод передачи указателя на контекст *OpenCL*. Мы еще не переопределили его, и в таком виде будет вызван аналогичный метод родительского класса. Он вполне функционален, но не распространяется на объекты, объявленные в теле этого класса. А среди них всего лишь один объект — сверточный слой *m_cW0*. Следовательно, и метод будет небольшим.

Как и аналогичные методы всех ранее рассмотренных классов, метод *CNeuronMHAttention::SetOpenCL* в параметрах получает указатель на объект работы с контекстом

OpenCL. Его нам и предстоит распространить по всем внутренним объектам. Но для начала, конечно, надо бы проверить действительность полученного указателя. Вместо этого мы вызовем аналогичный метод родительского класса, в котором уже организованы все контроли и передача указателя до наследованных объектов. Таким образом, после завершения работы метода родительского класса, нам остается лишь передать указатель в новые объекты, которые были объявлены в теле этого класса. Но при этом мы будем передавать не указатель, полученный в параметрах, а указатель из локальной переменной класса, унаследованной от родительского объекта. Дело в том, что метод родительского класса проверил полученный указатель и сохранил его в локальную переменную. Также он передал его всем объектам, которые мы унаследовали от родительского класса. Следовательно, чтобы все объекты работали в одном контексте, мы передаем во внутренние объекты уже проверенный указатель.

```
bool CNeuronMHAttention::SetOpenCL(CMyOpenCL *opencl)
  {
//--- вызов метода родительского класса
   CNeuronAttention::SetOpenCL(opencl);
//--- вызываем аналогичный метод для внутреннего слоя
   m_cW0.SetOpenCL(m_cOpenCL);
//---
   return(!!m_cOpenCL);
  }
```

После передачи указателя во все внутренние объекты, а в данном случае это один сверточный слой, мы выходим из метода и возвращаем результат, указывающий на действительность указателя используемого контекста.

На этом мы завершаем работу по созданию и инициализации объекта нашего класса многоголового внимания и переходим к следующему этапу — организации прямого прохода.

5.2.2.1 Метод прямого прохода Multi-Head Self-Attention

Мы уже организовали процесс создания и инициализации класса многоголового внимания *CNeuronMHAttention*. И сейчас, когда уже есть все внутренние объекты нашего класса, мы можем перейти к организации прямого прохода.

За осуществление прямого прохода во всех классах нашей библиотеки отвечает виртуальный метод *FeedForward*. Придерживаясь общей системы организации классов и их методов, а также принципов наследования, в данном классе мы сохраним определенную ранее структуру методов и переопределим метод *FeedForward.* Как и аналогичный метод родительского класса, в параметрах метод прямого прохода получает указатель на объект предыдущего нейронного слоя. По уже не раз опробованной схеме, в начале метода мы организуем блок контролей. В нем мы проверяем актуальность указателей на все используемые в методе динамические объекты. В данном случае мы проверяем указатель на полученный в параметрах нейронный слой, его буфер результатов и буфер результатов внутреннего слоя конкатенированного выхода блока внимания.

```
bool CNeuronMHAttention::FeedForward(CNeuronBase *prevLayer)
  {
//--- проверяем актуальность всех объектов
   if(!prevLayer || !prevLayer.GetOutputs() ||
      !m_cAttentionOut.GetOutputs())
      return false;
```

После успешного прохождения блока контролей мы формируем конкатенированные тензоры запросов *Query*, ключей *Key* и значений *Value*. Для этого мы вызываем методы прямого прохода внутренних сверточных слоев *m_cQuerys*, *m_cKeys* и *m_cValues*. Созвучность тензоров в архитектуре *Multi-Head Self-Attention* и вызываемых объектов неслучайна: это делает код более читабельным и позволяет отслеживать выстраиваемый алгоритм.

```
   if(!m_cQuerys.FeedForward(prevLayer))
      return false;
   if(!m_cKeys.FeedForward(prevLayer))
      return false;
   if(!m_cValues.FeedForward(prevLayer))
      return false;
```

Обязательно контролируем процесс выполнения операций.

Далее, согласно алгоритму *Multi-Head Self-Attention,* нам предстоит определить коэффициенты зависимости между элементами последовательности и вывести конкатенированный результат работы всех голов внимания. Данный функционал является связующим между внутренними нейронными слоями. Он не покрывается использование других объектов и будет полностью выстраиваться внутри данного метода.

Как вы помните, при построении всех процессов в методах классов нашей библиотеки мы создаем две ветки алгоритма: стандартными средствами *MQL5* и многопоточные вычисления на *GPU* с использованием технологии *OpenCL*. Как всегда, в данном разделе мы рассмотрим реализацию алгоритма стандартными средствами *MQL5*. А к реализации многопоточных вычислений с использованием технологии *OpenCL* мы вернемся позже.

И тут перед нами встает вопрос, как организовать работу. У нас есть три измерения:

- головы внимания,
- элементы последовательности,

- вектор описания одного элемента последовательности.

Матричные операции дают нам возможность оперировать только двумерными матрицами. Одно из используемых измерений будет вектором описания одного элемента последовательности. Нетрудно догадаться, что в большинстве случаев размер последовательности в десятки раз будет превышать количество голов внимания. Поэтому мы сделаем цикл перебора голов внимания, а в теле цикла проанализируем последовательности каждой головы внимания.

Но перед организацией цикла нам необходимо провести небольшую подготовительную работу. Разделим конкатенированные результаты предыдущего этапа реализуемого алгоритма на несколько матриц голов внимания. Для этого воспользуемся динамическим массивами матриц, что даст нам подобие трехмерных матриц. Индекс элемента в массиве укажет нам на индекс головы внимания, а каждый элемент массива будет представлен табличной матрицей, строки которой будут представлять отдельные элементы последовательности. Для удобства работы с массивами присвоим им имена, созвучные с их содержимым.

```
//--- разветвление алгоритма по вычислительному устройству
   MATRIX out;
   if(!m_cOpenCL)
     {
      if(!out.Init(m_iHeads, m_iUnits * m_iKeysSize))
         return false;
      MATRIX querys[], keys[], values[];
      if(!m_cQuerys.GetOutputs().m_mMatrix.Vsplit(m_iHeads, querys))
         return false;
      if(!m_cKeys.GetOutputs().m_mMatrix.Vsplit(m_iHeads, keys))
         return false;
      if(!m_cValues.GetOutputs().m_mMatrix.Vsplit(m_iHeads, values))
         return false;
```

После завершения подготовительной работы мы можем перейти непосредственно к операциям вычисления коэффициентов зависимости. При решении подобной задачи в методе прямого прохода родительского класса *CNeuronAttention* мы использовали матричные операции. Сейчас мы воспользуемся тем же алгоритмом, только нам потребуется повторить их в цикле с числом итераций равным количеству голов внимания.

Напомню, алгоритм *Multi-Head Self-Attention* предусматривает деление коэффициентов зависимости на квадратный корень из размерности вектора ключа *Key* и последующую нормализацию полученных значений функцией *Softmax* в разрезе элементов запросов *Query.*

$$SoftMax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Следуя алгоритму, мы перемножаем матрицы *querys* и транспонированную *keys*, делим на квадратный корень их размерности и сразу вычисляем экспоненциальное значение. В полученной матрице берем построчные суммы значений и организуем вложенный цикл нормализации данных.

```
   for(int head = 0; head < m_iHeads; head++)
     {
      //--- определяем Scores
      MATRIX sc = exp(querys[head].MatMul(keys[head].Transpose()) /
                                                          sqrt(m_iKeysSize));
      VECTOR sum = sc.Sum(1);
      for(uint r = 0; r < sc.Rows(); r++)
         if(!sc.Row(sc.Row(r) / sum[r], r))
            return false;
```

Как можно заметить, алгоритм полностью повторяет аналогичные операции родительского класса.

Теперь, когда у нас уже есть посчитанная матрица коэффициентов зависимостей между элементами, мы можем двигаться дальше по алгоритму *Multi-Head Self-Attention* и определить значения конкатенированного тензора результатов работы в части анализируемой головы внимания. Для этого нам достаточно вычислить произведения двух матриц — коэффициентов зависимости и значений *Values*.

```
      //--- выход блока внимания
      MATRIX temp = sc.MatMul(values[head]).Transpose();
```

Следует обратить особое внимание на сбор результатов в единый конкатенированный тензор. Вся логика построения алгоритма предполагает, что тензор конкатенированного результата будет представлять собой табличную матрицу. Каждая строка матрицы будет содержать вектор конкатенированного результата отдельного элемента последовательности. Я решил данную задачу следующим образом.

В результате операции умножения мы получили табличную матрицу, количество строк которой равно количеству элементов последовательности, а число столбцов равно размеру вектора описания одного элемента последовательности. Мы транспонируем матрицу, переформатируем ее в матрицу-строку, и добавим полученную строку в конкатенированную матрицу. На данном этапе в конкатенированной матрице каждая голова внимания будет иметь свою строку.

Аналогично поступаем с матрицей коэффициентов зависимости.

```
      if(!temp.Reshape(1, m_iUnits * m_iKeysSize))
         return false;
      if(!sc.Reshape(1, m_iUnits * m_iUnits))
         return false;
      if(!m_cScores.m_mMatrix.Row(sc.Row(0), head))
         return false;
      if(!out.Row(temp.Row(0), head))
         return false;
     }
```

После завершения итераций цикла и получения результатов работы всех голов внимания мы переформатируем конкатенированную матрицу. Сделаем количество столбцов равным количеству элементов последовательности и транспонируем матрицу. В результате получим количество строк равное числу элементов в анализируемой последовательности. Именно такой формат нам нужен для передачи на следующий сверточный слой нашего блока многоголового внимания. Сохраним матрицу в буфер результатов внутреннего слоя *m_cAttentionOut*.

```
      if(!out.Reshape(m_iHeads * m_iKeysSize, m_iUnits))
         return false;
      m_cAttentionOut.GetOutputs().m_mMatrix = out.Transpose();
     }
   else // Блок OpenCL
     {
      return false;
     }
```

На этом заканчивается блок разделения алгоритма в зависимости от устройства выполнения операций. Возвращаемся к использованию методов наших внутренних нейронных слоев. Для блока многопоточных операций с использованием технологии *OpenCL* установим временную «заглушку» в виде возврата ложного значения выполнения операций метода. К ней мы вернемся в следующих разделах.

Продолжаем двигаться по алгоритму *Multi-Head Self-Attention*. На следующем этапе нам предстоит понизить размерность конкатенированного тензора результатов всех голов внимания до размера тензора исходных данных. Для этих целей алгоритмом предусмотрено использование обучаемой матрицы W_0. Указанная матрица выполняет двойную роль. Во-первых, она служит для изменения размерности тензора. Во-вторых, она выполняет взвешенное сложение всех голов внимания в некую единую сущность, тем самым определяя влияние каждой головы внимания на конечный результат.

Мы же для реализации указанной задачи воспользуемся объектом сверточного слоя. Мы уже создали сверточный нейронный слоя *m_cW0*, и теперь нам остается вызвать его метод прямого прохода. В параметрах методу передадим указатель на объект нейронного слоя *m_cAttentionOut*. Не забудьте проверить результат выполнения операции.

```
   if(!m_cW0.FeedForward(GetPointer(m_cAttentionOut)))
      return false;
```

После успешного завершения операций метода в буфере результатов нашего нейронного слоя будет результат работы блока *Multi-Head Self-Attention*. Согласно алгоритму трансформера нам предстоит сложить полученный результат с исходными данными в единый тензор и нормализовать полученный результат по формулам:

$$\bar{a} = \frac{1}{h}\sum_{i=1}^{h} a_i$$

$$\hat{a} = \frac{a - \bar{a}}{\sqrt{\frac{1}{h}\sum_{i=1}^{h}(a - \bar{a})^2}}$$

При работе над родительским классом *CNeuronAttention* для данных операций мы создали отдельные методы. Теперь воспользуемся плодами проделанной ранее работы.

```
//--- суммируем с исходными данными и нормализуем
   if(!m_cW0.GetOutputs().SumArray(prevLayer.GetOutputs()))
      return false;
   if(!NormlizeBuffer(m_cW0.GetOutputs(), GetPointer(m_cStd), 0))
      return false;
```

И, конечно, не забываем контролировать процесс выполнения операций на каждом шаге.

Контроль процесса выполнения операций очень важен и должен стать хорошей привычкой, особенно при выполнении такого количества операций.

На этом завершается блок *Multi-Head Self-Attention* в алгоритме энкодера трансформера. Далее идет второй его блок — *Feed Forward.* В рамках этого блока нам предстоит провести сигнал через два нейронных слоя, что мы и сделаем, последовательно вызвав методы прямого прохода каждого нейронного слоя.

```
//--- FeedForward
   if(!m_cFF1.FeedForward(GetPointer(m_cW0)))
      return false;
   if(!m_cFF2.FeedForward(GetPointer(m_cFF1)))
      return false;
```

В заключение алгоритма прямого прохода нам предстоит повторить процедуру нормализации данных. Но теперь мы складываем буферы результатов блоков *Multi-Head Self-Attention* и *Feed Forward*.

```
//--- суммируем с выходом внимания и нормализуем
   if(!m_cOutputs.SumArray(m_cW0.GetOutputs()))
      return false;
   if(!NormlizeBuffer(m_cOutputs, GetPointer(m_cStd), 1))
      return false;
//---
   return true;
  }
```

Процедура нормализации завершает метод прямого прохода. После успешного завершения указанного процесса мы выходим из метода с результатом *true*. Переходим к реализации метода обратного прохода.

5.2.2.2 Методы обратного прохода Multi-Head Self-Attention

Мы уверенно движемся вперед по пути познания. И продолжаем реализацию нашего класса многоголового внимания *Multi-Head Self-Attention*. В предыдущих разделах мы уже реализовали методы инициализации и прямого прохода. Но в основе алгоритма обучения нейронного слоя лежит алгоритм обратного распространения градиента ошибки. Сейчас мы приступаем к реализации методов обратного прохода.

Мы уже говорили, что алгоритм Multi-Head Self-Attention является логическим продолжением технологии *Self-Attention*. Именно поэтому мы и создавали наш класс на базе класса *CNeuronAttention*. И да, все процессы очень похожи. Но все же есть небольшие отличия в реализации многоголового внимания. Для реализации этих отличий мы создали новый класс *CNeuronMHAttention*.

Продвигаясь по пути создания методов класса, давайте посмотрим на реализацию этих отличий в методах алгоритма обратного прохода.

В родительском классе для реализации алгоритма обратного прохода мы переопределили три виртуальных метода:

- *CNeuronAttention::CalcHiddenGradient* — метод расчета градиента ошибки через скрытый слой;
- *CNeuronAttention::CalcDeltaWeights* — метод расчета градиента ошибки до уровня матрицы весов;
- *CNeuronAttention::UpdateWeights* — метод обновления весовых коэффициентов.

Следовательно, для организации обратного прохода многоголового внимания нам также нужно будет переопределить соответствующие методы. Начнем работу с метода распределения градиента ошибки через скрытый слой нейронной сети *CalcHiddenGradient*.

Как и в методе родительского класса, в параметрах метода мы получаем указатель на объект предыдущего нейронного слоя. Именно в его буфер градиентов ошибки нам предстоит записать результат проделываемой работы.

В начале тела метода *CNeuronMHAttention::CalcHiddenGradient* идет ставший уже привычным и неотъемлемым атрибутом любого метода блок проверки указателей на используемые в методе объекты. Здесь мы, как и в аналогичном методе родительского класса, для исключения повторных контролей проверим только указатели на те объекты, к которым первое обращение будет напрямую из этого метода без использования методов внутренних нейронных слоев. Дело в том, что все методы внутренних нейронных слоев имеют аналогичный блок контролей. Вызывая их, мы повторно проверяем передаваемые указатели на объекты. А это дополнительные затраты ресурсов и времени. И если мы не можем отключить проверку в методах вложенных нейронных слоев, то мы исключим явное дублирование контролей в текущем методе.

Сразу надо обратить внимание, что мы исключаем только *явное дублирование*, а не *возможное*. Это тонкая грань, но за ней кроются большие риски.

Явным я называю такое дублирование, которое произойдет в любом случае. Если мы видим такое дублирование, то по возможности оставляем только одну точку контроля перед первым использованием объекта.

Обратите внимание, перед первым обращением к объекту должна быть хотя бы одна точка контроля.

Возможным я называю такое дублирование, которое может произойти при определенных обстоятельствах. В некоторых оно случаях может и не произойти. Мы не исключаем такое дублирование, так как риск критической ошибки при отсутствии контроля превышает возможные выгоды от повышения производительности программы.

```
bool CNeuronMHAttention::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
//--- проверяем актуальность всех объектов
   if(!m_cOutputs || !m_cGradients ||
      m_cOutputs.Total() != m_cGradients.Total())
      return false;
```

После успешного прохождения блока контролей переходим непосредственно к процедуре распределения градиента ошибки. Как вы помните, при прямом проходе на выходе нейронного слоя данные нормализуются. И, конечно, мы должны скорректировать градиент ошибки на производную функции нормализации. В родительском классе мы вывели данную процедуру в отдельный метод *CNeuronAttention::NormlizeBufferGradient*. Сейчас нам достаточно вызвать его с передачей соответствующих параметров.

```
//--- масштабируем градиент на нормализацию
   if(!NormlizeBufferGradient(m_cOutputs, m_cGradients, GetPointer(m_cStd), 1))
      return false;
```

Далее мы проведем градиент ошибки через внутренние нейронные слои блока *Feed Forward*. Это два сверточных слоя: *m_cFF2* и *m_cFF1*. Для проведения градиента через эти нейронные слои мы последовательно вызываем аналогичные методы указанных нейронных слоев. Не забываем проверять результат выполнения операций.

```
//--- проводим градиент ошибки через блок Feed Forward
   if(!m_cFF2.CalcHiddenGradient(GetPointer(m_cFF1)))
      return false;
   if(!m_cFF1.CalcHiddenGradient(GetPointer(m_cW0)))
      return false;
```

После пропуска градиента ошибки через блок *Feed Forward* мы вспоминаем, что перед нормализацией данных на выходе нейронного слоя мы складывали тензоры результатов блоков *Multi-Head Self-Attention* и *Feed Forward*. Следовательно, и градиент ошибки мы должны провести по обоим направлением. Для этого после получения градиента ошибки от блока *Feed Forward* в буфере внутреннего нейронного слоя *m_cW0* мы сложим два тензора.

```
   if(!m_cW0.GetGradients().SumArray(m_cGradients))
      return false;
```

Скорректируем на производную процесса нормализации данных.

```
//--- корректируем градиент на нормализацию
   if(!NormlizeBufferGradient(m_cW0.GetOutputs(), m_cW0.GetGradients(),
                                                  GetPointer(m_cStd), 0))
      return false;
```

Мы продолжаем использование методов внутренних нейронных слоев. Вызовем метод распределения градиента сверточного слоя *m_cW0* и проверим результат выполнения операций.

```
//--- распределения градиента ошибки по головам внимания
   if(!m_cW0.CalcHiddenGradient(GetPointer(m_cAttentionOut)))
      return false;
```

Далее нам предстоит распределить градиент ошибки от конкатенированного результата блока *Multi-Head Self-Attention* к внутренним нейронным слоям *m_cQuerys, m_cKeys* и *m_cValues.* Как вы помните, при прямом проходе путь от указанных внутренних нейронных слоев до *m_cAttentionOut* полностью был воссоздан внутри метода. Аналогично нам предстоит воссоздать и продвижение обратного сигнала.

Раз мы создаем новый поток операций, то, согласно нашей концепции, необходимо организовать два параллельных потока операций. Один — стандартными средствами *MQL5*, а второй — в парадигме многопоточных операций с использованием технологии *OpenCL.*

```
//--- разветвление алгоритма по вычислительному устройству
   if(!m_cOpenCL)
     {
      MATRIX gradients[];
      MATRIX querys[], querys_grad = MATRIX::Zeros(m_iHeads, m_iUnits * m_iKeysSize);
      MATRIX keys[], keys_grad = MATRIX::Zeros(m_iHeads, m_iUnits * m_iKeysSize);
      MATRIX values[], values_grad = MATRIX::Zeros(m_iHeads, m_iUnits * m_iKeysSize);
      MATRIX attention_grad = m_cAttentionOut.GetGradients().m_mMatrix;
```

Как всегда, в данном разделе мы рассмотрим реализацию средствами *MQL5*. К организации многопоточных операций вернемся в одном из последующих разделов.

Итак, вначале мы проведем небольшую подготовительную работу. Как и в прямом проходе, в данном блоке мы организуем работу в разрезе отдельных голов внимания. Так как все данные содержатся в конкатенированных буферах, мы подготовим локальные матрицы и разобьем буферы на отдельные матрицы в соответствии с головами внимания.

```
      if(!m_cQuerys.GetOutputs().m_mMatrix.Vsplit(m_iHeads, querys) ||
         !m_cKeys.GetOutputs().m_mMatrix.Vsplit(m_iHeads, keys) ||
         !m_cValues.GetOutputs().m_mMatrix.Vsplit(m_iHeads, values) ||
         !attention_grad.Reshape(m_iUnits, m_iHeads * m_iKeysSize) ||
         !attention_grad.Vsplit(m_iHeads, gradients))
         return false;
```

Далее создадим цикл с количеством итераций равным числу используемых голов внимания.

```
      for(int head = 0; head < m_iHeads; head++)
        {
```

При прямом проходе значения конкатенированного буфера результатов собираются путем умножения значений тензора результатов нейронного слоя *m_cValues* на соответствующие элементы матрицы коэффициентов зависимости с последующим сложением векторов. Теперь нам предстоит организовать обратный процесс: распределение градиента ошибки по этим двум направлениям.

Сначала мы перенесем градиент ошибки на внутренний нейронный слой *m_cValues*. Но прежде проведем небольшую подготовительную работу.

Для распределения градиента на нейронный слой *m_cValues* необходимо матрицу градиентов ошибки умножить на матрицу коэффициентов зависимости. Следовательно, нам сначала нужно извлечь такую матрицу для анализируемой головы внимания.

Затем перемножаем матрицы и добавляем результат к локальной копии конкатенированной матрицы градиентов слоя *m_cValues.*

```
//--- распределение градиента на Values
   MATRIX score = MATRIX::Zeros(1, m_iUnits * m_iUnits);
   if(!score.Row(m_cScores.m_mMatrix.Row(head), 0) ||
      !score.Reshape(m_iUnits, m_iUnits))
      return false;
   MATRIX temp = (score.Transpose().MatMul(gradients[head])).Transpose();
   if(!temp.Reshape(1, m_iUnits * m_iKeysSize) ||
      !values_grad.Row(temp.Row(0), head))
      return false;
```

После этого перейдем к распределению градиента по второму пути алгоритма через матрицу коэффициентов зависимости на нейронные слои *m_cQuerys* и *m_cKeys.* По существу, нам сначала надо определить градиент ошибки на уровне матрицы коэффициентов зависимости и уже с нее распределить градиент ошибки до уровня указанных внутренних нейронных слоев.

Здесь следует вспомнить, что матрица коэффициентов зависимости нормализуется функцией *Softmax* в разрезе запросов *Query*. Чтобы правильно скорректировать градиент ошибки на производную функции *Softmax* нам необходим как минимум полный вектор градиентов ошибки для значений, участвующих в одной операции нормализации. Записать его мы можем в локальную матрицу.

Что ж, задача понятна, можно приступать к реализации. Для распределения градиента ошибки до матрицы коэффициентов зависимости достаточно умножить полученный градиент на матрицу результатов последнего прямого прохода нейронного слоя *m_cValues*.

После получения вектора градиента ошибки на уровне матрицы коэффициентов зависимости мы должны скорректировать его на производную функцию *Softmax*.

$$Softmax(x_i)' = \begin{cases} i = j, & y_i(1 - y_i) \\ i \neq j, & -y_i y_j \end{cases}$$

Организуем цикл, в котором скорректируем градиент ошибки на производную функции нормализации Softmax.

```
//--- распределение градиента до Score
gradients[head] = gradients[head].MatMul(values[head].Transpose());
//--- корректировка градиента на производную Softmax
for(int r = 0; r < m_iUnits; r++)
  {
   MATRIX ident = MATRIX::Identity(m_iUnits, m_iUnits);
   MATRIX ones = MATRIX::Ones(m_iUnits, 1);
   MATRIX result = MATRIX::Zeros(1, m_iUnits);
   if(!result.Row(score.Row(r), 0))
      return false;
   result = ones.MatMul(result);
   result = result.Transpose() * (ident - result);
   if(!gradients[head].Row(result.MatMul(gradients[head].Row(r)) /
                                                  sqrt(m_iKeysSize), r))
      return false;
  }
```

Следующим этапом мы распределим градиент ошибки на значения результатов нейронных слоев *m_cQuerys* и *m_cKeys.* Но мы не будем сразу записывать значения в буферы данных указанных нейронных слоев. Мы лишь будем накапливать суммы градиентов ошибки в заранее подготовленные матрицы *querys_grad* и *keys_grad*.

Технически мы умножаем скорректированный градиент ошибки на противоположную матрицу. Умножив на матрицу *Keys*, получаем градиент ошибки для *Querys*, и наоборот. Переформатируем полученные матрицы и добавим их в соответствующие локальные матрицы.

```
//--- распределение градиента на Querys и Keys
temp = (gradients[head].MatMul(keys[head])).Transpose();
if(! temp.Reshape(1, m_iUnits * m_iKeysSize) ||
   !querys_grad.Row(temp.Row(0), head))
   return false;
temp = (gradients[head].Transpose().MatMul(querys[head])).Transpose();
if(! temp.Reshape(1, m_iUnits * m_iKeysSize) ||
   !keys_grad.Row(temp.Row(0), head))
   return false;
}
```

После завершения итераций цикла мы получили конкатенированные матрицы градиентов ошибки для всех внутренних слоев. Остается привести матрицы к необходимому формату и скопировать значения в соответствующие буфера данных.

```
      if(!querys_grad.Reshape(m_iHeads * m_iKeysSize, m_iUnits) ||
         !keys_grad.Reshape(m_iHeads * m_iKeysSize, m_iUnits) ||
         !values_grad.Reshape(m_iHeads * m_iKeysSize, m_iUnits))
         return false;
      m_cQuerys.GetGradients().m_mMatrix = querys_grad.Transpose();
      m_cKeys.GetGradients().m_mMatrix = keys_grad.Transpose();
      m_cValues.GetGradients().m_mMatrix = values_grad.Transpose();
     }
   else // Блок OpenCL
     {
      return false;
     }
```

В результате мы провели градиент ошибки до уровня внутренних нейронных слоев. Мы решили поставленную ранее задачу и завершаем блок разделения алгоритма в зависимости от вычислительного устройства. В ветке многопоточных операций мы временно оставим выход из метода с ложным результатом, к которому вернемся позже.

Но мы еще не довели градиент ошибки до предыдущего слоя. Дальнейшее распределение градиента ошибки будем осуществлять с помощью методов внутренних нейронных слоев.

Мы уже заполнили буферы градиентов ошибки всех внутренних слоев. Нам достаточно вызвать метод распределения градиента ошибки через слой, чтобы получить градиент ошибки на уровне исходных данных. Но остается один открытый вопрос: все три внутренних нейронных слоя (*m_cQuerys, m_cKeys, m_cValues*) в качестве исходных данных используют один тензор от предыдущего слоя. Значит, все три слоя должны передать градиент ошибки в буфер предыдущего слоя. К тому же, перед нормализацией результат работы блока *Multi-Head Self-Attention* складывался с тензором исходных данных. Следовательно, это четвертый поток градиента ошибки, который нам надо передать на уровень предыдущего слоя.

Но наши методы передачи градиентов построены таким образом, что при сохранении градиента ошибки в буфер предыдущего слоя данные перезаписываются, удаляя предыдущие значения. Это сделано намеренно, чтобы исключить излишние операции очистки буфера градиентов ошибки перед запуском очередной итерации обратного прохода. Для решения этого вопроса после запуска метода *CalcHiddenGradient* каждого внутреннего нейронного слоя мы будем копировать данные градиентов ошибки в отдельный буфер, где и будем суммировать с ранее накопленными значениями. И тут надо вспомнить, что градиент ошибки на выходе из блока *Multi-Head Self-Attention* уже содержится в буфере градиентов ошибки нейронного слоя *m_cW0*. Казалось бы, в нем и можно было бы накапливать градиент ошибки для предыдущего слоя. Но это ошибочное мнение. Если мы сейчас будем накапливать градиент ошибки в указанном буфере, то это исказит данные при последующем распределении градиента ошибки до матрицы весовых коэффициентов этого слоя. В то же время нам ничто не мешает провести распределение градиента ошибки до матрицы слоя *m_cW0* сейчас. Для этого есть все необходимые данные. Вызываем метод *CalcDeltaWeights* указанного нейронного слоя и далее используем его буфер для накопления суммарного градиента ошибки.

```
//--- перенос градиента ошибки на предыдущий слой
   if(!m_cW0.CalcDeltaWeights(GetPointer(m_cAttentionOut), false))
      return false;
   CBufferType* attention_grad = m_cW0.GetGradients();
   if(!m_cValues.CalcHiddenGradient(prevLayer))
      return false;
   if(!attention_grad.SumArray(prevLayer.GetGradients()))
      return false;
   if(!m_cQuerys.CalcHiddenGradient(prevLayer))
      return false;
   if(!attention_grad.SumArray(prevLayer.GetGradients()))
      return false;
   if(!m_cKeys.CalcHiddenGradient(prevLayer))
      return false;
   if(!prevLayer.GetGradients().SumArray(attention_grad))
      return false;
//---
   return true;
  }
```

Следует обратить внимание на последнюю группу команд. Если при выполнении предыдущих операций мы копировали данные из буфера градиентов предыдущего слоя, то в завершение метода мы наоборот берем суммарный градиент ошибки из буфера внутреннего нейронного слоя и прибавляем его к значениям буферу предыдущего слоя. Ведь именно в буфере предыдущего слоя нам нужно поучить результат. Из него потом методы предыдущего слоя возьмут градиент ошибки и распределят дальше по нейронной сети.

На этом задача, поставленная перед данным методом, решена. Завершаем работу метода с положительным результатом.

Далее мы будем работать над еще двумя методами, которые продолжат выполнение алгоритма обратного распространения ошибки в данном классе.

После распределения ошибки по всем нейронным слоям нашей сети нужно довести градиент ошибки до уровня каждого весового коэффициента. Наш класс *CNeuronMHAttention* не содержит отдельного буфера для матрицы весовых коэффициентов. Все обучаемые параметры заключены во внутренних нейронных слоях. Поэтому единственное, что нам нужно сделать в методе распределения градиента ошибки до матрицы весовых коэффициентов *CalcDeltaWeights*, — это последовательно вызвать аналогичный метод всех внутренних слоев. При этом, конечно, не забываем проверить результат выполнения операций.

Напомню, что в предыдущем методе мы уже передали градиент ошибки до матрицы весов внутреннего слоя *m_cW0*. Необходимо исключить его из данной итерации.

```
bool CNeuronMHAttention::CalcDeltaWeights(CNeuronBase *prevLayer, bool read)
  {
//--- вызываем аналогичный метод для всех внутренних слоев
   if(!m_cFF2.CalcDeltaWeights(GetPointer(m_cFF1), false))
      return false;
   if(!m_cFF1.CalcDeltaWeights(GetPointer(m_cW0), false))
      return false;
   if(!m_cQuerys.CalcDeltaWeights(prevLayer, false))
      return false;
   if(!m_cKeys.CalcDeltaWeights(prevLayer, false))
      return false;
   if(!m_cValues.CalcDeltaWeights(prevLayer, read))
      return false;
//---
   return true;
  }
```

После распределения градиентов ошибки до матриц весовых коэффициентов остается только обновить весовые коэффициенты наших внутренних нейронных слоев. Этот функционал возлагается на метод *UpdateWeights*. Несмотря на всю сложность работы самого класса, метод обновления матриц весовых коэффициентов оказывается очень кратким и простым в построении. В этом нам помогло именно наследование объектов.

Напомню, свой класс *CNeuronMHAttention* мы создавали наследником от класса *CNeuronAttention*. Мы добавили лишь один объект внутреннего нейронного слоя *m_cW0*. В процессе выполнения операций метода *UpdateWeights* используемых нами сверточных нейронных слоев, все операции выполняются только с элементами внутри объекта, без обращения к данным других объектов. Так сказать, «варятся в собственном соку». Поэтому мы можем вызвать аналогичный метод родительского класса, в котором этот процесс уже реализован для наследуемых объектов. После успешного выполнения метода родительского класса нам остается обновить матрицу коэффициентов лишь одного внутреннего нейронного слоя *m_cW0*.

```
bool CNeuronMHAttention::UpdateWeights(int batch_size, TYPE learningRate,
                                       VECTOR &Beta, VECTOR &Lambda)
  {
//--- вызов метода родительского класса
   if(!CNeuronAttention::UpdateWeights(batch_size, learningRate, Beta, Lambda))
      return false;
//--- вызываем аналогичный метод для всех внутренних слоев
   if(!m_cW0.UpdateWeights(batch_size, learningRate, Beta, Lambda))
      return false;
//---
   return true;
  }
```

Конечно, мы проверим результат выполнения всех операций и вернем логическое значение их выполнения вызвавшей программе.

Таким образом, приближаемся к завершению работы над классом реализации технологии *Multi-Head Self-Attention*. По существу, мы уже реализовали весь алгоритм стандартными средствами *MQL5*. Можно даже создать скрипт и протестировать его работу. Но нам еще предстоит дополнить наш класс методами работы с файлами.

5.2.2.3 Методы работы с файлами

Мы уже далеко продвинулись в работе над реализацией алгоритма *Multi-Head Self-Attention*. В предыдущих разделах мы уже реализовали стандартными средствами *MQL5* операции прямого и обратного прохода нашего класса *CNeuronMHAttention.* Теперь для возможности полноценного его использования в наших моделях необходимо дополнить его методами работы с файлами. Как бы вам ни казалось, корректная работа этих методов для промышленного использования не менее важна корректной работы методов прямого и обратного проходов.

Да, мы можем создать модель и протестировать ее работу и без сохранения результатов обучения. Но для проведения повторного теста нам придется заново обучать нашу модель. А в процессе промышленной эксплуатации нам бы не хотелось повторять процесс обучения каждый раз. Напротив, довольно часто тратятся большие усилия на разработку и обучения модели на больших наборах данных, что позволяет построить по-настоящему рабочую модель. При этом ожидается, что в процессе промышленной эксплуатации достаточно будет запустить модель, и она будет полностью готова к функционированию на реальных данных. Поэтому, подходя к работе над методами работы с файлами, мы должны продумать их функционал таким образом, чтобы на выходе мы полностью восстановили состояние модели с минимальными затратами. Что ж, мы уже не раз выполняли данную работу и давайте воспользуемся уже отработанным алгоритмом.

Вначале посмотрим на структуру нашего класса многоголового внимания *CNeuronMHAttention*.

```
class CNeuronMHAttention    :  public CNeuronAttention
  {
protected:
   CNeuronConv       m_cW0;
   int               m_iHeads;

public:
                     CNeuronMHAttention(void);
                    ~CNeuronMHAttention(void);
   //---
   virtual bool      Init(const CLayerDescription *desc) override;
   virtual bool      SetOpenCL(CMyOpenCL *opencl) override;
   virtual bool      FeedForward(CNeuronBase *prevLayer) override;
   virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer, bool read) override;
   virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                   VECTOR &Beta, VECTOR &Lambda) override;
   //--- методы работы с файлами
   virtual bool      Save(const int file_handle) override;
   virtual bool      Load(const int file_handle) override;
   //--- метод идентификации объекта
   virtual int       Type(void) override const { return(defNeuronMHAttention);  }
  };
```

На первый взгляд нет ничего сложного. В теле класса объявляется лишь один сверточный слой *m_cW0* и одна переменная *m_iHeads*, указывающая на количество используемых голов внимания. Основное количество объектов наследуется от родительского класса *CNeuronAttention*. Мы уже создали аналогичный метод при работе над родительским классом, а сейчас можем им воспользоваться. Я советую еще раз заглянуть в метод родительского класса

CNeuronAttention::Save и убедиться, что в нем есть сохранение всех необходимых нам данных. Только после этого можно приступать к работе над методом сохранения данных текущего класса. На этот раз здесь все действительно очень просто.

В параметрах метод *CNeuronMHAttention::Save* получает хендл файла для записи данных. В теле метода мы сразу передаем полученный хендл в аналогичный метод родительского класса, в котором уже реализованы все контроли. Помимо контролей в методе родительского класса реализовано сохранение унаследованных объектов и их данных. Таким образом, проверяя результат работы метода родительского класса, мы сразу получаем консолидированный результат прохождения блока контролей и сохранения унаследованных объектов. Нам остается лишь сохранить количество используемых голов внимания и данные сверточного слоя *m_cW0*.

```
bool CNeuronMHAttention::Save(const int file_handle)
  {
//--- вызов метода родительского класса
   if(!CNeuronAttention::Save(file_handle))
      return false;
//--- сохраняем константы
   if(FileWriteInteger(file_handle, m_iHeads) <= 0)
      return false;
//--- вызываем аналогичный метод для всех внутренних слоев
   if(!m_cW0.Save(file_handle))
      return false;
//---
   return true;
  }
```

Метод загрузки данных *CNeuronMHAttention::Load* строится по правилу загрузки данных из файла в строгом соответствии с последовательностью их записи. Поэтому в теле метода полученный в параметрах хендл файла мы сразу передаем в аналогичный метод родительского класса и проверяем результат его работы.

```
bool CNeuronMHAttention::Load(const int file_handle)
  {
//--- вызов метода родительского класса
   if(!CNeuronAttention::Load(file_handle))
      return false;
```

После выполнения операций метода родительского класса мы считываем из файла количество используемых голов внимание и данные внутреннего сверточного слоя *m_cW0*. С загрузкой константы все очень просто: мы просто читаем значение из файла и сохраняем в нашу переменную *m_iHeads*. Но перед обращением к методу загрузки мы должны проверить тип загружаемого объекта. Только при совпадении типов объектов мы вызываем метод загрузки данных и проверяем результат выполнения операций.

```
   m_iHeads = FileReadInteger(file_handle);
   if(CheckPointer(m_cW0) == POINTER_INVALID)
     {
      m_cW0 = new CNeuronConv();
      if(CheckPointer(m_cW0) == POINTER_INVALID)
         return false;
     }
   if(FileReadInteger(file_handle)!=defNeuronConv ||
      !m_cW0.Load(file_handle))
      return false;
```

Ожидается, что после успешного выполнения операций родительского класса мы получим полностью восстановленные унаследованные объекты. Но давайте вспомним, что объекты мы унаследовали, а инициализировали их в соответствующем методе этого класса и с параметрами отличными от родительского класса. Ведь в этом классе практически для всех объектов мы делали поправку на количество используемых голов внимания. При этом в методе загрузки данных родительского класса мы не только загружаем данные объектов из файла, но и инициализируем несохраненные объекты. Это такие объекты, данные которых используются только в рамках одной итерации прямого и обратного проходов.

Поэтому возвращаемся к методу родительского класса и еще раз критически оцениваем все операции. И тут внимания нужно обратить на следующие строки кода.

```
bool CNeuronAttention::Load(const int file_handle)
  {
  ......
   m_iUnits = FileReadInteger(file_handle);
  ......
   if(!m_cScores.BufferInit(m_iUnits, m_iUnits, 0))
      return false;
  ......
//---
   return true;
  }
```

В них инициализируется буфер матрицы коэффициентов зависимости *m_cScores*. Как можно заметить, инициализация проходит нулевыми значениями в размере достаточном только для одной головы внимания. А это не соответствует требованиям нашего алгоритма *Multi-Head Self-Attention*. Логично будет добавить в метод загрузки нашего класса повторную инициализацию буфера с приданием ему нужного размера.

```
//--- инициализируем Scores
   if(!m_cScores.BufferInit(m_iHeads, m_iUnits * m_iUnits))
      return false;
//---
   return true;
  }
```

После успешного завершения выполнения всех операций мы выходим из метода с положительным результатом.

На этом мы завершаем работу стандартными средствами *MQL5* над классом *CNeuronMHAttention,* в котором мы реализовали алгоритм *Multi-Head Self-Attention*. В следующем разделе мы дополним

его функционал возможностью совершения многопоточных операций с помощью технологии *OpenCL*.

5.2.3 Организация параллельных вычислений Multi-Head Self-Attention

Мы продолжаем наше уверенное движение вперед по пути познания и строительства библиотеки для создания моделей машинного обучения в среде *MQL5*. В данном разделе мы планируем завершить работу еще над одним классом нейронных слоев CNeuronMHAttention, в котором реализован алгоритм *Multi-Head Self-Attention*. В предыдущих разделах мы уже полностью реализовали алгоритм стандартными средствами *MQL5*. Сейчас дополним его функционал возможностью использования технологии *OpenCL* для организации процесса вычислений в многопоточном режиме с задействованием ресурсов *GPU*.

Для каждого из рассмотренных ранее нейронных слоев мы уже проводили подобную работу. И, конечно, вместе мы справимся и с этой задачей. Напомню общий алгоритм построения данного процесса. Сначала мы создаем программу *OpenCL*. Затем дополняем код основной программы функционалом вызова данной программы и передачи необходимых данных в обоих направлениях. Нам нужно будет отправить исходные данные в программу до ее выполнения и посчитать результаты работы после ее выполнения.

Как обычно, начинаем с создания программы *OpenCL*. И сразу в голове всплывает вопрос: а надо ли нам создавать новую программу? Для чего нам создавать новые кернелы? Ответ, конечно, очевиден: для реализации функционала. Но давайте вспомним, что наш класс мы наследовали от похожего класса реализации алгоритма *Self-Attention*. Мы не раз говорили о преемственности указанных алгоритмов. Можем ли мы использовать созданные ранее кернелы для реализации процессов в этом классе?

На самом деле, учитывая схожесть процессов, нам было бы выгоднее использовать одни кернелы для обоих реализаций. Во-первых, это сокращает количество *OpenCL*-объектов, число которых в системе тоже не безгранично. Во-вторых, всегда удобнее поддерживать и оптимизировать один объект, чем копировать общие блоки между несколькими схожими объектами. Будь-то кернелы или классы.

Но как нам это реализовать? Созданные ранее кернелы работают в рамках одной головы внимания. Конечно, мы можем на стороне основной программы осуществить копирование данных в отдельные буферы и последовательно вызывать кернелы для каждой головы внимания. Такой подход возможен, но он иррациональный. Излишнее копирование данных само по себе не самый лучший вариант решения. А последовательный вызов кернелов для каждой головы внимания вообще не дает возможности одновременного расчета всех голов внимания в параллельных потоках.

На самом деле, у нас есть возможность использования ранее созданных кернелов и без излишнего копирования, но с небольшой доработкой.

Первое, что мы уже сделали в самом начале при создании класса, так это полное использование конкатенированных буферов данных. То есть все наши буферы данных содержат данные сразу всех голов внимания. Передавая данные в память контекста, мы передаем данные всех голов внимания. А значит, на стороне *OpenCL* мы можем параллельно работать со всеми головами внимания. Нам лишь надо правильно определить смещение в буфере данных до нужных значений. И это те изменения, которые мы должны внести в кернел.

Чтобы определить это смещение, нам необходимо понимание общего количества используемых голов внимания и порядковый номер рабочей головы внимания. Если общее количество мы

можем передать в параметрах, то порядковый номер текущей — нет. Для организации передачи таких данных нам потребовалось бы создавать цикл с последовательным вызовом кернела для каждой головы внимания, а мы пытаемся уйти от этого.

Давайте вспомним, как организована функция постановки кернела в очередь выполнения. У функции *CLExecute* есть параметр *work_dim*, отвечающий за размерность пространства задач. Также функция в параметрах принимает динамический массив *global_work_size[]*, в котором указывается общее количество выполняемых задач в каждом измерении.

```
bool  CLExecute(
// хендл на кернел OpenCL программы
   int          kernel,
// размерность пространства задач
   uint         work_dim,
// начальное смещение в пространстве задач
   const uint&  global_work_offset[],
// общее количество задач
   const uint&  global_work_size[]
   );
```

И если раньше мы использовали только одно измерение, то сейчас мы можем задействовать два. Одно по-прежнему будем использовать для перебора элементов последовательности, а второе — для перебора голов внимания.

Что ж, решение найдено, и мы можем приступать к реализации. Но тут есть еще один вопрос: создавать новый кернел или нет. Все говорит за изменение ранее созданного. Однако в этом случае после завершения работы нам придется сделать шаг назад и скорректировать методы класса *CNeuronAttention*. Иначе получим критическую ошибку при попытке запуска кернела.

Для себя я решил вносить изменения в ранее созданный кернел и методы основной программы. Вы же можете другой вариант.

Приступаем к работе. Посмотрим на внесенные в кернел прямого прохода изменения.

В теле кернела запрашиваем идентификаторы запущенного потока и общее количество потоков в двух измерениях. Первое измерение укажет номер обрабатываемого запроса и длину последовательности. Второе измерение укажет номер активной головы внимания.

Тут же мы определяем смещение до начала анализируемого вектора в тензоре запросов и матрице коэффициентов зависимости.

```
const int q = get_global_id(0);
const int units = get_global_size(0);
const int h = get_global_id(1);
const int heads = get_global_size(1);
int shift_query = key_size * (q * heads + h);
int shift_scores = units * (q * heads + h);
```

Как вы понимаете, от предыдущей версии кернел отличается наличием второго измерения, учитывающего головы внимания. Соответственно, расчет смещения также посчитан с учетом многоголового внимания.

Далее организуем систему из двух вложенных циклов для расчета одного вектора матрицы коэффициентов зависимости. Это связано с тем, что для расчета одного элемента

последовательности на выходе блока внимания нам потребуется целый вектор матрицы коэффициентов зависимости. Разумеется, в рамках одной головы внимания.

Также перед запуском системы циклов подготовим локальную переменную *summ* для суммирования всех значений вектора. Данная сумма нам потребуется для последующей нормализации значений вектора.

Внешний цикл имеет количество итераций равное числу элементов последовательности. Он сразу укажет нам анализируемый элемент в тензоре ключей *Key* и номер столбца в матрице коэффициентов зависимости. В теле цикла мы определим смещение в тензоре ключей до начала вектора анализируемого элемента последовательности и подготовим переменную для подсчета результата умножения двух векторов.

Во вложенном цикле с числом итераций равным размеру вектора ключей мы выполним операцию умножения вектора запросов на вектор ключей.

После завершения итераций вложенного цикла мы возьмем экспоненту от полученного результата умножения векторов, полученное значение запишем в тензор матрицы коэффициентов зависимости и прибавим к нашей сумме значений вектора.

```
TYPE summ = 0;
for(int s = 0; s < units; s++)
  {
   TYPE score = 0;
   int shift_key = key_size * (s * heads + h);
   for(int k = 0; k < key_size; k ++)
      score += querys[shift_query + k] * keys[shift_key + k];
   score = exp(score / sqrt((TYPE)key_size));
   summ += score;
   scores[shift_scores + s] = score;
  }
```

После завершения всех итераций системы цикла мы получим вектор с посчитанными, но не нормализованными коэффициентами зависимости одного вектора запроса ко всем векторам ключей. Для завершения процесса нормализации вектора нам необходимо разделить содержимое вектора на сумму всех его значений, которую мы предусмотрительно собрали в переменной *summ.*

Для выполнения этой операции создадим еще один цикл с количеством итераций равным количеству элементов в последовательности.

```
for(int s = 0; s < units; s++)
   scores[shift_scores + s] /= summ;
```

Как можно заметить, данный блок отличается от предыдущей реализации только в части расчета смещение элементов в тензорах. Теперь, когда у нас есть нормализованный вектор зависимости одного запроса ко всем элементам последовательности тензора ключей, мы можем посчитать взвешенный вектор одного элемента последовательности на выходе одной головы внимания. Для этого создадим систему из двух вложенных циклов.

Вначале определим смещение в тензоре результатов до начала вектора анализируемого элемента.

Затем создадим внешний цикл по числу элементов в векторе результатов. В теле цикла сначала подготовим переменную для накопительного подсчета значения одного элемента вектора. Создадим вложенный цикл с числом итераций равным числу элементов последовательности. В нем будем перебирать все элементы тензора значений. В каждом векторе описания элемента будем брать по одному значению, соответствующему счетчику итераций внешнего цикла, и умножать его на элемент нормализованного вектора коэффициентов зависимости в соответствии с счетчиком итераций вложенного цикла. После завершения полного цикла итераций вложенного цикла в нашей переменной *query* будет собрано одно значение вектора описания анализируемого элемента последовательности тензора результатов блока внимания. Его мы запишем в соответствующий элемент буфера результатов работы кернела.

```
  shift_query = window * (q * heads + h);
  for(int i = 0; i < window; i++)
    {
     TYPE query = 0;
     for(int v = 0; v < units; v++)
        query += values[window * (v * heads + h) + i] * scores[shift_scores + v];
     outputs[shift_query + i] = query;
    }
}
```

После завершения итераций внешнего цикла в буфере тензора результатов мы получим целый вектор описания одного элемента последовательности.

Как видите, в результате операций одного кернела мы получаем один вектор описания элемента последовательности тензора результатов одной головы внимания. А для расчета полного тензора нам необходимо запустить пул задач в размере произведения количества элементов последовательности на количество голов внимания. Это мы и делаем, запуская кернел в двухмерном пространстве задач.

По существу, чтобы перевести кернел из плоскости одноголового внимания в плоскость многоголового, нам было достаточно организовать запуск кернела в двухмерном пространстве и скорректировать расчет смещения в буферах данных.

Проведем аналогичную работу с кернелами обратного прохода. Как вы помните, в блоке *Self-Attention,* в отличие от реализации других нейронных слоев, распределение градиента ошибки через внутреннее пространство скрытого нейронного слоя мы организовали двумя последовательными кернелами. И нам необходимо «перевести на рельсы» многоголового внимания оба кернела. Но давайте смотреть по порядку.

Первым мы разберем кернел *AttentionCalcScoreGradient*. Параметры кернела остаются без изменений. Здесь все те же буферы данных и одна константа размера вектора описания одного элемента.

```
__kernel void AttentionCalcScoreGradient(__global TYPE *scores,
                                         __global TYPE *scores_grad,
                                         __global TYPE *values,
                                         __global TYPE *values_grad,
                                         __global TYPE *outputs_grad,
                                         __global TYPE *scores_temp,
                                         int window)
  {
```

В теле кернела, как и в кернеле прямого прохода, мы добавляем получение идентификации потока во втором измерении и соответствующим образом меняем расчет смещения в буферах данных.

```
const int q = get_global_id(0);
const int units = get_global_size(0);
const int h = get_global_id(1);
const int heads = get_global_size(1);
int shift_value = window * (q * heads + h);
int shift_score = units * (q * heads + h);
```

Алгоритм кернела мы не меняем. Как и при реализации алгоритма *Self-Attention,* логически кернел можно разбить на два блока.

В первом мы распределяем градиент ошибки на тензор значений *Values*. Здесь мы создаем систему из двух вложенных циклов. Внешний цикл будет иметь число итераций, равное размеру вектора описания одного элемента последовательности в тензоре значений. Сразу же в теле цикла создаем локальную переменную для сбора градиента ошибки анализируемого элемента.

Надо понимать, что при прямом проходе каждый элемент последовательности тензора значений оказывает посильное влияние на значение каждого элемента последовательности тензора результатов. И сила этого влияния определяется соответствующим столбцом матрицы коэффициентов зависимости, в котором каждая строка соответствует одному элементу последовательности тензора результатов. Следовательно, для получения вектора градиента ошибки для одного элемента последовательности тензора значений нам нужно умножить соответствующий столбец матрицы коэффициентов зависимости на тензор градиентов ошибки на уровне результатов бока внимания.

$$G_{Value} = Score^T * G_{Self-Attention}$$

Для выполнения этой операции мы и организовываем вложенный цикл с числом итераций равным числу элементов в последовательности. В теле данного цикла мы перемножим два вектора, а результат запишем в соответствующий элемент буфера градиентов ошибки тензора значений.

```
//--- Распределение градиента на Values
   for(int i = 0; i < window; i ++)
     {
      TYPE grad = 0;
      for(int g = 0; g < units; g++)
         grad += scores[units * (g * heads + h) + q] *
                 outputs_grad[window * (g * heads + h) + i];
      values_grad[shift_value + i] = grad;
     }
```

Здесь мы также внесли изменения только в части определения смещения до анализируемых элементов в буферах данных.

Второй блок данного кернела отвечает за распределение градиента до уровня матрицы коэффициентов зависимости. Вначале мы создадим систему из двух вложенных циклов и посчитаем градиент ошибки для одной строки матрицы коэффициентов зависимости. Здесь есть один очень важный момент. Градиент ошибки мы считаем именно для строки матрицы, а не столбца. Тонкость в том, что нормализация матрицы функцией *Softmax* осуществлялась именно построчно, поэтому и корректировать на производную от *Softmax* мы тоже должны построчно. А чтобы определить градиент ошибки для одной строки матрицы, надо взять соответствующий вектор из тензора градиентов ошибки на уровне результатов блока внимания и умножить на тензор ключей соответствующей головы внимания.

$$G_{Score} = G_{Self-Attention} * V^T$$

Для выполнения операции умножения организуем вложенный цикл.

```
//--- Распределение градиента на Score
   for(int k = 0; k < units; k++)
     {
      TYPE grad = 0;
      for(int i = 0; i < window; i++)
         grad += outputs_grad[shift_value + i] *
                 values[window * (k * heads + h) + i];
      scores_temp[shift_score + k] = grad;
     }
```

После выполнения полного цикла итераций нашей системы циклов в тензоре мы получим одну строку градиентов ошибки для матрицы коэффициентов зависимости. И прежде чем передать градиент ошибки дальше, необходимо его скорректировать на производную функции *Softmax*.

```
//--- Корректируем на производную Softmax
   for(int k = 0; k < units; k++)
     {
      TYPE grad = 0;
      TYPE score = scores[shift_score + k];
      for(int i = 0; i < units; i++)
         grad += scores[shift_score + i] *
                 ((int)(i == k) - score) * scores_temp[shift_score + i];
      scores_grad[shift_score + k] = grad;
     }
  }
```

Результат выполнения операций записываем в соответствующие элементы тензора градиентов ошибки.

На этом завершается работа с первым кернелом алгоритма обратного распространения ошибки. Как вы могли заметить, изменения коснулись только определения смещения в буферах данных и дополнительным измерением пространства задач.

Переходим ко второму кернелу алгоритма обратного распространения ошибки *AttentionCalcHiddenGradient*. В нем нам предстоит распределить градиент ошибки от матрицы коэффициентов зависимости до буферов внутренних нейронных слоев *m_cQuerys* и *m_cKeys.*

С точки зрения математики, операция не сложная. Градиент ошибки на уровне матрицы коэффициентов зависимости мы уже определили в предыдущем кернеле. Теперь нам нужно умножить матрицу коэффициентов зависимости на противоположный тензор.

Как и в предыдущем кернеле, шапка и параметры кернела абсолютно не изменились. Здесь мы видим тот же набор буферов и параметров.

```
__kernel void AttentionCalcHiddenGradient(__global TYPE *querys,
                                          __global TYPE *querys_grad,
                                          __global TYPE *keys,
                                          __global TYPE *keys_grad,
                                          __global TYPE *scores_grad,
                                          int key_size)
  {
```

В теле кернела мы сразу идентифицируем поток в двух измерениях задач. Как вы понимаете, второе измерение мы добавили для идентификации активной головы внимания. Соответствующим образом изменяем смещение в буферах градиентов до анализируемых элементов последовательности.

```
   const int q = get_global_id(0);
   const int units = get_global_size(0);
   const int h = get_global_id(1);
   const int heads = get_global_size(1);
   int shift_query = key_size * (q * heads + h);
   int shift_score = units * (q * heads + h);
```

Как уже сказано выше, в теле кернела нам предстоит распределить градиент ошибки на два внутренних нейронных слоя из одного источника. Для распределения градиента ошибки используется один и тот же алгоритм в обоих направлениях. Да и оба вектора получателя имеют одинаковый размер. Все это позволяет нам осуществлять расчет градиента ошибки для обоих

тензоров параллельно в теле одной системы циклов. Количество итераций во внешнем цикле равно размеру вектора, для которого мы рассчитываем градиент ошибки. В его теле мы подготовим переменные для накопления градиентов ошибки и создаем вложенный цикл с числом итераций равным количеству элементов в последовательности. В теле вложенного цикла мы одновременно считаем значения от произведения двух пар векторов.

```
//--- Распределение градиента на Querys и Keys
   const TYPE k = 1 / sqrt((TYPE)key_size);
//---
   for(int i = 0; i < key_size; i++)
     {
      TYPE grad_q = 0;
      TYPE grad_k = 0;
      for(int s = 0; s < units; s++)
        {
         grad_q += keys[key_size * (s * heads + h) + i] *
                   scores_grad[shift_score + s];
         grad_k += querys[key_size * (s * heads + h) + i] *
                   scores_grad[units * (s * heads + h) + q];
        }
      querys_grad[shift_query + i] = grad_q * k;
      keys_grad[shift_query + i] = grad_k * k;
     }
  }
```

После выхода из вложенного цикла в каждой переменной будет по одному значению для векторов градиентов ошибки искомых тензоров. Запишем их в соответствующие элементы тензоров. После выполнения полного числа итераций системы циклов мы получаем два искомых вектора градиентов ошибки.

Завершаем работу с кернелами программы *OpenCL*. Итак, мы лишь сделали, так сказать, «косметические» правки в кернелах алгоритма *Self-Attention* для их перевода в область многоголового внимания.

Теперь нам предстоит дополнить основную программу функционалом вызова данных кернелов из методов как класса *CNeuronAttention*, так и класса *CNeuronMHAttention*. Обычно мы начинаем эту работу с создания констант для работы с кернелами. Но в данном случае константы уже созданы.

Далее мы создавали кернелы в контексте *OpenCL*. Но сейчас мы не создавали новые кернелы. А те, которые мы немного скорректировали, уже объявлены в теле основной программы. Поэтому мы пропускаем и этот шаг.

Переходим к внесению изменений непосредственно в методы классов. Чтобы новые кернелы работали в классе *CNeuronAttention*, мы добавляем второй элемент в массивы смещения и пространства задач. Для смещения укажем 0 в обоих измерениях. Для пространства задач первое значение оставляем без изменения, во второй элемент массива внесем 1 (используется одна голова внимания). Также при постановке кернела в очередь выполнения укажем двухмерность пространства задач.

```
      int off_set[] = {0, 0};
      int NDRange[] = {m_iUnits, 1};
      if(!m_cOpenCL.Execute(def_k_AttentionFeedForward, 2, off_set, NDRange))
         return false;
```

После этого мы можем полноценно использовать обновленный кернел прямого прохода.

Такие несложные манипуляции делаем для вызова всех трех кернелов в методах класса *CNeuronAttention.*

На этом мы восстановили работоспособность методов класса *CNeuronAttention*, в котором реализован алгоритм *Self-Attention*. На стороне основной программы изменений тоже немного.

Переходим к работе над нашим классом *CNeuronMHAttention* с реализацией алгоритма *Multi-Head Self-Attention*. Как обычно, начнем работу с метода прямого прохода. Прежде чем поставить кернел в очередь выполнения операций, необходимо провести подготовительную работу. В первую очередь проверяем наличие необходимых буферов в памяти контекста *OpenCL*.

```
bool CNeuronMHAttention::FeedForward(CNeuronBase *prevLayer)
  {
   ......
//--- разветвление алгоритма по вычислительному устройству
   MATRIX out;
   if(!m_cOpenCL)
     {
   ......
     }
   else // блок OpenCL
     {
      //--- проверяем буфера данных
      if(m_cQuerys.GetOutputs().GetIndex() < 0)
         return false;
      if(m_cKeys.GetOutputs().GetIndex() < 0)
         return false;
      if(m_cValues.GetOutputs().GetIndex() < 0)
         return false;
      if(m_cScores.GetIndex() < 0)
         return false;
      if(m_cAttentionOut.GetOutputs().GetIndex() < 0)
         return false;
```

После проверки всех необходимых буферов мы передаем указатели на буферы в параметры кернела. Туда же мы передаем константы, необходимые для работы кернела.

Обратите внимание, что предавая параметры кернелу, мы дважды указали переменную *m_iKeysSize*, в которой содержится размер вектора ключей одного элемента последовательности. Мы указали его как для параметра размера вектора ключей, так и для параметра размера вектора значений. Два параметра в кернеле — вынужденная мера. При использовании одной головы внимания для размера вектора значений мы должны были бы указать размер вектора исходных данных. Это требование алгоритма *Self-Attention.* Но при использовании технологии многоголового внимания матрица *W0* позволяет нам использовать различные варианты размера вектора значений.

```
   //--- передача параметров кернелу
   if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward,
                                def_attff_keys, m_cKeys.GetOutputs().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward,
                    def_attff_outputs, m_cAttentionOut.GetOutputs().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward,
                           def_attff_querys, m_cQuerys.GetOutputs().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward,
                                      def_attff_scores, m_cScores.GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionFeedForward,
                           def_attff_values, m_cValues.GetOutputs().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgument(def_k_AttentionFeedForward,
                                               def_attff_key_size, m_iKeysSize))
      return false;
   if(!m_cOpenCL.SetArgument(def_k_AttentionFeedForward,
                                                 def_attff_window, m_iKeysSize))
      return false;
```

На этом заканчивается подготовительная работа, и мы переходим к организации процедуры запуска кернела. Для этого укажем размер пространства задач в двух измерениях. В первом измерении укажем размер последовательности, а во втором — количество голов внимания. Вызовем метод поставки кернела в очередь выполнения.

```
   //--- постановка кернела в очередь выполнения
   int off_set[] = {0, 0};
   int NDRange[] = {m_iUnits, m_iHeads};
   if(!m_cOpenCL.Execute(def_k_AttentionFeedForward, 2, off_set, NDRange))
      return false;
  }
```

Здесь мы заканчиваем работу над методом прямого прохода и переходим к методу распределения градиента ошибки через скрытый слой *CalcHiddenGradient*. Как вы помните, выше для реализации процесса этого метода мы подготовили два кернела, которые нам предстоит последовательно запустить. Первым мы будем запускать кернел распределения градиента ошибки до матрицы коэффициентов зависимости *AttentionCalcScoreGradient*.

Алгоритм проведения подготовительной работы и запуска кернела аналогичен тому, что мы использовали выше при запуске кернела прямого прохода.

```
bool CNeuronMHAttention::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
//--- разветвление алгоритма по вычислительному устройству
   if(!m_cOpenCL)
     {
   ......
   // блок MQL5
     }
   else // блок OpenCL
     {
      //--- проверяем буферы данных
      if(m_cValues.GetOutputs().GetIndex() < 0)
         return false;
      if(m_cValues.GetGradients().GetIndex() < 0)
         return false;
      if(m_cScores.GetIndex() < 0)
         return false;
      if(m_cAttentionOut.GetGradients().GetIndex() < 0)
         return false;
      if(m_cScoreGrad < 0)
         return false;
      if(m_cScoreTemp < 0)
         return false;
```

После проверки буферов мы передаем указатели на них и необходимые константы в параметры кернела.

```
      //--- передача параметров кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
               def_attscr_outputs_grad, m_cAttentionOut.GetGradients().GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
                                      def_attscr_scores, m_cScores.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
                                         def_attscr_scores_grad, m_cScoreGrad))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
                                         def_attscr_scores_temp, m_cScoreTemp))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
                           def_attscr_values, m_cValues.GetOutputs().GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionScoreGradients,
                     def_attscr_values_grad, m_cValues.GetGradients().GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_AttentionScoreGradients,
                                                def_attscr_window, m_iKeysSize))
         return false;
```

Осуществляем постановку кернела в очередь выполнения операций. Как и в случае прямого прохода, мы создаем двухмерное пространство задач. В первом измерении мы указываем

количество анализируемых элементов в последовательности, а во втором — количество голов внимания.

```
//--- постановка кернела в очередь выполнения
int off_set[] = {0, 0};
int NDRange[] = {m_iUnits, m_iHeads};
if(!m_cOpenCL.Execute(def_k_AttentionScoreGradients, 2, off_set, NDRange))
   return false;
```

Тут же начинаем подготовительную работу перед запуском второго кернела. Проверяем буфера данных в памяти контекста *OpenCL*. Проверке подлежат только те буферы, которые мы не проверили при запуске первого кернела.

```
//--- проверка буферов данных
if(m_cQuerys.GetOutputs().GetIndex() < 0)
   return false;
if(m_cQuerys.GetGradients().GetIndex() < 0)
   return false;
if(m_cKeys.GetOutputs().GetIndex() < 0)
   return false;
if(m_cKeys.GetGradients().GetIndex() < 0)
   return false;
```

Передадим указатели на буферы данных в параметры второго кернела. Туда же добавим необходимые константы.

```
//--- передача аргументов кернелу
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                           def_atthgr_keys, m_cKeys.GetOutputs().GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                      def_atthgr_keys_grad, m_cKeys.GetGradients().GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                         def_atthgr_querys, m_cQuerys.GetOutputs().GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                  def_atthgr_querys_grad, m_cQuerys.GetGradients().GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_AttentionHiddenGradients,
                                      def_atthgr_scores_grad, m_cScoreGrad))
   return false;
if(!m_cOpenCL.SetArgument(def_k_AttentionHiddenGradients,
                                          def_atthgr_key_size, m_iKeysSize))
   return false;
```

После проведения подготовительной работы мы вызываем метод постановки кернела в очередь выполнения задач. Обратите внимание, что на этот раз мы не создаем новые массивы с указанием пространства задач, ведь оно у нас не изменилось, и мы можем воспользоваться уже существующими массивами от запуска предыдущего кернела.

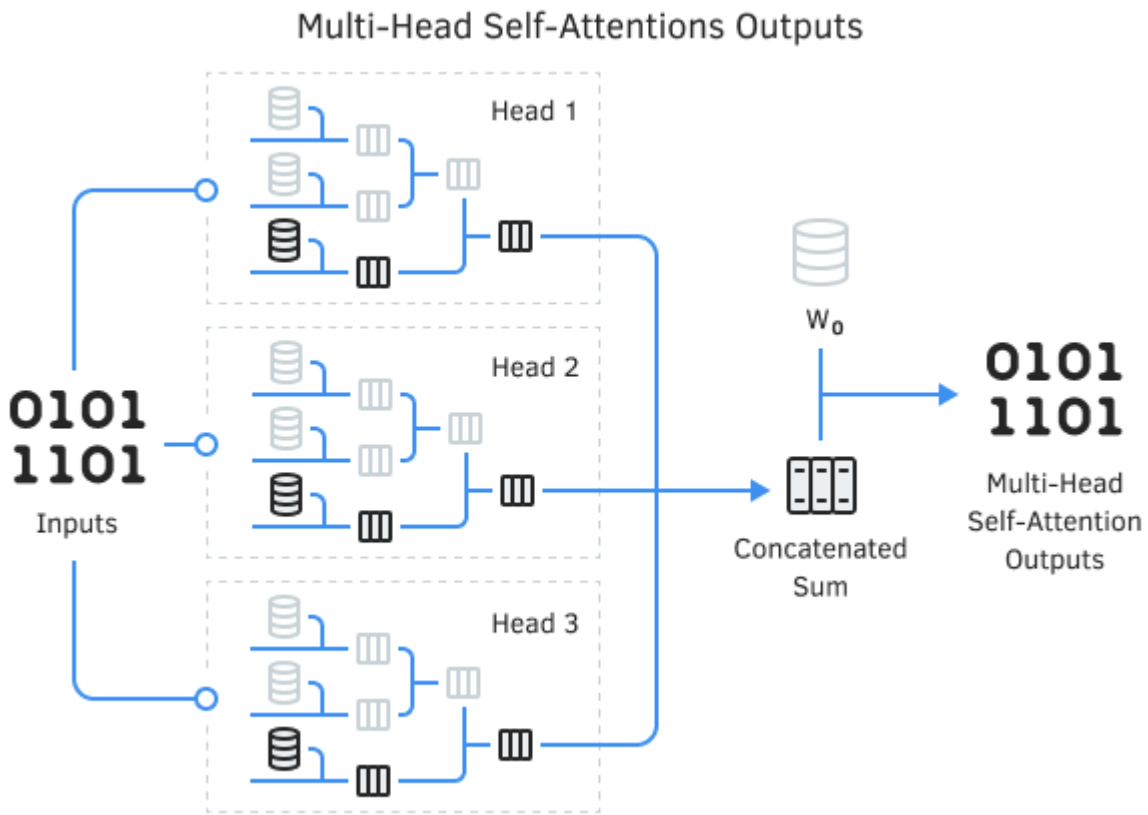
```
      //--- постановка кернела в очередь выполнения
      if(!m_cOpenCL.Execute(def_k_AttentionHiddenGradients, 2, off_set, NDRange))
         return false;
   }
```

На этом завершается наша работа над реализацией алгоритма *Multi-Head Self-Attention* как в части организации многопоточных вычислений, так и в целом. Мы с вами реализовали весь функционал класса *CNeuronMHAttention*. Теперь можем перейти к всестороннему тестированию его работы на обучающей и тестовой выборках.

5.2.4 Построение Multi-Head Self-Attention в Python

Мы уже реализовали алгоритм *Multi-Head Self-Attention* средствами *MQL5* и даже добавили возможность осуществления вычислений в многопоточном режиме с использование технологии *OpenCL*. Теперь давайте посмотрим на вариант реализации подобного алгоритма на языке *Python* с использованием библиотеки *Keras* для *TensorFlow*. Мы познакомились с этой библиотекой при создании предыдущих моделей. Правда, до этого мы использовали только готовые нейронные слои, предлагаемые библиотекой, и с их помощью строили линейные модели.

Модель *Multi-Head Self-Attention* нельзя назвать линейной. Параллельная работа нескольких голов внимания — это уже отказ от линейности модели. Да и в самом алгоритме *Self-Attention* исходные данные одновременно идут в четырех направлениях.



Поэтому для построения модели *Multi-Head Self-Attention* мы рассмотрим еще один функционал, предлагаемый указанной библиотекой, — создание пользовательских нейронных слоев.

Слой — это вызываемый объект, который принимает на вход один или несколько тензоров и выводит один или несколько тензоров. Он включает в себя вычисление и состояние.

Все нейронные слои в библиотеке *Keras* представляют собой классы, унаследованные от базового класса *tf.keras.layers.Layer*. Следовательно, создавая свой новый нейронный слой, мы также будем наследоваться от указанного базового класса.

Базовый класс предусматривает следующие параметры:

- *trainable* — флаг, указывающий на необходимость обучения параметров нейронного слоя;
- *name* — название слоя;
- *dtype* — тип результатов и весовых коэффициентов слоя;
- *dynamic* — флаг, указывающий на невозможность использования слоя для создания графа статических вычислений.

```
tf.keras.layers.Layer(
    trainable=True, name=None, dtype=None, dynamic=False, **kwargs
)
```

Также для каждого слоя архитектурой библиотеки определен минимальный набор методов:

- *__init__* — метод инициализации слоя,
- *call* — метод вычислений (прямого прохода).

В методе инициализации мы определяем пользовательские атрибуты слоя и создаем матрицы весовых коэффициентов, структура которых не зависят от формата и структуры исходных данных. Но при решении практических задач чаще всего мы не знаем структуру исходных данных и, как следствие, не можем создать матрицы весовых коэффициентов без понимания размерности исходных данных. В таких случаях инициализация матриц весовых коэффициентов и других объектов переносится в метод *build(self, input_shape)*. Данный метод вызывается один раз при первом вызове метода *call*.

В методе *call* описываются операции прямого прохода, которые необходимо совершить с исходными данными. Результат операций возвращается в виде одного или нескольких тензоров. Для слоев, используемых в линейных моделях, существует ограничение на результат в виде одного тензора.

Каждый нейронный слой имеет следующие атрибуты (приведен список наиболее употребляемых атрибутов):

- *name* — имя слоя;
- *dtype* — тип весовых коэффициентов;
- *trainable_weights* — список обучаемых переменных;
- *non_trainable_weights* — список не обучаемых переменных;
- *weights* — объединяет списки обучаемых и не обучаемых переменных;
- *trainable* — логический флаг, указывающий на необходимость обучения параметров слоя;
- *activity_regularizer* — дополнительная функция регуляризации для выхода нейронного слоя.

Преимущества такой реализации очевидны: мы не создаем методы обратного прохода. Весь функционал реализуется библиотекой. Нам достаточно лишь правильно описать логику прямого прохода в методе *call*.

Данный подход позволяет создавать довольно сложные архитектурные решения. При этом создаваемый слой может содержать другие вложенные нейронные слои. При этом параметры внутренних нейронных слоев включаются в список параметров внешнего нейронного слоя.

5.2.4.1 Создание класса нового нейронного слоя

Давайте перейдем к практической части и посмотрим на реализацию нашего нейронного слоя многоголового внимания. Для его реализации мы создаем новый класс *MHAttention* наследником от базового класса всех нейронных слоев *tf.keras.layers.Layer*.

```
# Модель Multi-Head Self-Attention
class MHAttention(tf.keras.layers.Layer):
```

Вначале мы переопределим метод инициализации слоя *__init__*. В параметрах метода инициализации мы будем указывать две константы:

- *key_size* — размер вектора описания одного элемента последовательности в теноре ключей Key;
- *heads* — количество голов внимания.

В теле метода мы сохраним параметры в локальные переменные для будущего использования и сразу посчитаем размер конкатенированного выхода голов внимания в переменную *m_iDimension*.

Для вашего удобства я постарался максимально повторить наименование переменных из реализации MQL5.

Затем объявим внутренние объекты нашего нейронного слоя. Но обратите внимание, что в данном случае мы не указываем размер вектора одного элемента последовательности исходных данных. Это стало возможно благодаря использованию многомерных тензоров.

Библиотека TensorFlow работает с многомерными массивами или тензорами, представленными в виде объектов. Такой подход делает понимание модели более удобным и наглядным. Для возможности реализации в OpenCL мы были вынуждены использовать одномерные буферы данных, а для получения доступа к необходимому элементу рассчитывали смещение в одномерном буфере. Сейчас же, при использовании многомерных массивов, для доступа к элементу матрицы нам достаточно указать строку и столбец элемента. Это удобно и наглядно.

Второе преимущество такого подхода — нам нет необходимости указывать размерность исходных данных. Мы можем ее получить из самого тензора. Этим мы и воспользуемся. Мы не будем запрашивать у пользователя размер вектора описания одного элемента последовательности исходных данных — просто получим тензор исходных данных в виде матрицы. Каждая строка такой матрицы представляет собой вектор описания одного элемента последовательности. Мы можем оперировать с размером этого вектора. То есть первое измерение указывает на количество элементов последовательности, а второе означает длину вектора описания одного элемента последовательности.

Но есть и вторая сторона медали. На момент инициализации класса мы еще не получили исходные данные. Соответственно, их размерность нам не известна. И пользователь не указал их в параметрах. Поэтому не все объекты мы можем создать в методе инициализации. Но не беда. Сделаем то, что можем.

В методе инициализации мы объявим объекты, создание которых возможно без понимания размерности исходных данных:

- *m_cQuerys* — нейронный слой формирования конкатенированного тензора запросов *Query;*
- *m_cKeys* — нейронный слой формирования конкатенированного тензора ключей *Key;*
- *m_cValues* — нейронный слой формирования конкатенированного тензора значений *Values;*

- *m_cNormAttention* — слой нормализации данных блока *Multi-Head Self-Attention;*
- *m_cNormOutput* — слой нормализации результатов нейронного слоя.

```
def __init__(self,key_size, heads, **kwargs):
  super(MHAttention, self).__init__(**kwargs)

  self.m_iHeads = heads
  self.m_iKeysSize = key_size
  self.m_iDimension=self.m_iHeads*self.m_iKeysSize;

  self.m_cQuerys = tf.keras.layers.Dense(self.m_iDimension)
  self.m_cKeys = tf.keras.layers.Dense(self.m_iDimension)
  self.m_cValues = tf.keras.layers.Dense(self.m_iDimension)
  self.m_cNormAttention=tf.keras.layers.LayerNormalization(epsilon=1e-6)
  self.m_cNormOutput=tf.keras.layers.LayerNormalization(epsilon=1e-6)
```

После создания метода инициализации перейдем к работе над методом *build*. Именно этот метод позволит нам инициализировать недостающие объекты. Данный метод запускается только один раз перед первым вызовом метода *call* и получает в параметрах размерность исходных данных. Благодаря этому мы можем инициализировать объекты, структуры и/или параметры, которые зависят от размера исходных данных.

В теле метода мы сохраняем последнее измерение тензора исходных данных в качестве размера вектора описания одного элемента последовательности исходных данных в локальную переменную *m_iWindow*. Потом создадим еще три внутренних нейронных слоя:

- *m_cW0* — полносвязный слой понижающей матрицы W_0;
- *m_cFF1* — первый полносвязный слой блока *Feed Forward*;
- *m_cFF2* — второй полносвязный слой блока *Feed Forward*.

```
def build(self, input_shape):
  self.m_iWindow=input_shape[-1]
  self.m_cW0 = tf.keras.layers.Dense(self.m_iWindow)
  self.m_cFF1=tf.keras.layers.Dense(4*self.m_iWindow,
                                    activation=tf.nn.swish)
  self.m_cFF2=tf.keras.layers.Dense(self.m_iWindow)
```

Вот мы и определили все внутренние объекты, необходимые для реализации алгоритма *Multi-Head Self-Attention* внутри нашего нового слоя. Но прежде чем приступить к реализации, давайте еще раз посмотрим каким образом мы можем записать алгоритм многоголового внимания средствами матричной математики. Ведь работая с многомерными тензорами, мы должны оперировать матричными операциями.

Первый шаг — определение тензоров *Query, Key, Value*. Для получения данных запросов нам необходимо умножить тензор исходных данных на соответствующую матрицу весовых коэффициентов. Эту операцию мы возлагаем на три внутренних нейронных слоя.

```
def call(self, data):
  batch_size = tf.shape(data)[0]
  query = self.m_cQuerys(data)
  key = self.m_cKeys(data)
  value = self.m_cValues(data)
```

Второй шаг — определяем матрицу коэффициентов зависимости. По алгоритму *Self-Attention* сначала необходимо умножить тензор запросов на транспонированный тензор ключей.

$$Score = Query * Key^T$$

Для одной головы внимания все просто. Но у нас конкатенированные тензоры, которые в последнем измерении содержат данные всех голов внимания. Умножение их в таком виде даст нам результат сопоставимый с одноголовым вниманием. Как вариант мы можем двухмерный тензор перевести в трехмерный, выделив голову внимания в отдельное измерение.

$$[Units, Concatenate] => [Units, Heads, Vector]$$

Умножение двух последних измерений в таком виде тоже совсем не то, что нам хотелось бы получить. Но вот если еще поменять местами первое и второе измерения, тогда мы можем умножить два последних измерения для получения желаемого результата.

$$[Units, Heads, Vector] => [Heads, Units, Vector]$$

Описанную процедуру вынесем в отдельную функцию *split_heads.*

```
def split_heads(self, x, batch_size):
  x = tf.reshape(x, (batch_size, -1,
                             self.m_iHeads,
                             self.m_iKeysSize))
  return tf.transpose(x, perm=[0, 2, 1, 3])
```

Внутри метода *call* преобразуем тензоры и перемножим их согласно алгоритму *Self-Attention*.

```
query = self.split_heads(query, batch_size)
key = self.split_heads(key, batch_size)
value = self.split_heads(value, batch_size)
score = tf.matmul(query, key, transpose_b=True)
```

Далее полученные коэффициенты зависимости нам необходимо разделить на квадратный корень из размерности вектора ключей и нормализовать функцией *Softmax* по последнему измерению тензора.

```
score = score / tf.math.sqrt(tf.cast(self.m_iKeysSize, tf.float32))
score = tf.nn.softmax(score, axis=-1)
```

Нам остается умножить нормализованные коэффициенты зависимости на тензор ключей *Value*.

```
attention = tf.matmul(score, value)
```

И в результате этой операции мы получим итог работы блока внимания для каждой головы внимания. Для продолжения алгоритма нам нужен конкатенированный тензор всех голов внимания. Следовательно, нам надо осуществить обратную процедуру трансформации тензора. Мы еще раз переставляем местами первое и второе измерения и меняем размерность тензора с трехмерного на двухмерное.

```
attention = tf.transpose(attention, perm=[0, 2, 1, 3])
attention = tf.reshape(attention,(batch_size, -1, self.m_iDimension))
```

После этого с помощью матрицы W_0 мы приводим конкатенированный тензор результатов к размеру тензора исходных данных. Складываем два тензора и нормализуем полученный результат.

```
attention = self.m_cW0(attention)
attention=self.m_cNormAttention(data + attention)
```

На этом заканчивается первый блок алгоритма *Multi-Head Self-Attention* и далее следует два последовательных полносвязных слоя блока *Feed Forward*. Первый нейронный слой будет с функцией активации *Swish*, а второй — без функции активации.

```
output=self.m_cFF1(attention)
output=self.m_cFF2(output)
```

В заключение метода мы складываем тензоры результатов блоков *Multi-Head Self-Attention* и *Feed Forward* и нормализуем слой. Результат операций возвращаем в виде тензора.

```
output=self.m_cNormOutput(attention+output)
return output
```

Мы реализовали минимальный набор методов класса, достаточный для проверки его функциональных возможностей. Но в таком виде, мы не сможем сохранить модель с данным классом. А это, как вы понимаете, совсем нехорошо, ведь мы хотим построить и обучить модель с последующей возможностью промышленной эксплуатации. Поэтому, для нас возможность сохранения модели и последующее ее восстановление является одним из ключевых требований.

Прежде всего, для создания возможности сохранения нового объекта, коим является наш нейронный слой, необходимо добавить его в список пользовательских объектов и дать возможность сериализации объекта. Это позволяет сделать директива *register_keras_serializable*, которую мы добавим перед объявлением класса нашего нейронного слоя.

```
# Модель Multi-Head Self-Attention
@tf.keras.utils.register_keras_serializable(package="Custom", name='MHAttention')
class MHAttention(tf.keras.layers.Layer):
```

Но это не все. Нам еще нужно добавить метод *get_config*, который бы возвращал содержимое переменных для сохранения в файл. Обратите внимание, что среди переменных есть как указываемые пользователем при инициализации объекта класса, так и сохраняемые из размерности исходных данных. Ведь наши весовые коэффициенты настроены именно на эти размерности.

```
def get_config(self):
  config={'key_size': self.m_iKeysSize,
          'heads': self.m_iHeads,
          'dimension': self.m_iDimension,
          'window': self.m_iWindow
          }
  base_config = super(MHAttention, self).get_config()
  return dict(list(base_config.items()) + list(config.items()))
```

За восстановление данных из списка конфигурации отвечает метод *from_config*. Здесь тоже есть свои нюансы. Дело в том, что в обычной логике в словаре конфигурации указываются параметры

из метода инициализации класса. Но мы сохранили и данные, зависящие от размерности исходных данных. А их, как вы помните, нет в параметрах метода инициализации. В чистом виде мы получим ошибку о наличии неизвестных параметров. Поэтому в начале метода мы их удаляем из справочника конфигурации, но при этом сохраняем значения в локальные переменные. И только после этого восстанавливаем слой.

```
@classmethod
def from_config(cls, config):
  dimension=config.pop('dimension')
  window=config.pop('window')
  layer = cls(**config)
  layer._build_from_signature(dimension, window)
  return layer
```

После инициализации нашего нейронного слоя из справочника конфигурации нам необходимо передать в соответствующие переменные предварительно извлеченные нами значения о конфигурации исходных данных. Для выполнения этого функционала мы вызовем метод _build_from_signature, который нам предстоит также переопределить.

```
def _build_from_signature(self, dimension, window):
  self.m_iDimension=dimension
  self.m_iWindow=window
```

На этом мы завершаем работу над классом нашего нейронного слоя и можем перейти к созданию модели для тестирования созданного нейронного слоя *Multi-Head Self-Attention*.

5.2.4.2 Создание скрипта для тестирования технологии Multi-Head Self-Attention

Для тестирования работы нашего нового класса нейронного слоя *Multi-Head Self-Attention* мы создадим скрипт с реализацией модели нейронной сети, в которой и будем использовать новый тип нейронного слоя. Создавать свой скрипт мы будем на базе скрипта *lstm.py*, который использовали при тестировании рекуррентных моделей ранее. Перед началом работы создадим копию указанного скрипта с названием файла *attention.py*. В новой копии скрипта мы удалим ранее созданные модели. Оставим только сверточную модель и лучшую рекуррентную модель — они нам послужат базой для сравнения новых моделей.

```
# Модель с 2-мерным сверточным слоем
model3 = keras.Sequential([keras.Input(shape=inputs),
                           # Переформатируем тензор в 4-мерный.
        # Указываем 3 измерения, т.к. 4-е измерение определяется размером пакета
                           keras.layers.Reshape((-1,4,1)),
                           # Сверточный слой с 8 фильтрами
                           keras.layers.Conv2D(8,(3,1),1,activation=tf.nn.swish,
        kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           # Подвыборочный слой
                           keras.layers.MaxPooling2D((2,1),strides=1),
        # Переформатируем тензор в 2-мерный для полносвязных слоев
                           keras.layers.Flatten(),
                           keras.layers.Dense(40, activation=tf.nn.swish,
        kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(40, activation=tf.nn.swish,
        kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(40, activation=tf.nn.swish,
        kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(targerts, activation=tf.nn.tanh)
                          ])

# Модель LSTM блок без полносвязных слоев
model4 = keras.Sequential([keras.Input(shape=inputs),
        # Переформатируем тензор в 3-мерный.
        # Указываем 2 измерения, т.к. 3-е измерение определяется размером пакета
                           keras.layers.Reshape((-1,4)),
        # 2 последовательных LSTM блока
        # 1-й содержит 40 элементами
                           keras.layers.LSTM(40,
        kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5),
                           return_sequences=False),
        # 2-й выдает результат вместо полносвязного слоя
                           keras.layers.Reshape((-1,2)),
                           keras.layers.LSTM(targerts)
                          ])
```

Для построения первой модели мы создали довольно простую архитектуру из одного слоя внимания, трех полносвязных скрытых слоев и одного полносвязного слоя результатов. Практически такую же архитектуру модели мы использовали выше для построения сверточной модели. Использование похожих моделей позволяет максимально точно оценить влияние новых решений на общий результат модели.

```
heads=8
key_dimension=4

model5 = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
        # Переформатируем тензор в 3-мерный. Указываем 2 измерения,
        # т.к. 3-е измерение определяется размером пакета
        # первое измерение - элементы последовательности
        # второе измерение - вектор описания одного элемента
                           keras.layers.Reshape((-1,4)),
                           MHAttention(key_dimension,heads),
```

Так как наш слой внимания возвращает тензор того же размера, который получает на входе, то перед использованием блока полносвязных слоев нам необходимо вернуть данные в двухмерное пространство.

```
        # Переформатируем тензор в 2-мерный для полносвязных слоев
                           keras.layers.Flatten(),
                           keras.layers.Dense(40, activation=tf.nn.swish,
        kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(40, activation=tf.nn.swish,
        kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(40, activation=tf.nn.swish,
        kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(targerts, activation=tf.nn.tanh)
                          ])
```

Надо сказать, что не смотря на внешнюю схожесть моделей, модель с использованием слоя механизма внимания использует в 5 раз меньше параметров.

Но использование одного слоя внимания — слишком упрощенная модель, она применяется только для сравнительного эксперимента. На практике чаще всего используют несколько последовательных слоев внимания. Я предлагаю на реальных данных оценить влияние использования нескольких слоев внимания в модели. Для проведения такого эксперимента мы добавим в нашу предыдущую модель последовательно еще три слоя внимания с теми же параметрами.

Layer (type)	Output Shape	Param #
reshape_3 (Reshape)	(None, 40, 4)	0
mh_attention (MHAttention)	(None, 40, 4)	776
flatten_1 (Flatten)	(None, 160)	0
dense_7 (Dense)	(None, 40)	6440
dense_8 (Dense)	(None, 40)	1640
dense_9 (Dense)	(None, 40)	1640
dense_10 (Dense)	(None, 2)	82
Total params: 10,578		

Модель с использованием слоя Multi-Heads Self-Attention

```
model6 = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
        # Переформатируем тензор в 3-мерный. Указываем 2 измерения,
        # т.к. 3-е измерение определяется размером пакета
        # первое измерение - элементы последовательности
        # второе измерение - вектор описание одного элемента
                           keras.layers.Reshape((-1,4)),
                           MHAttention(key_dimension,heads),
                           MHAttention(key_dimension,heads),
                           MHAttention(key_dimension,heads),
                           MHAttention(key_dimension,heads),
        # Переформатируем тензор в 2-мерный для полносвязных слоев
                           keras.layers.Flatten(),
                           keras.layers.Dense(40, activation=tf.nn.swish,
        kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(40, activation=tf.nn.swish,
        kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(40, activation=tf.nn.swish,
        kernel_regularizer=keras.regularizers.l1_l2(l1=1e-7, l2=1e-5)),
                           keras.layers.Dense(targerts, activation=tf.nn.tanh)
                         ])
```

Компилировать все нейронные модели мы будем с одинаковыми параметрами. Используем метод оптимизации *Adam*, в качестве ошибки сети используем среднеквадратичное отклонение, добавляем дополнительную метрику *accuracy*.

```
model3.compile(optimizer='Adam',
               loss='mean_squared_error',
               metrics=['accuracy'])
```

С такими же параметрами мы компилировали модели нейронных сетей и ранее.

Напомню, что рекуррентные модели чувствительны к последовательности подаваемого на вход сигнала. Поэтому при обучении рекуррентной нейронной сети, в отличие от остальных моделей, нельзя перемешивать исходные данные. Именно с этой целью при запуске рекуррентной модели мы указываем значение *False* для параметра *shuffle*. При этом у сверточной модели и моделей с использованием слоя внимания указанный параметр установлен *True*. Остальные параметры обучения моделей остаются без изменений, в том числе и ранний выход из процесса обучения при достижении минимума ошибки на обучающей выборке.

```
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=20)

history3 = model3.fit(train_data, train_target,
                      epochs=500, batch_size=1000,
                      callbacks=[callback],
                      verbose=2,
                      validation_split=0.01,
                      shuffle=True)
```

После небольшого обучения моделей визуализируем результаты в графическом представлении. Мы построим два графика. На одном из них выведем динамику изменения ошибки в процессе обучения и валидации.

```
# Отрисовка результатов обучения моделей
plt.figure()
plt.plot(history3.history['loss'], label='Conv2D train')
plt.plot(history3.history['val_loss'], label='Conv2D validation')
plt.plot(history4.history['loss'], label='LSTM only train')
plt.plot(history4.history['val_loss'], label='LSTM only validation')
plt.plot(history5.history['loss'], label='MH Attention train')
plt.plot(history5.history['val_loss'], label='MH Attention validation')
plt.plot(history6.history['loss'], label='MH Attention 4 layers train')
plt.plot(history6.history['val_loss'], label='MH Attention 4 layers validation')
plt.ylabel('$MSE$ $loss$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics')
plt.legend(loc='upper right', ncol=2)
```

На втором графике выведем аналогичные результаты по показателю *Accuracy.*

```
plt.figure()
plt.plot(history3.history['accuracy'], label='Conv2D train')
plt.plot(history3.history['val_accuracy'], label='Conv2D validation')
plt.plot(history4.history['accuracy'], label='LSTM only train')
plt.plot(history4.history['val_accuracy'], label='LSTM only validation')
plt.plot(history5.history['accuracy'], label='MH Attention train')
plt.plot(history5.history['val_accuracy'], label='MH Attention validation')
plt.plot(history6.history['accuracy'], label='MH Attention 4 layers train')
plt.plot(history6.history['val_accuracy'], label='MH Attention 4 layers validation')
plt.ylabel('$Accuracy$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics')
plt.legend(loc='lower right', ncol=2)
```

Затем загрузим тестовую выборку и проверим на ней работу предварительно обученных моделей.

```
# Загрузка тестовой выборки
test_filename = os.path.join(path,'test_data.csv')
test = np.asarray( pd.read_table(test_filename,
                   sep=',',
                   header=None,
                   skipinitialspace=True,
                   encoding='utf-8',
                   float_precision='high',
                   dtype=np.float64,
                   low_memory=False))

# Разделение тестовой выборки на исходные данные и цели
test_data=test[:,0:inputs]
test_target=test[:,inputs:]

# Проверка результатов моделей на тестовой выборке
test_loss3, test_acc3 = model3.evaluate(test_data, test_target, verbose=2)
test_loss4, test_acc4 = model4.evaluate(test_data, test_target, verbose=2)
test_loss5, test_acc5 = model5.evaluate(test_data, test_target, verbose=2)
test_loss6, test_acc6 = model6.evaluate(test_data, test_target, verbose=2)
```

Результаты работы модели на тестовой выборке в числовом виде выведем в журнал и визуализируем на графике.

```
# Вывод результатов тестирования в журнал
print('Conv2D model')
print('Test accuracy:', test_acc3)
print('Test loss:', test_loss3)

print('LSTM only model')
print('Test accuracy:', test_acc4)
print('Test loss:', test_loss4)

print('MH Attention model')
print('Test accuracy:', test_acc5)
print('Test loss:', test_loss5)

print('MH Attention 4l Model')
print('Test accuracy:', test_acc5)
print('Test loss:', test_loss5)

plt.figure()
plt.bar(['Conv2D','LSTM', 'MH Attention','MH Attention\n4 layers'],
        [test_loss3,test_loss4,test_loss5,test_loss6])
plt.ylabel('$MSE$ $loss$')
plt.title('Test results')

plt.figure()
plt.bar(['Conv2D','LSTM', 'MH Attention','MH Attention\n4 layers'],
        [test_acc3,test_acc4,test_acc5,test_acc6])
plt.ylabel('$Accuracy$')
plt.title('Test results')
plt.show()
```

Завершаем нашу работу над механизмом *Multi-Head Self-Attention*. Мы воссоздали данный механизм средствами *MQL5* и на языке *Python.* В данном разделе мы подготовили скрипт на языке *Python*, в котором создается в общей сложности четыре модели нейронных сетей:

- сверточная модель,
- рекуррентная нейронная сеть,
- две модели с использованием технологии *Multi-Head Self-Attention*.

При выполнения скрипта мы проведем небольшое обучение всех четырех моделей на одном наборе исходных данных. Сравним результаты работы обученных моделей на наборе тестовых данных. Это даст нам возможность сравнить работу различных архитектурных решений на реальных данных. Результаты тестов будут даны в следующей главе.

5.2.5 Сравнительное тестирование моделей с использованием механизмов внимания

Мы с вами проделали большую работу в процессе изучения и реализации алгоритма *Multi-Head Self-Attention*. Мы даже успели его реализовать в нескольких платформах. И если ранее мы создавали новые классы только для нашей библиотеки на языке *MQL5,* то сейчас мы познакомились с возможностью создания пользовательских нейронных слоев и на языке *Python* с использованием библиотеки *TensorFlow.* Теперь пришло время посмотреть на плод трудов наших и оценить возможности, предлагаемые нам новой технологией.

Как обычно, тестирование мы начинаем с моделей, созданных стандартными средствами *MQL5*. Мы уже начали эту работу при тестировании работы алгоритма *Self-Attention.* Для проведения нового теста мы возьмем скрипт *attention_test.mq5* из предыдущего теста и создадим его копию с именем *attention_test2.mq5*.

Напомню, в процессе создания нового класса многоголового внимания мы во многом наследовали процессы из алгоритма *Self-Attention.* Где-то полностью наследовали методы, а где-то брали за основу методы *Self-Attention* и создавали новый функционал благодаря небольшим правкам. Так и здесь, скрипт для тестирования не потребует внесения больших изменений — все изменения коснутся лишь блока объявления нового слоя.

Первое наше изменение — это, конечно, тип создаваемого нейронного слоя. В параметре *type* мы укажем константу *defNeuronMHAttention*, соответствующую классу многоголового внимания.

Еще мы должны указать количество используемых голов внимания. Данный показатель мы укажем в параметр *step*. Согласен, что наименование параметра совсем не созвучно. Но было принято решения не создавать дополнительного параметра, а использовать имеющиеся свободные поля.

После этого мы еще раз пробежимся по коду скрипта и внимательно посмотри на ключевые точки контроля выполнения операций.

Вот и все. Таких изменений достаточно для первого тестирования, чтобы оценить чистое влияние архитектуры решения на результат работы модели.

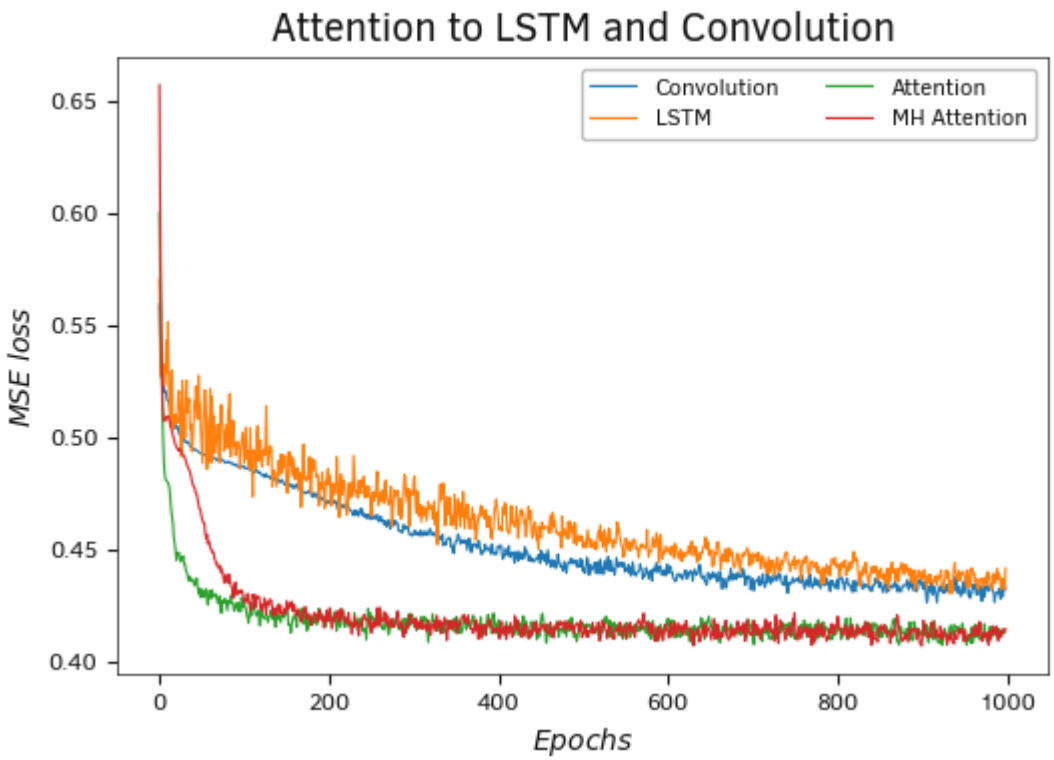
```
//--- Слой внимания
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   descr.type = defNeuronMHAttention;
   descr.count = BarsToLine;
   descr.window = NeuronsToBar;
   descr.window_out = 8;
   descr.step = 8;                // Количество голов внимания
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
```

Непосредственно тестирование мы проводили на все той же обучающей выборке с сохранением всех прочих параметров работы моделей. Их результаты представлены на графике ниже.

Мы уже говорили, что даже использование технологии *Self-Attention* дает превосходство нам над рассмотренными ранее архитектурными решениями сверточных и рекуррентных моделей. Увеличение голов внимания также дает положительный результат.

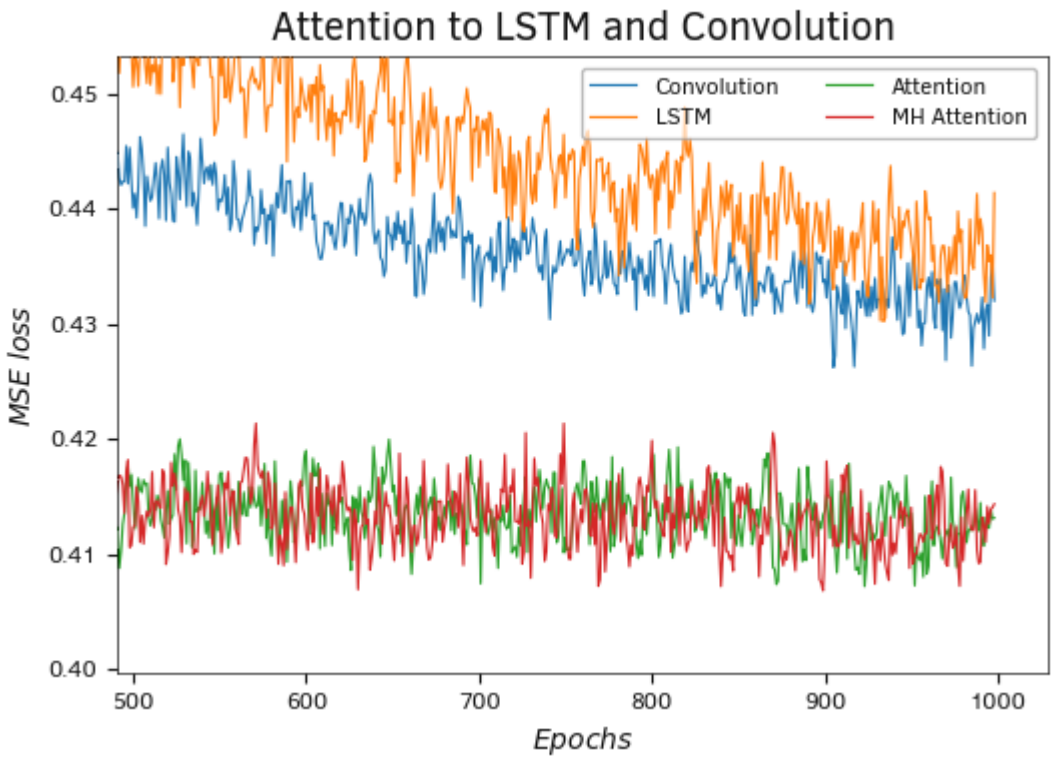
На представленных графиках динамики ошибки нейронной сети на обучающей выборке хорошо видно, что модели с использованием механизма внимания обучаются гораздо быстрее других моделей. Увеличение числа параметров при добавлении голов внимания требует немного

больше времени на обучение. Но это увеличение не критично. В то же время дополнительные головы внимания позволяют снизить ошибку работы модели.



Сравнительное тестирование моделей внимания

А увеличение масштаба графика четко демонстрирует, что ошибка моделей с использованием механизма внимания на протяжении всего обучения находится ниже. При этом использование дополнительных голов внимания еще повышает планку.



Сравнительное тестирование моделей внимания

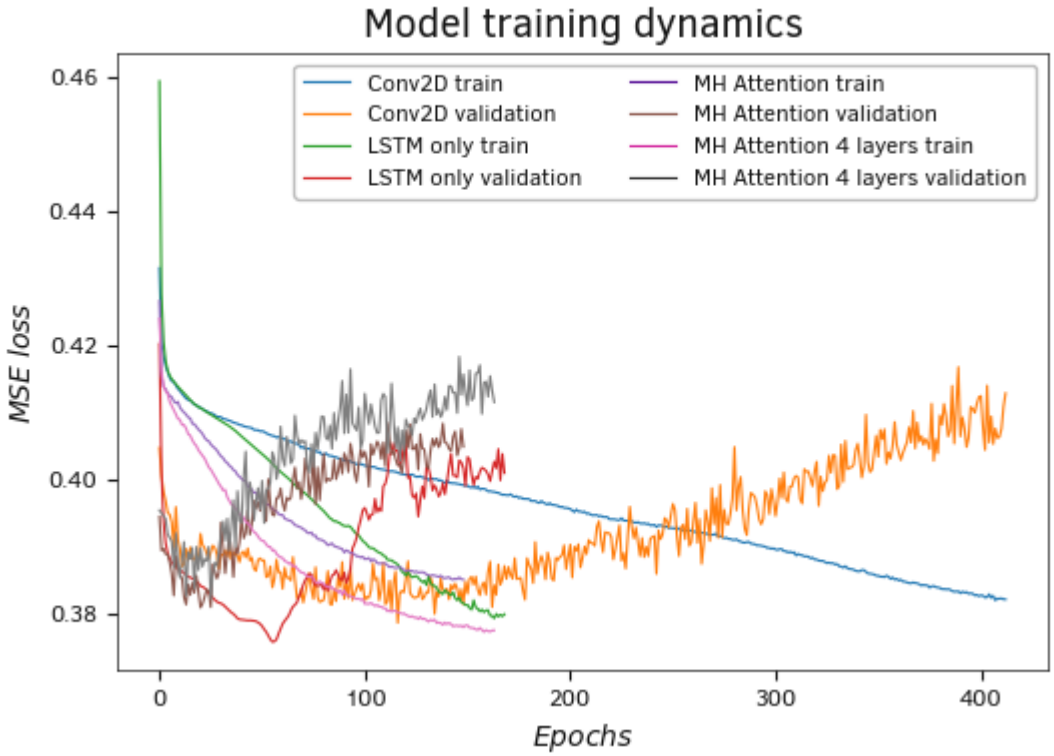
И здесь хочется напомнить, что самым большим количеством обучаемых параметров обладает модель с использованием сверточного слоя. Это дает лишний повод задуматься над рациональностью использования ресурсов и взяться за изучение новых технологий, которые появляются каждый день.

Говоря о рациональности использования ресурсов, хочу также предостеречь от бесконтрольного увеличения используемых голов внимания. Каждая голова внимания — это потребление дополнительных ресурсов. Необходимо находить баланс между объемом потребляемых ресурсов и той пользой, которую они дают на общий результат. Здесь нельзя вывести какую-то константу. Такое решение должно приниматься отдельно для каждого конкретного случая.

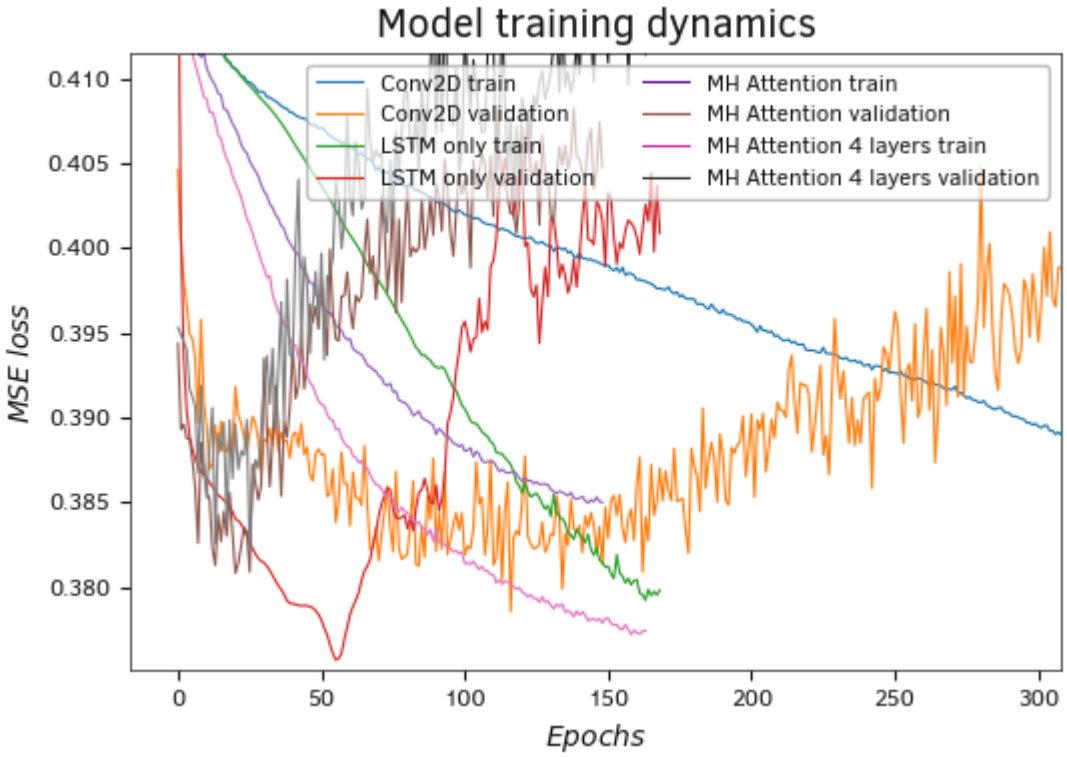
Результаты тестового обучения моделей, написанных на языке *Python*, также подтверждают сделанные выше выводы. Модели с использованием механизмов внимания обучаются быстрее и при этом менее подвержены переобучению модели. Это подтверждается меньшим разрывом между графиками ошибки при обучении и на валидации. Увеличение числа используемых слоев внимания позволяет снизить общую ошибку работы модели при прочих равных условиях.

При увеличении масштаба графика можно заметить, что у моделей с использованием механизмов внимая линии более прямолинейные и менее рваные. Этот свидетельствует о более четком выявлении зависимостей и поступательном движении к минимизации ошибки. Отчасти это можно объяснить нормализацией результатов внутри блока *Self-Attention*, что позволяет иметь на выходе результат с одинаковыми статистическими показателями.

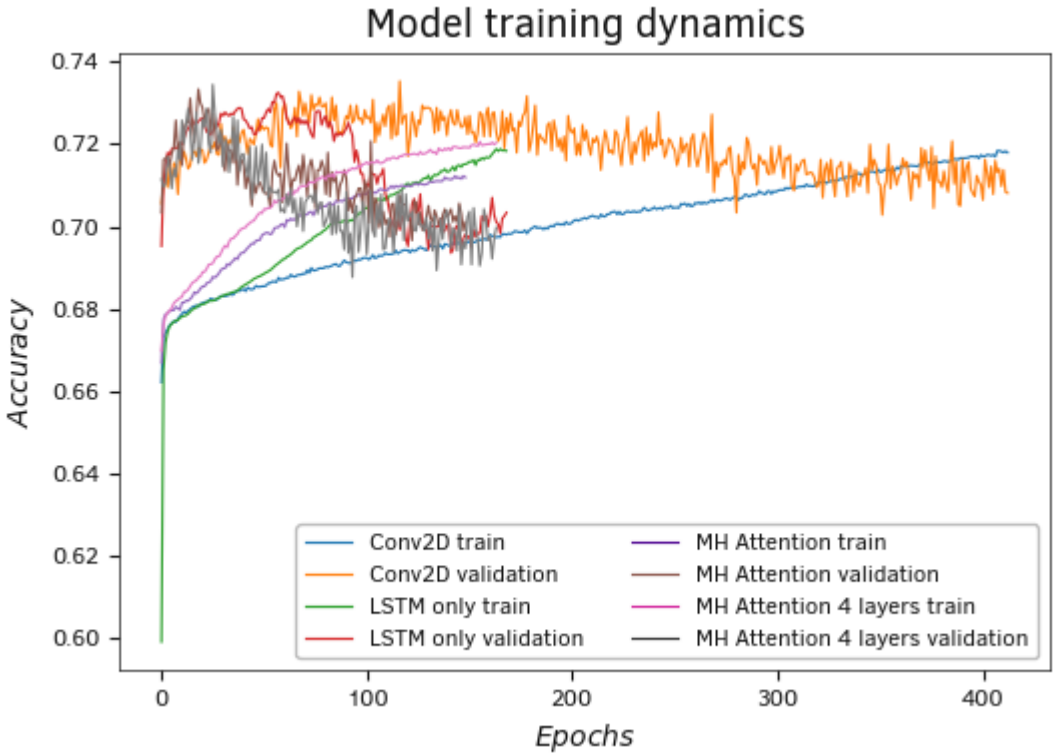
График результатов тестирования по показателю *Accuracy* также подтверждает наши выводы.



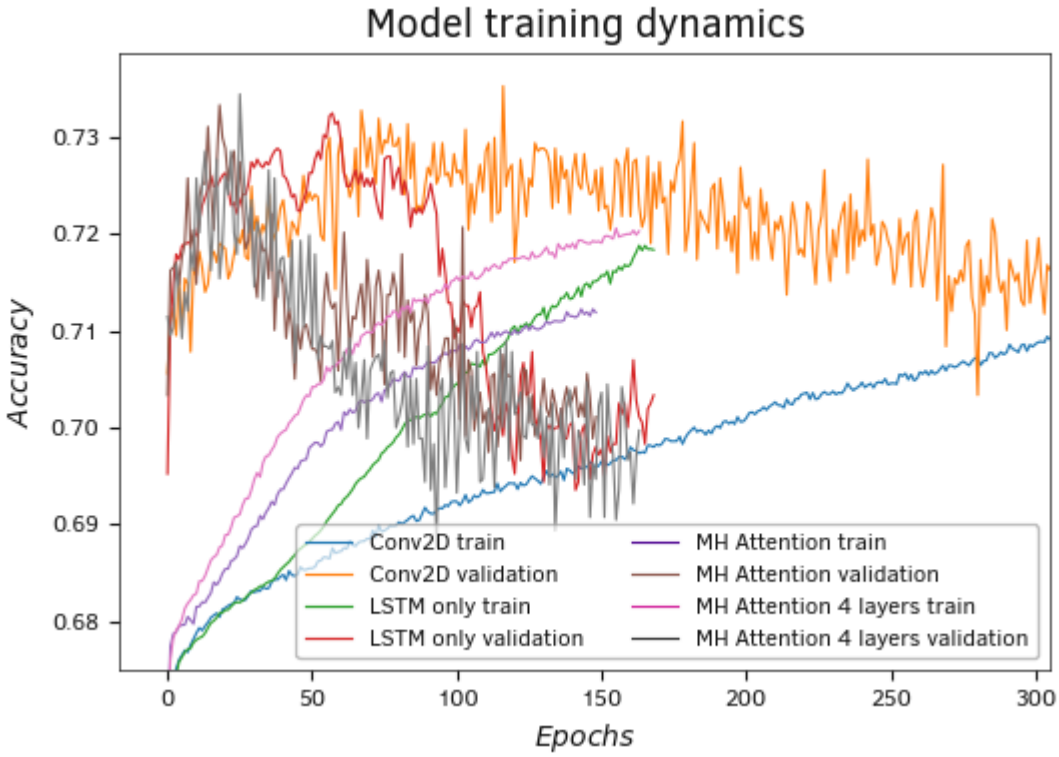
Результаты тестового обучения моделей внимания Python



Результаты тестового обучения моделей внимания Python



Результаты тестового обучения моделей внимания Python



Результаты тестового обучения моделей внимания Python

5.3 Архитектура GPT

В июне 2018 года *OpenAI* представила миру модель нейронной сети *GPT*, которая сразу показала лучшие результаты по целому ряду языковых тестов. В феврале 2019 года появилась *GPT-2*, а в мае 2020 года все узнали о *GPT-3*. Данные модели продемонстрировали возможность генерации нейронной сетью связанного текста. Также проводились эксперименты по генерации музыки и изображений. Основным же недостатком моделей можно назвать требования к вычислительным ресурсам. Для обучения первой *GPT* потребовался месяц на машине с 8 *GPU*. Этот недостаток отчасти компенсируется возможностью использования предварительно обученных моделей для решения новых задач. Но размеры модели требуют ресурсов для ее функционирования.

Концептуально модели *GPT* построены на базе уже рассмотренного нами трансформера. Основная идея заключается в предварительном обучении модели без учителя на большом объеме данных с последующей тонкой настройкой на относительно небольшом количестве размеченных данных.

Причиной двухэтапного обучения является размер модели. Современные модели глубокого машинного обучения, подобные *GPT*, насчитывают большое количество параметров, число которых уже исчисляется сотнями миллионов. Следовательно, обучение подобных нейронных сетей требует огромной обучающей выборки. При использовании обучения с учителем создание размеченной обучающей выборки потребует значительных трудозатрат. В то же время в сети сейчас есть много оцифрованных и не размеченных текстов, которые отлично подходят для обучения модели без учителя. Однако результаты обучения без учителя по статистике уступают обучению с учителем. Поэтому после обучения без учителя осуществляется тонкая настройка модели на сравнительно небольшой выборке размеченных данных.

Обучение без учителя позволяет *GPT* изучить языковую модель, а тонкая настройка на размеченных данных настраивает модель для выполнения конкретных задач. Таким образом, одна предварительно обученная модель может быть тиражирована и настроена на выполнения различных языковых задач. Ограничением выступает язык исходной выборки для обучения без учителя.

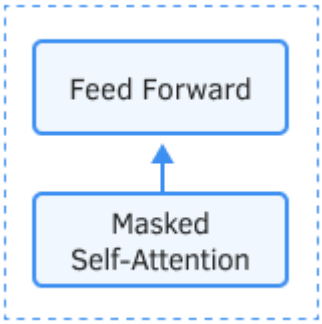
Как показала практика, подобный подход дает неплохие результаты в широком спектре языковых задач. К примеру, модель *GPT-3* способна генерировать связанные тексты на заданную тему. Но тут следует отметить, что указанная модель содержит 175 млрд параметров и предварительно обучена на наборе данных в 570 ГБ.

Несмотря на то, что модели *GPT* были разработаны для обработки естественного языка, они также показали достойные результаты и в задачах генерации музыки и изображений.

Теоретически можно использовать модели *GPT* с любыми последовательностями оцифрованных данных. Вопрос в достаточности данных и ресурсов для предварительного обучения без учителя.

5.3.1 Описание архитектуры и принципов реализации

Рассмотрим отличительные особенности моделей *GPT* от ранее рассмотренного Трансформера. Прежде всего, в моделях *GPT* отказались от использования энкодера, оставив только декодер. При этом отказ от энкодера повлек и отказ от внутреннего слоя *Encoder-Decoder Self-Attention*. На рисунке ниже представлен блок трансформера в *GPT*.



Блок GPT

Так же как и в классическом Трансформере, в моделях *GPT* данные блоки выстраиваются друг над другом. Каждый блок имеет свои матрицы весовых коэффициентов для механизма внимания и полносвязных слоев *Feed Forward*. Количество таких блоков определяет размер модели. Как оказалось, стек блоков может быть довольно большим. В *GPT-1* и самой маленькой из *GPT-2* (*GPT-2 Small*) их 12, в *GPT-2 Extra Large* — 48, а в *GPT-3* их уже 96.

Как и традиционные языковые модели, *GPT* позволяет находить взаимосвязи только с предшествующими элементами последовательности, не позволяя заглядывать в будущее. Но в отличие от трансформера, использует не маскирование элементов, а вносит изменения в вычислительный процесс. В *GPT* обнуляются коэффициенты внимания в матрице *Score* для последующих элементов.

В то же время *GPT* можно отнести к авторегрессионным моделям. Генерируя по одному токену последовательности на каждой итерации, полученный токен добавляется к входной последовательности и подается на вход модели для следующей итерации.

Как и в классическом трансформере, внутри механизма *Self-Attention* для каждого токена генерируются три вектора: запроса *Query*, ключа *Key* и значения *Value*. В авторегрессионной модели, когда на каждой новой итерации входная последовательность изменяется только на 1 токен, нет необходимости пересчитывать векторы для каждого токена. Поэтому в GPT каждый слой осуществляет расчет векторов только для новых элементов последовательности и сохраняет их для каждого элемента последовательности. Каждый блок трансформера сохраняет свои векторы для последующего использования.

Такой подход позволяет модели генерировать тексты слово за словом до получения конечного токена.

И конечно, в моделях *GPT* используется механизм многоголового внимания *Multi-Head Self-Attention*.

5.3.2 Построение модели GPT средствами MQL5

Перед началом работы над моделью *GPT* сразу сказать, не ожидайте получить в конце раздела некоего монстра, способного решать любые задачи. Мы лишь выстраиваем алгоритмы модели. Работа этих алгоритмов будет сопоставима с задействованными вычислительными ресурсами. И мы, конечно, получим и оценим результат работы этих алгоритмов. Но обо всем по порядку.

Приступая к реализации нашего варианта данной модели, давайте вкратце повторим алгоритм:

1. На вход блока Multi-Head Self-Attention подается тензор исходных данных, где каждый элемент последовательности представлен неким токеном (вектором значений).

Одна последовательность для всех голов (потоков). Далее действия в пунктах 2–5 идентичны для каждой головы внимания.

2. Для каждого токена рассчитываются три вектора (*Query, Key, Value*) умножением вектора токена на соответствующую обучаемую матрицу весовых коэффициентов *W*.
3. Перемножая векторы *Query* и *Key*, определяем попарные зависимости между элементами последовательности. На данном этапе вектор *Query* каждого элемента последовательности умножается на векторы *Key* текущего и всех предшествующих элементов последовательности.
4. Матрица полученных коэффициентов зависимости нормализуется с использованием функции *Softmax* в разрезе каждого запроса (*Query*). При этом для последующих элементов последовательности устанавливается нулевой коэффициент внимания.
5. В результате выполнения пунктов 3 и 4 получаем квадратную матрицу Score размерностью равной количеству элементов в последовательности, где сумма всех элементов в разрезе каждого *Query* равна единице.
6. Перемножаем нормализованные коэффициенты внимания на векторы *Value* соответствующих элементов последовательности, складываем полученные векторы и получаем скорректированное на внимание значение для каждого элемента последовательности.
7. Далее определяем взвешенный результат внимания. Для этого конкатенированный тензор результатов всех голов внимания умножается на обучаемую матрицу W_0.
8. Полученный тензор складывается с входной последовательностью и нормализуется.
9. За механизмом *Multi-Heads Self-Attention* следуют два полносвязных слоя блока *Feed Forward*. Первый (скрытый) слой содержит в четыре раза больше нейронов, чем входная последовательность с функцией активации *ReLU* (мы вместо нее использовали функцию *Swish*). Размерность второго слоя равна размерности входной последовательности, а нейроны не используют функцию активации.
10. Результат отработки полносвязных слоев суммируем с тензором, подаваемым на вход блока *Feed Forward*, и нормализуем полученный тензор.

Теперь, когда мы освежили в голове основные этапы процесса, давайте приступим к реализации. Для реализации нового типа нейронного слоя создадим новый класс *CNeuronGPT* наследником от базового класса нейронных слоев нашей модели *CNeuronBase*. Несмотря на использование алгоритма *Self-Attention* в работе модели я не стал наследоваться от уже имеющихся у нас классов нейронных слоев с использованием механизмов внимания. Это связано с некоторыми особенностями реализации модели, с которыми мы познакомимся в процессе работы.

Наверное, одним из основных отличий является возможность выстраивания нескольких однородных слоев в рамках одного класса. Если раньше мы использовали отдельные слои для реализации части функционала модели, то теперь мы говорим о полноценном создании нескольких копий создаваемого слоя со своими весовыми коэффициентами. Для этого в теле метода мы объявляем не отдельные нейронные слои, а целые коллекции слоев. Среди них вы увидите знакомые по работе с предыдущими классами названия переменных, но они уже будут содержать указатели на коллекции нейронных слоев. В то же время мы сохранили функционал, скрываемый за именами объектов. Кроме того, мы добавили две новые переменные:

- *m_iLayers* — количество нейронных слоев в блоке;
- *m_iCurrentPosition* — номер текущего элемента в последовательности.

```
class CNeuronGPT    :  public CNeuronBase
  {
protected:
   CArrayLayers      m_cQuerys;
   CArrayLayers      m_cKeys;
   CArrayLayers      m_cValues;
   CArrayLayers      m_cScores;
   CArrayLayers      m_cAttentionOut;
   CArrayLayers      m_cW0;
   CArrayLayers      m_cFF1;
   CArrayLayers      m_cFF2;
   //---
   int               m_iLayers;
   int               m_iWindow;
   int               m_iUnits;
   int               m_iKeysSize;
   int               m_iHeads;
   CBufferType       m_dStd[];
   int               m_iCurrentPosition;
   int               m_iScoreTemp;

   virtual bool      NormlizeBuffer(CBufferType *buffer, CBufferType *std,
                                                          uint std_shift);
   virtual bool      NormlizeBufferGradient(CBufferType *output,
                     CBufferType *gradient, CBufferType *std, uint std_shift);
public:
                     CNeuronGPT(void);
                    ~CNeuronGPT(void);
   //---
   virtual bool      Init(const CLayerDescription *desc) override;
   virtual bool      SetOpenCL(CMyOpenCL *opencl) override;
   virtual bool      FeedForward(CNeuronBase *prevLayer) override;
   virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer, bool read) override;
   virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                   VECTOR &Beta, VECTOR &Lambda) override;
   //---
   virtual int       GetUnits(void) const { return m_iUnits;   }
   virtual int       GetLayers(void) const { return m_iLayers; }
   //--- методы работы с файлами
   virtual bool      Save(const int file_handle) override;
   virtual bool      Load(const int file_handle) override;
   //--- метод идентификации объекта
   virtual int       Type(void) override  const { return(defNeuronGPT);  }
  };
```

Добавление переменной *m_iCurrentPosition* — это вторая архитектурная особенность данной модели. Мы уже говорили, что *GPT* относится к авторегрессионным моделям. На каждом шаге она возвращает один элемент последовательности и на новой итерации подает его себе на вход. Что-то похожее мы говорили о рекуррентных моделях. Только если в рекуррентных моделях скрытое состояние добавлялось к текущему состоянию окружающей среды, то в случае с *GPT* для воссоздания языковой модели это и есть новое состояния. Конечно, применительно к

финансовым рынкам мы немного отойдем от этой обратной связи и будем подавать на вход реальное новое состояние, но принципы обработки сигнала сбережем.

Логика такова: если на каждой новой итерации обновляется только один элемент последовательности, то зачем нам каждый раз пересчитывать одни и те же значения. Это не эффективно. Давайте пересчитывать только новый элемент последовательности, а для предыдущих элементов последовательности воспользуемся значениями с предыдущих итераций. Именно с этой целью мы вводим переменную для хранения индекса текущего элемента последовательности *m_iCurrentPosition*. С принципом ее использования познакомимся по ходу реализации.

Но обо всем по порядку. Работу над методами класса мы, как обычно, начнем с конструктора класса. В нем мы инициализируем переменные начальными значения. Как и в рассмотренных ранее классах механизмов внимания мы используем статические объекты, которые не требуют создания в конструкторе класса. Деструктор класса остается пустым.

```
CNeuronGPT::CNeuronGPT(void) :   m_iHeads(8),
                                 m_iWindow(0),
                                 m_iKeysSize(0),
                                 m_iUnits(0),
                                 m_iLayers(0),
                                 m_iCurrentPosition(0)
  {
  }
```

Придерживаясь нашей уже не раз используемой схемы работы над классами, следующим мы соберем метод инициализации класса. Данный метод наследуется от родительского класса *CNeuronBase* и переопределяется в каждом новом классе.

В параметрах метод получает указатель на объект описания создаваемого нейронного слоя, и мы сразу осуществляем проверку действительности полученного указателя, а также наличие указанных минимально необходимых параметров для правильной инициализации экземпляра класса.

```
bool CNeuronGPT::Init(const CLayerDescription *desc)
  {
//--- проверяем исходные данные
   if(!desc || desc.type != Type() || desc.count <= 0 || desc.window <= 0 ||
      desc.window_out <= 0 || desc.step <= 0 || desc.layers <= 0)
      return false;
```

После успешного прохождения блока контролей сохраняем полученные параметры в соответствующие переменные нашего класса.

```
//--- сохраняем константы
   m_iWindow   = desc.window;
   m_iUnits    = desc.count;
   m_iKeysSize = desc.window_out;
   m_iHeads    = desc.step;
   m_iLayers   = desc.layers;
   if(!ArrayResize(m_dStd, m_iLayers))
      return false;
   for(int l = 0; l < m_iLayers; l++)
      if(!m_dStd[l].BufferInit(1, 2, 1))
         return false;
```

А затем, как и в случае с ранее созданными классами с использованием механизма внимания, немного скорректируем описание создаваемого нейронного слоя и вызовем метод инициализации родительского класса. Напомню, что в описании создаваемого нейронного слоя мы обнуляем параметр размера окна исходных данных перед вызовом метода родительского класса. Это позволяет нам удалить неиспользуемые объекты буферов родительского класса.

```
//--- вызываем метод инициализации родительского класса
   CLayerDescription *temp = new CLayerDescription();
   if(!temp || !temp.Copy(desc))
      return false;
   temp.window_out = 1;
   temp.window     = 0;
   temp.activation = AF_NONE;
   if(!CNeuronBase::Init(desc))
      return false;
   delete temp;
```

После этого создадим цикл с числом итераций равным числу создаваемых однородных нейронных слоев. И все остальные объекты будем создавать в теле данного цикла.

```
//--- запускаем цикл для создания объектов внутренних слоев
   for(int layer = 0; layer < m_iLayers; layer++)
     {
```

Операции в теле цикла очень похожи на выполняемые операции в методах инициализации классов с использованием механизма *Self-Attention*, но все же отличия есть.

Для начала в теле цикла мы создаем экземпляр объекта для описания создаваемых нейронных слоев *CLayerDescription* и заполним его необходимыми данными. Так как мы определились подавать на вход нейронной сети только обновление состояния, а не всю информацию о паттерне, то я решил отказаться от использования сверточных нейронных слоев и использовать базовый полносвязный нейронный слой. Следовательно, в поле *type* объекта описания нейронного слоя мы укажем константу *defNeuronBase*. Размер окна исходных данных в таком случае будет равен размеру вектора описания одного элемента последовательности. Надо сказать, что в данном случае весь объем подаваемых на вход исходных данных воспринимается как описание одного элемента последовательности.

Далее мы вспоминаем, что в модели используется механизм *Multi-Head Self-Attention*, поэтому нам предстоит создать из одного вектора исходных данных три вектора (*Query, Key, Value*) для каждой головы внимания. Сразу напомню еще один момент: при реализации механизма *Multi-Head Self-Attention* мы использовали конкатенированные векторы. Сейчас мы пошли еще дальше — создаем один тензор не только для всех голов внимания, но и сразу объединяем все три вышеупомянутые

сущности (*Query, Key, Value*). Но так как он будет содержать только один элемент последовательности, то и его размер будет не таким уж и большим. В поле *count* укажем размер равный трем векторам одного элемента последовательности тензора ключей для каждой головы внимания. Функции активации, как и прежде, у создаваемого слоя не будет. Метод оптимизации параметров возьмем тот, что пользователем указал в описании нейронного слоя из параметров метода.

```
temp = new CLayerDescription();
if(!temp)
   return false;
temp.type = defNeuronBase;
temp.window = m_iWindow;
temp.count = (int)(3 * m_iKeysSize * m_iHeads);
temp.activation = AF_NONE;
temp.optimization = desc.optimization;
```

После создания объекта описания нейронного слоя и указания всех необходимых параметров создадим первый внутренний нейронный слой *Querys*. Инициализируем с помощью предварительно созданного объекта описания нейронного слоя. Обязательно не забываем контролировать процесс выполнения операций. После успешного выполнения двух первых операций добавляем слой в соответствующую коллекцию.

```
//--- инициализируем Querys
CNeuronBase *Querys = new CNeuronBase();
if(!Querys)
  {
   delete temp;
   return false;
  }
if(!Querys.Init(temp))
  {
   delete Querys;
   delete temp;
   return false;
  }
if(!m_cQuerys.Add(Querys))
  {
   delete Querys;
   delete temp;
   return false;
  }
```

Несмотря на создание конкатенированного тензора, мы оставили название нейронного слоя *Querys*, что сохраняет преемственность с созданными ранее классами механизмов внимания. Впрочем, внутренние нейронные слои *Keys* и *Values* мы тоже создадим, хотя уже с другими параметрами.

Внутренние нейронные слои *Keys* и *Values* мы будем использовать для накопления исторических данных о полученных текущих состояниях. Так сказать, это память нашего нейронного слоя, и она должна быть достаточной для хранения всего анализируемого паттерна. Но так как мы уже посчитали состояние этих векторов в полносвязном нейронном слое *Querys*, то и матрицы весовых коэффициентов нам в них не нужны. Поэтому перед инициализацией указанных внутренних нейронных слоев внесем изменение в объект описания нейронного слоя: укажем

размер окна исходных данных равным нулю и количество элементов в нейронном слое достаточным для хранения всего тензора описания паттерна.

```
//--- инициализируем Keys
CNeuronBase *Keys = new CNeuronBase();
if(!Keys)
  {
   delete temp;
   return false;
  }
temp.window = 0;
temp.count = (int)(m_iUnits * m_iKeysSize * m_iHeads);
if(!Keys.Init(temp))
  {
   delete Keys;
   delete temp;
   return false;
  }
if(!Keys.GetOutputs().Reshape(m_iUnits, m_iKeysSize * m_iHeads))
   return false;
if(!m_cKeys.Add(Keys))
  {
   delete Keys;
   delete temp;
   return false;
  }
```

В остальном алгоритм создания внутреннего нейронного слоя аналогичен созданию слоя *Querys*:

- создаем новый экземпляр объекта нейронного слоя,
- инициализируем нейронный слой,
- добавляем нейронный слой в соответствующую коллекцию.

```
//--- инициализируем Values
CNeuronBase *Values = new CNeuronBase();
if(!Values)
  {
   delete temp;
   return false;
  }
if(!Values.Init(temp))
  {
   delete Values;
   delete temp;
   return false;
  }
if(!Values.GetOutputs().Reshape(m_iUnits, m_iKeysSize * m_iHeads))
   return false;
if(!m_cValues.Add(Values))
  {
   delete Values;
   delete temp;
   return false;
  }
```

После создания нейронных слоев *Querys, Keys* и *Values* переходим к созданию матрицы коэффициентов зависимости *Score*. Здесь тоже есть особенности реализации. Напомню, что данная матрица в алгоритме реализации *Self-Attention* имеет квадратный размер со стороной квадрата равной количеству элементов последовательности. Каждый элемент матрицы представляет собой коэффициент парной зависимости между элементами последовательности, где строки матрицы соответствуют векторам тензора запросов *Query,* а столбцы матрицы — векторам тензора *Key*.

Теперь подумаем, как мы можем реализовать такую матрицу, если у нас есть только один вектор запроса *Query,* описывающий только последнее состояние. Следовательно, матрица *Score* в данном случае вырождается в вектор. Разумеется, для каждой головы внимания. Конечно, нейронный слой вектора коэффициентов зависимостей *Score* не содержит матрицы весовых коэффициентов. Поэтому изменяем количество элементов в нейронном слое и создаем новый внутренний нейронный слой по указанному выше алгоритму. Воспользуемся возможностью и сделаем матрицу прямоугольной. Строки матрицы будут соответствовать головам внимания.

```
//--- инициализируем Scores
CNeuronBase *Scores = new CNeuronBase();
if(!Scores)
  {
   delete temp;
   return false;
  }
temp.count = (int)(m_iUnits * m_iHeads);
if(!Scores.Init(temp))
  {
   delete Scores;
   delete temp;
   return false;
  }
if(!Scores.GetOutputs().Reshape(m_iHeads, m_iUnits))
   return false;
if(!m_cScores.Add(Scores))
  {
   delete Scores;
   delete temp;
   return false;
  }
```

Следующий объект, который мы будем создавать, — это нейронный слой для конкатенированного выхода результатов работы голов внимания *AttentionOut*. Здесь ситуация аналогична матрице коэффициентов зависимости. Мы уже обсудили причины вырождения матрицы коэффициентов зависимости в вектор, а для получения результата работы головы внимания согласно алгоритму *Self-Attention* нам необходимо умножить матрицу коэффициентов зависимости на тензор значений *Value.*

$$Output = Softmax\left(\frac{Query * Key^T}{\sqrt{Dimension_{Key}}}\right)Value$$

Но в нашем случае с одним вектором запроса *Query* на выходе мы также получаем по одному вектору на каждую голову внимания. Поэтому укажем корректный размер слоя и выполним алгоритм его инициализации.

```
//--- инициализируем AttentionOut
CNeuronBase *AttentionOut = new CNeuronBase();
if(!AttentionOut)
  {
   delete temp;
   return false;
  }
temp.count = (int)(m_iKeysSize * m_iHeads);
if(!AttentionOut.Init(temp))
  {
   delete AttentionOut;
   delete temp;
   return false;
  }
if(!AttentionOut.GetOutputs().Reshape(m_iHeads, m_iKeysSize))
   return false;
if(!m_cAttentionOut.Add(AttentionOut))
  {
   delete AttentionOut;
   delete temp;
   return false;
  }
```

Следуя алгоритму многоголового внимания далее нам предстоит упорядочить результаты работы всех голов внимания в единый вектор и привести его размер в соответствие с размером вектора исходных данных. В алгоритме механизма *Multi-Head Self-Attention* эта операция осуществляется с помощью матрицы W_0. Мы же эту операцию будем выполнять базовым полносвязным нейронным слоем без функции активации.

Снова создаем новый экземпляр объекта нейронного слоя. Не забываем проверить результат проведения операции.

```
//--- инициализируем W0
CNeuronBase *W0 = new CNeuronBase();
if(!W0)
  {
   delete temp;
   return false;
  }
```

В объект описания нейронного слоя вносим необходимые параметры:

- размер окна исходных данных равен размеру ранее созданного слоя конкатенированного результата голов внимания;
- количество элементов на выходе из нейронного слоя равно размеру вектора исходных данных;
- функция активации не используется.

И инициализируем нейронный слой с помощью объекта описания нейронного слоя.

```
temp.window = temp.count;
temp.count = m_iWindow;
temp.activation = AF_NONE;
if(!W0.Init(temp))
  {
   delete W0;
   delete temp;
   return false;
  }
if(!m_cW0.Add(W0))
  {
   delete W0;
   delete temp;
   return false;
  }
```

После успешной инициализации объекта нейронного слоя добавляем его в соответствующую коллекцию.

На этом мы завершаем работу по инициализации объектов механизма *Multi-Head Self-Attention*, и нам остается создать два нейронных слоя блока *Feed Forward.* Первый нейронный слой содержит на выходе в 4 раза больше нейронов, чем получает тензор на входе, и активируется функцией Swish.

```
//--- инициализируем FF1
CNeuronBase *FF1 = new CNeuronBase();
if(!FF1)
  {
   delete temp;
   return false;
  }
temp.window = m_iWindow;
temp.count = temp.window * 4;
temp.activation = AF_SWISH;
temp.activation_params[0] = 1;
temp.activation_params[1] = 0;
if(!FF1.Init(temp))
  {
   delete FF1;
   delete temp;
   return false;
  }
if(!m_cFF1.Add(FF1))
  {
   delete FF1;
   delete temp;
   return false;
  }
```

Второй нейронный слой блока *Feed Forward* уже без функции активации. Он возвращает размер тензора до размера исходных данных. Здесь также используем базовый полносвязный нейронный слой. Внесем необходимые правки в объект описания нейронного слоя и инициализируем нейронный слой.

```
//--- инициализируем FF2
CNeuronBase *FF2 = new CNeuronBase();
if(!FF2)
  {
   delete temp;
   return false;
  }
temp.window = temp.count;
temp.count = m_iWindow;
temp.activation = AF_NONE;
if(!FF2.Init(temp))
  {
   delete FF2;
   delete temp;
   return false;
  }
if(!m_cFF2.Add(FF2))
  {
   delete FF2;
   delete temp;
   return false;
  }
delete temp;
}
```

Проверяем результаты выполнения операций на каждом шаге и добавляем созданный нейронный слой в соответствующую коллекцию.

На этом этапе мы создали все объекты, необходимые для работы одного нейронного слоя. Удаляем объект описания нейронного слоя и переходим на новую итерацию нашего цикла, в которой создадим объекты для работы очередного слоя.

Таким образом, по завершении всех итераций цикла мы получим объекты для работы стольких нейронных слоев, сколько указал пользователь при вызове метода инициализации этого нейронного слоя.

Далее, для исключения копирования данных между буферами внутренних нейронных слоев и текущего слоя, подменим указатели на буферы результатов и градиентов текущего слоя.

```
//--- для исключения копирования буферов осуществим их подмену
   if(m_cFF2.Total() < m_iLayers)
      return false;
   if(!m_cOutputs)
      delete m_cOutputs;
   CNeuronBase *neuron = m_cFF2.At(m_iLayers - 1);
   if(!neuron)
      return false;
   m_cOutputs = neuron.GetOutputs();
   if(!m_cGradients)
      delete m_cGradients;
   m_cGradients = neuron.GetGradients();
```

В заключение метода инициализации вызовем метод распределения между объектами класса указателя на контекст *OpenCL* и выйдем из метода инициализации.

```
   SetOpenCL(m_cOpenCL);
//---
   return true;
  }
```

Для полного закрытия вопроса с инициализацией класса предлагаю рассмотреть еще метод распространения указателя на объект контекста *OpenCL* между внутренними объектами слоя.

Несмотря на изменение типа внутренних объектов с нейронного слоя на коллекцию нейронных слоев, структура и алгоритм метода распространения указателя на контекст *OpenCL* практически не изменились. Такое стало возможно благодаря написанному нами ранее аналогичному методу в классе коллекции нейронных слоев.

В параметрах наш метод *SetOpenCL* получает указатель на объект контекста *OpenCL*. В теле метода мы сначала вызываем аналогичный метод родительского класса, в котором уже реализованы все необходимые контроли, при этом указатель сохраняется в соответствующую переменную класса. После этого поочередно проверяем указатели всех внутренних объектов нейронного слоя и вызываем для них аналогичный метод.

```
bool CNeuronGPT::SetOpenCL(CMyOpenCL *opencl)
  {
   CNeuronBase::SetOpenCL(opencl);
   m_cQuerys.SetOpencl(m_cOpenCL);
   m_cKeys.SetOpencl(m_cOpenCL);
   m_cValues.SetOpencl(m_cOpenCL);
   m_cScores.SetOpencl(m_cOpenCL);
   m_cAttentionOut.SetOpencl(m_cOpenCL);
   m_cW0.SetOpencl(m_cOpenCL);
   m_cFF1.SetOpencl(m_cOpenCL);
   m_cFF2.SetOpencl(m_cOpenCL);
   if(m_cOpenCL)
     {
      uint size = sizeof(TYPE) * m_iUnits * m_iHeads;
      m_iScoreTemp = m_cOpenCL.AddBuffer(size, CL_MEM_READ_WRITE);
      for(int l = 0; l < m_iLayers; l++)
         m_dStd[l].BufferCreate(m_cOpenCL);
     }
   else
     {
      for(int l = 0; l < m_iLayers; l++)
         m_dStd[l].BufferFree();
     }
//---
   return(!!m_cOpenCL);
  }
```

Таким образом мы завершаем работу по инициализации класса и переходим непосредственно к реализации алгоритма работы нейронного слоя. И, как всегда, начнем с реализации метода прямого прохода.

5.3.2.1 Метод прямого прохода GPT

Мы продолжаем нашу работу по реализации алгоритма GPT, предложенного командой *OpenAI*. Мы уже создали базовый скелет класса из объектов для реализации алгоритма. Сейчас же мы приступаем непосредственно к его реализации. Да, в классе будет использован уже знакомый нам алгоритм *Self-Attention*, но с некоторыми особенностями реализации.

Как и во всех ранее рассмотренных классах, весь функционал прямого прохода реализован в методе *CNeuronGPT::FeedForward*. Как вы знаете, данный метод является виртуальным и наследуется от базового класса нейронной сети, при этом он переопределяется в каждом классе для реализации конкретного алгоритма. В параметрах метод получает указатель на объект предыдущего нейронного слоя, в буфере которого содержатся исходные данные для выполнения алгоритма.

Работу метода, как и во всех предыдущих реализациях, мы начинаем с блока контролей. В данном блоке мы проверяем действительность указателей на объекты, задействованные в работе метода. Такая операция позволяет нам исключить многие критические ошибки при обращении к недействительным объектам.

```
bool CNeuronGPT::FeedForward(CNeuronBase *prevLayer)
  {
//--- проверяем актуальность всех объектов
   if(!prevLayer || !prevLayer.GetOutputs())
      return false;
```

Далее мы увеличиваем индекс текущего объекта в буферах ключей *Key* и значений *Value* *m_iCurrentPosition*. Нам данный указатель необходим для организации работы стека в данных буферах. Фактически в алгоритме *Self-Attention* осуществляется взвешенное суммирование различных контекстов в единый вектор. По правилам математики, от перестановки мест слагаемых сумма не меняется. То есть абсолютно неважно, на какой позиции в буфере данных находится элемент, главное его наличие. В этом минус данного алгоритма для работы с таймсериями, но и плюс для нашей реализации. При организации работы стека данных в буферах ключей *Key* и значений *Value* мы не будем осуществлять затратный полный сдвиг данных. Вместо этого мы будем перемещать по стеку указатель и перезаписывать данные в соответствующие элементы буферов данных.

```
//--- увеличиваем указатель на текущий объект в стеке данных
   m_iCurrentPosition++;
   if(m_iCurrentPosition >= m_iUnits)
      m_iCurrentPosition = 0;
```

Следующий нехитрый шаг сделан для организации корректной работы нашей внутренней многослойной архитектуры. Полученный в параметрах указатель на предыдущий нейронный слой нужен нам только для первого внутреннего слоя. А последующие внутренние нейронные слои в качестве исходных данных будут использовать выход предшествующего внутреннего нейронного слоя. Поэтому для внутреннего использования мы введем локальную переменную для хранения указателя на предыдущий нейронный слой. Сейчас мы присвоим ей указатель, полученный в параметрах метода, но по завершении итераций каждого внутреннего нейронного слоя мы будем записывать в нее новый указатель. Таким образом, мы можем организовать работу цикла по перебору всех внутренних нейронных слоев. При этом в теле цикла мы будем работать с одной переменной указателя на объект. А по факту, каждый нейронный слой будет обращаться к буферу своих исходных данных.

```
   CNeuronBase *prevL = prevLayer;
```

Как я уже сказал, основной функционал нашего метода прямого прохода будет реализован в теле цикла, перебирающего внутренние нейронные слои. Поэтому, далее мы организуем такой цикл. Сразу в теле цикла извлекаем из коллекции указатель на объект внутреннего нейронного слоя *Querys*, соответствующего текущему внутреннему нейронному слою. Проверяем действительность извлеченного указателя и запускаем метод прямого прохода данного объекта.

```
//--- запускаем цикл перебора всех внутренних слоев
   for(int layer = 0; layer < m_iLayers; layer++)
     {
      CNeuronBase *Querys = m_cQuerys.At(layer);
      if(!Querys || !Querys.FeedForward(prevL))
         return false;
```

Дальнейший функционал у нас не покрывается методами внутренних объектов. Поэтому, как и в предыдущих реализациях механизма *Self-Attention*, реализуем его в теле метода. И тут надо вспомнить, что во всех реализациях нашей библиотеки мы давали пользователю возможность выбора устройства и технологии осуществления математических операций. В данном классе мы не будем нарушать наших принципов и также реализуем разделения алгоритма в зависимости от используемого вычислительного устройства. Но прежде проведем небольшую подготовительную работу и извлечем из коллекций указатели на объекты анализируемого внутреннего слоя. Не забудем проверить действительность полученных указателей.

```
      CNeuronBase *Querys = m_cQuerys.At(layer);
      if(!Querys || !Querys.FeedForward(prevL))
         return false;
      CNeuronBase *Keys = m_cKeys.At(layer);
      if(!Keys)
         return false;
      CNeuronBase *Values = m_cValues.At(layer);
      if(!Values)
         return false;
      //--- инициализируем Scores
      CNeuronBase *Scores = m_cScores.At(layer);
      if(!Scores)
         return false;
      //--- инициализируем AttentionOut
      CNeuronBase *AttentionOut = m_cAttentionOut.At(layer);
      if(!AttentionOut)
         return false;
```

Далее мы осуществляем разделение алгоритма в зависимости от используемого вычислительного устройства. В данной главе мы рассмотрим организацию процесса стандартными средствами *MQL5*, а к реализации многопоточных вычислений с использованием технологии *OpenCL* мы вернемся в следующих разделах.

```
//--- разветвление алгоритма по вычислительному устройству
if(!m_cOpenCL)
  {
   MATRIX array[];
   if(!Querys.GetOutputs().m_mMatrix.Vsplit(3, array))
      return false;
   if(!Keys.GetOutputs().Row(array[1].Row(0), m_iCurrentPosition))
      return false;
   if(!Values.GetOutputs().Row(array[2].Row(0), m_iCurrentPosition))
      return false;
```

Как вы помните, в ходе выполнения прямого прохода указанного объекта мы формируем сразу все векторы для тензоров *Query, Key* и *Value* для всех голов внимания. На следующем шаге перенесем векторы двух последних тензоров в соответствующие стеки. Для этого разделим буфер результатов слоя Querys на 3 равные части: query, key, value. Скопируем данные в соответствующие буферы данных. При копировании данных воспользуемся переменной *m_iCurrentPosition* для определения смещения в буферах.

Затем проведем небольшую подготовительную работу. Для облегчения доступа к элементам объектов создадим локальные указатели на буферы результатов внутренних нейронных слоев *Query* и *Key.* Также подготовим динамические массивы для выполнения расчетной части.

```
   MATRIX out;
   if(!out.Init(m_iHeads, m_iKeysSize))
      return false;
   MATRIX array_keys[], array_values[];
   MATRIX array_querys[];
   MATRIX keys = Keys.GetOutputs().m_mMatrix;
   MATRIX values = Values.GetOutputs().m_mMatrix;
```

Аналогично построению алгоритма прямого прохода в ранее рассмотренной реализации многоголового внимания мы разделим матрицы данных в соответствии с головами внимания.

```
   if(!array[0].Vsplit(m_iHeads, array_querys))
      return false;
   if(!keys.Reshape(m_iUnits, m_iHeads * m_iKeysSize))
      return false;
   if(!keys.Vsplit(m_iHeads, array_keys))
      return false;
   if(!values.Reshape(m_iUnits, m_iHeads * m_iKeysSize))
      return false;
   if(!values.Vsplit(m_iHeads, array_values))
      return false;
```

После этого организуем вложенный цикл для вычислений. В нем мы будем перебирать используемые головы внимания. Тут же в теле мы извлечем вектор *Query* и матрицу *Keys* анализируемой головы внимания. Перемножим их, полученный вектор разделим на квадратный корень из размерности вектора описания одного элемента последовательности в матрице *Keys*. Нормализуем с помощью функции *Softmax*.

```
      //--- определяем Scores
      for(int head = 0; head < m_iHeads; head++)
        {
         MATRIX score=array_querys[head].MatMul(array_keys[head].Transpose())/
                                                                  sqrt(m_iKeysSize);
         //--- нормализуем Scores
         if(!score.Activation(score,AF_SOFTMAX))
            return false;
         if(!Scores.GetOutputs().Row(score.Row(0), head))
            return false;
```

Таким образом, после нормализации данных сумма всех коэффициентов зависимости будет равна единице. Это дает нам основание ожидать на выходе блока *Self-Attention* вектор с соответствующими характеристиками. Нормализованные данные сохраняем в буфер для последующего использования при обратном проходе.

После расчета и нормализации вектора коэффициентов зависимости у нас есть все необходимые данные для расчета значений выхода блока *Self-Attention.* Умножим нормализованной вектор Score на тензор значений *Value*. Полученный вектор копируем в локальную матрицу результатов.

```
         //--- выход блока внимания
            MATRIX o = score.MatMul(array_values[head]);
            if(!out.Row(o.Row(0), head))
               return false;
        }
```

В результате выполнения всех итераций системы циклов в нашей матрице *out* будут собраны данные конкатенированного выхода блока *Multi-Heads Self-Attention*. Их мы переносим в буфер результатов нейронного слоя *AttentionOut* для последующего использования в нашем алгоритме.

```
      if(!out.Reshape(1, m_iHeads * m_iKeysSize))
         return false;
      AttentionOut.GetOutputs().m_mMatrix = out;
     }
   else // Блок OpenCL
     {
      return false;
     }
```

На этом заканчивается блок разделения операций в зависимости от вычислительного устройства. Далее будем использовать методы наших внутренних объектов.

Согласно алгоритму *Multi-Heads Self-Attention* следующим шагом нам предстоит из конкатенированного выхода всех голов внимания создать единый упорядоченный взвешенный вектор результатов всего блока многоголового внимания. Для этого в алгоритме метода предусмотрена матрица W_0. Мы же возложили этот функционал на внутренний полносвязный нейронный слой *W0*. Извлекаем указатель на объект соответствующего нейронного слоя и вызываем его метод прямого прохода. С целью предотвращения критической ошибки не забываем предварительно проверить действительность указателя на объект.

```
//--- взвешенный выход всех голов внимания
CNeuronBase *W0 = m_cW0.At(layer);
if(!W0 || !W0.FeedForward(AttentionOut))
   return false;
```

Мы приближаемся к завершению реализации алгоритма блока *Multi-Heads Self-Attention*. Согласно алгоритму модели *GPT*, нам необходимо сложить полученный результат с исходными данными и нормализовать результат по формулам.

$$\bar{a} = \frac{1}{h}\sum_{i=1}^{h} a_i$$

$$\hat{a} = \frac{a - \bar{a}}{\sqrt{\frac{1}{h}\sum_{i=1}^{h}(a - \bar{a})^2}}$$

Вначале вызываем метод суммирования двух буферов *CBufferType::SumArray*. Затем нормализуем данные с помощью метода *CNeuronGPT::NormlizeBuffer*. Его алгоритм полностью повторяет одноименный метод класса *CNeuronAttention*.

```
//--- суммируем с исходными данными и нормализуем
if(!W0.GetOutputs().SumArray(prevL.GetOutputs()))
   return false;
if(!NormlizeBuffer(W0.GetOutputs(), GetPointer(m_dStd[layer]), 0))
   return false;
```

После успешной нормализации всех данных нам предстоит провести сигнал через два внутренних нейронных слоя блока *Feed Forward.* В этой операции нет ничего сложного: мы просто последовательно извлекаем указатели на соответствующие объекты нейронных слоев, проверяем действительность указателей и вызываем метод прямого прохода для каждого внутреннего слоя.

```
//--- прямой проход блока Feed Forward
CNeuronBase *FF1 = m_cFF1.At(layer);
if(!FF1 || !FF1.FeedForward(W0))
   return false;
CNeuronBase *FF2 = m_cFF2.At(layer);
if(!FF2 || !FF2.FeedForward(FF1))
   return false;
```

В заключение мы складываем полученный результат блока *Feed Forward* с результатом работы блока *Multi-Heads Self-Attention*. Полученные значения нормализуем.

```
         //--- суммируем с выходом внимания и нормализуем
         CBufferType *prev = FF2.GetOutputs();
         if(!prev.SumArray(W0.GetOutputs()))
            return false;
         if(!NormlizeBuffer(prev, GetPointer(m_dStd[layer]), 1))
            return false;
```

На этом завершается прямой проход для одного внутреннего слоя. Можем перейти к следующей итерации цикла и следующему внутреннему нейронному слою. Но прежде нужно изменить указатель на нейронный слой исходных данных, как мы обсуждали в начале метода. Результаты прямого прохода содержатся в буфере внутреннего нейронного слоя *FF2*. Указатель на него запишем в локальную переменную *prevL*, с которой будет работать на следующей итерации цикла.

```
         prevL = FF2;
        }
  //---
     return true;
    }
```

Таким образом, по завершении всех итераций цикла перебора вложенных нейронных слоев мы получим полный пересчет прямого прохода для нашего блока. Чтобы изменить количество таких нейронных слоев, достаточно изменить один параметр при вызове метода инициализации данного типа класса *CNeuronGPT* модели *GPT*.

На этом завершаем работу над методом прямого прохода и переходим к организации процесса обратного прохода.

5.3.2.2 Методы обратного прохода GPT

В предыдущих разделах мы рассмотрели архитектуру модели *GPT* и даже реализовали методы для инициализации нашего нового класса и прямого прохода по алгоритму модели. Теперь настало время посмотреть на возможный вариант реализации обратного прохода данного алгоритма.

Как вы помните, для реализации обратного прохода в каждом новом классе мы переопределяем три метода:

- *CalcHiddenGradient* — метод расчета градиента ошибки через скрытый слой;
- *CalcDeltaWeights* — метод расчета градиента ошибки до уровня матрицы весов;
- *UpdateWeights* — метод обновления весовых коэффициентов.

Данный класс не будет исключением — переопределим все три метода. Начнем работу с первого по алгоритму обратного прохода и, наверное, всегда самого сложного метода — распределение градиента ошибки через скрытый слой. Именно в этом методе нам предстоит повторить весь алгоритм прямого прохода в обратном порядке.

В параметрах метод получает указатель на объект предыдущего слоя, в который нам предстоит передать градиент ошибки. И сразу в теле метода мы организовываем блок проверок. В нем мы, по уже сложившейся хорошей традиции, проверяем действительность указателей на все используемые в методе объекты. Такой подход позволяет исключить многие критические ошибки в процессе выполнения кода метода.

```
bool CNeuronGPT::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
//--- проверяем актуальность всех объектов
   if(!m_cOutputs || !m_cGradients ||
      m_cOutputs.Total() != m_cGradients.Total())
      return false;
```

Далее, по аналогии с методом прямого прохода, организуем цикл перебора внутренних нейронных слоев. Только в соответствии с принципами обратного прохода, цикл мы тоже организуем с обратным отсчетом итераций. Все дальнейшие итерации будут выполняться в теле цикла и повторяться для всех вложенных слоев нашей модели.

```
//--- запускаем цикл перебора всех внутренних слоев в обратном порядке
   for(int layer = m_iLayers - 1; layer >= 0; layer--)
     {
      CNeuronBase *FF2 = m_cFF2.At(layer);
      if(!FF2)
         return false;
      CBufferType *Gradients = FF2.GetGradients();
      //--- масштабируем градиент на нормализацию
      if(!NormlizeBufferGradient(FF2.GetOutputs(), Gradients,
                                                GetPointer(m_dStd[layer]), 1))
         return false;
```

В теле цикла мы сначала извлекаем указатель на соответствующий нейронный слой выхода блока *Feed Forward FF2* и корректируем его буфер градиентов ошибки на производную функции нормализации. Подробно причины такой операции обсуждались при построении аналогичного метода алгоритма *Self-Attention*.

После этого последовательно вызываем методы распределения градиента ошибки для внутренних слоев блока *Feed Forward*. Методы мы вызываем также в обратном порядке: сначала для второго слоя, а потом для первого.

```
//--- проводим градиент через блок Feed Forward
CNeuronBase *FF1 = m_cFF1.At(layer);
if(!FF2.CalcHiddenGradient(FF1))
   return false;
CNeuronBase *W0 = m_cW0.At(layer);
if(!FF1.CalcHiddenGradient(W0))
   return false;
```

При прямом проходе мы складывали результаты работы блоков *Multi-Heads Self-Attention* и *Feed Forward*. Так же и сейчас нужно провести градиент ошибки по двум направлениям. Мы складываем градиенты ошибок на уровне выхода указанных блоков. Затем суммарный тензор корректируем на производную функции нормализации слоя.

```
CBufferType *attention_grad = W0.GetGradients();
if(!attention_grad.SumArray(Gradients))
   return false;
//--- масштабируем градиент на нормализацию
if(!NormlizeBufferGradient(W0.GetOutputs(), attention_grad,
                                       GetPointer(m_dStd[layer]), 0))
   return false;
```

Далее распределяем градиент ошибки по головам внимания, путем вызова метода распределения градиента ошибки внутреннего нейронного слоя W_0.

```
//--- инициализируем Scores
CNeuronBase *Scores = m_cScores.At(layer);
if(!Scores)
   return false;
//--- распределяем градиент ошибки по головам внимания
CNeuronBase *AttentionOut = m_cAttentionOut.At(layer);
if(!W0.CalcHiddenGradient(AttentionOut))
   return false;
```

До этих пор все было просто и прозрачно. Мы просто в обратном порядке вызывали соответствующие методы наших внутренних нейронных слоев. Но далее начинается тот блок алгоритма, который не покрывается методами внутренних нейронных слоев. Он был реализован внутри метода прямого прохода. Соответственно, и обратный проход градиента ошибки нам тоже предстоит воссоздать полностью. Но для начала выполним подготовительную работу и создадим локальные указатели на объекты *Querys, Keys, Values.* Не забываем сразу проверить действительность полученных указателей на объекты.

```
//--- получаем указатели на объекты Querys, Keys, Values
CNeuronBase *Querys = m_cQuerys.At(layer);
if(!Querys)
   return false;
CNeuronBase *Keys = m_cKeys.At(layer);
if(!Keys)
   return false;
CNeuronBase *Values = m_cValues.At(layer);
if(!Values)
   return false;
```

Далее мы вспоминаем о необходимости создания двух вариантов реализации алгоритма: стандартными средствами *MQL5* и в режиме многопоточных операций с использованием технологии *OpenCL*. Создаем разветвление алгоритма в зависимости от выбранного устройства выполнения математических операций. Как обычно, в данном разделе мы рассмотрим реализацию алгоритма стандартными средствами *MQL5*, а к реализации блока многопоточных операций вернемся в последующих разделах.

Для организации вычислений стандартными средствами *MQL5* мы подготовим динамические массивы. В один мы загрузим из буфера данные градиентов ошибок, другие заполним результатами прямого прохода, а третьи — нулевыми значениями для последующих операций накопления суммы градиентов ошибки.

```
//--- разветвление алгоритма по вычислительному устройству
attention_grad = AttentionOut.GetGradients();
if(!m_cOpenCL)
  {
   MATRIX gradients[];
   if(!attention_grad.m_mMatrix.Vsplit(m_iHeads, gradients))
      return false;
   if(!Querys.GetGradients().m_mMatrix.Reshape(3, m_iHeads * m_iKeysSize))
      return false;
   MATRIX values[];
   if(!Values.GetOutputs().m_mMatrix.Vsplit(m_iHeads, values))
      return false;
   MATRIX keys[];
   if(!Keys.GetOutputs().m_mMatrix.Vsplit(m_iHeads, keys))
      return false;
   MATRIX querys[];
   MATRIX query = Querys.GetOutputs().m_mMatrix;
   if(!query.Reshape(3, m_iHeads * m_iKeysSize) ||
      !query.Resize(1, query.Cols()))
      return false;
   if(!query.Vsplit(m_iHeads, querys))
      return false;
   MATRIX querys_grad = MATRIX::Zeros(m_iHeads, m_iKeysSize);
   MATRIX keys_grad = querys_grad;
   MATRIX values_grad = querys_grad;
```

Первым мы распределим градиент ошибки на тензор *Value.* Сразу скажу, что распределять градиент ошибки мы будем не на весь тензор, а только на текущий элемент. Это вполне объяснимо. Давайте подумаем, с какой целью мы распределяем градиент ошибки. Ведь это не просто работа ради работы. Мы хотим с этого что-то получить. По существу весь процесс

обучения направлен на оптимизацию параметров модели. Мы хотим получить максимально правдоподобную модель. А распределяя градиент ошибки мы хотим получить ориентир для оптимизации параметров модели. Теперь смотрите, распределив градиент ошибки на тензор значений *Value* мы должны его передать по двум направлениям: на предыдущий слой и на матрицу весовых коэффициентов формирования данного тензора текущего слоя.

Передать на предыдущий слой мы можем только градиент ошибки для текущего состояния. Больше буфер предыдущего слоя просто не в состоянии принять, ведь при прямом проходе он нам передал только текущее состояние, для которого он и ожидает градиент ошибки.

Распределить на матрицу весовых коэффициентов мы тоже можем только градиент ошибки текущего состояния. Для распределения ошибки с предыдущих состояний нам нужны и исходные данные предыдущих состояний. А предыдущий слой их не предоставляет, и мы их не сохраняли в буферах нашего слоя.

Таким образом, распределение градиента на элементы тензора значений, разумеется, кроме текущего состояния, является тупиковой задачей и не имеет смысла.

Общий подход таков: при прямом проходе мы считаем только текущее состояние, а также дополнительно берем из памяти уже пересчитанные на предыдущих итерациях. При обратном проходе ситуация аналогичная: считается, что градиент ошибки предыдущих состояний уже учтен при работе методов обратного прохода на предыдущих итерациях. Это значительно сокращает количество операций при совершении каждой итерации прямого и обратного проходов.

Надеюсь, логика ясна. Возвращаемся к нашему методу обратного прохода. Мы остановились на передаче градиента ошибки тензору значений *Value.* Для выполнения этой итерации мы сначала создадим локальный указатель на вектор коэффициентов зависимости, затем организуем цикл.

Наш цикл будет перебирать активные головы внимания. Здесь мы сразу сохраним в локальную матрицу вектор коэффициентов зависимости, который соответствует анализируемой голове внимания. Умножим полученный от предшествующих итераций вектор градиента ошибки на коэффициент зависимости текущего элемента последовательности. Полученные значения сохраняем в матрицу градиентов ошибки на буфере *Values*.

```
for(int head = 0; head < m_iHeads; head++)
  {
   MATRIX score = MATRIX::Zeros(1, m_iUnits);
   if(!score.Row(Scores.GetOutputs().m_mMatrix.Row(head), 0))
      return false;
//--- распределение градиента на Values
   if(!values_grad.Row((gradients[head] *
                            score[0, m_iCurrentPosition]).Row(0), head))
      return false;
```

Далее нужно распределить градиент по второму направлению: через матрицу коэффициентов зависимости на тензоры запросов *Query* и ключей *Key*. Но вначале нам предстоит провести градиент через вектор коэффициентов зависимости. Умножаем матрицы градиентов ошибки на выходе блока внимания и матрицы значений *Values* и получаем градиент на уровне вектора коэффициентов зависимости.

Итак, у нас есть вектор градиентов ошибки для одной головы внимания. Но напомню, что при прямом проходе мы осуществляли нормализацию вектора коэффициентов зависимости функцией *Softmax.* Следовательно, полученные нами градиенты ошибки справедливы для нормализованных

данных. Для дальнейшего распределения градиентов ошибки нам необходимо скорректировать градиенты ошибки на производную указанной функции.

Особенностью функции *Softmax* можно назвать необходимость полного набора значений тензора для вычисления значения каждого элемента. Аналогично и для вычисления производной одного элемента необходим полный набор значений результатов работы функции. В нашем случае результатами работы функции является нормированный вектор коэффициентов зависимости, который мы получили при прямом проходе. Вектор градиентов ошибки мы тоже уже получили. Таким образом, у нас есть все необходимые исходные данные для выполнения операций определения производной функции и корректировки градиента ошибки. Формула производной функции *Softmax* имеет следующий вид:

$$(Softmax(x_i))' = \begin{cases} i = j & y_i(1 - y_i) \\ i \neq j & -y_i y_j \end{cases}$$

Практическую часть операций корректировки градиента ошибки мы осуществляем с помощью матричных операций *MQL5*. А после корректировки градиентов ошибки полученный вектор разделим на квадратный корень из размерности вектора ключа *Key* одного элемента последовательности. Такую же операцию мы совершали при прямом проходе для предотвращения бесконтрольного роста не нормированных коэффициентов зависимости.

```
//--- распределение градиента на Querys и Keys
MATRIX score_grad = gradients[head].MatMul(values[head].Transpose());
//---
MATRIX ident = MATRIX::Identity(m_iUnits, m_iUnits);
MATRIX ones = MATRIX::Ones(m_iUnits, 1);
score = ones.MatMul(score);
score = score.Transpose() * (ident - score);
score_grad = score_grad.MatMul(score.Transpose()) /
                                                  sqrt(m_iKeysSize);
MATRIX temp = score_grad.MatMul(keys[head]);
if(!querys_grad.Row(temp.Row(0), head))
   return false;
temp = querys[head] * score_grad[0, m_iCurrentPosition];
if(!keys_grad.Row(temp.Row(0), head))
   return false;
}
```

В результате проделанных операций получаем скорректированный градиент ошибки для одного элемента вектора коэффициентов зависимости. Но мы не будем его сохранять в очередной буфер данных. Вместо этого мы сразу распределим его на соответствующие элементы тензоров запросов *Query* и ключей *Key*. Для этого нужно умножить данное значение на вектор противоположного тензора. И если для определения градиента ошибки на векторе запросов *Qwery* у нас есть полный набор элементов последовательности в тензоре ключей *Key*, то в тензоре запросов *Qwery* у нас есть только один элемент последовательности. Следовательно, градиент ошибки на тензор ключей *Key* мы будем проводить только для текущего элементу последовательности. Полученные значения градиентов ошибки сохраняем в подготовленные нами ранее матрицы.

Получением градиентов ошибки на уровнях тензоров запросов *Query* и ключей *Key* мы завершаем операции цикла перебора голов внимания.

После завершения полного цикла итераций в наших матрицах *querys_grad*, *keys_grad* и *values_grad* накопились градиенты ошибки к текущему элементу последовательности по всем головам внимания. Нам лишь остается перенести его значения в буфер градиентов ошибки нашего внутреннего слоя *Querys*.

```
      if(!querys_grad.Reshape(1, m_iHeads * m_iKeysSize) ||
         !keys_grad.Reshape(1, m_iHeads * m_iKeysSize) ||
         !values_grad.Reshape(1, m_iHeads * m_iKeysSize))
         return false;
      if(!Querys.GetGradients().Row(querys_grad.Row(0), 0) ||
         !Querys.GetGradients().Row(keys_grad.Row(0), 1) ||
         !Querys.GetGradients().Row(values_grad.Row(0), 2))
         return false;
      if(!Querys.GetGradients().Reshape(1, Querys.GetGradients().Total()))
         return false;
     }
    else // Блок OpenCL
     {
      return false;
     }
```

На этом завершается блок разделения операций алгоритма в зависимости от устройства выполнения операций. Далее мы продолжим выполнения алгоритма с использованием методов наших внутренних нейронных слоев.

Ранее мы уже получили конкатенированный тензор градиентов ошибки, который включает данные всех голов внимания и сразу со всех трех сущностей (*Query, Key, Value*). Теперь с помощью метода распределения градиента через скрытый слой нашего внутреннего нейронного слоя *Querys.CalcHiddenGradient* мы можем перенести градиент ошибки в буфер предыдущего слоя. Но прежде чем выполнить эту операцию, нужно определиться, в буфер какого объекта мы будем записывать градиенты ошибки. Дело в том, что данный класс мы создавали как многослойный блок, и все операции метода выполняются в цикле перебора активного слоя нашего блока. Следовательно, на объект предыдущего нейронного слоя, указатель которого мы получили в параметрах данного метода, мы передаем данные только с первого нейронного слоя нашего блока. У него будет индекс 0 в коллекции вложенных нейронных слоев нашего блока *GPT*. Все же остальные вложенные нейронные слои должны передать градиент ошибки в буфер внутреннего нейронного слоя *FF2* предыдущего вложенного нейронного слоя. Напомню, что *FF2* — это внутренний нейронный слой результатов блока *Feed Forward.*

Поэтому мы создадим локальный указатель на объект предыдущего нейронного слоя и запишем в него указатель на нужный объект в зависимости от индекса активного вложенного нейронного слоя нашего блока *GPT*. И только после получения верного указателя на объект корректного предыдущего слоя мы переносим в его буфер градиент ошибки.

```
        //--- перенос градиента ошибки на предыдущий слой
        CNeuronBase *prevL = (layer == 0 ? prevLayer : m_cFF2.At(layer - 1));
        if(!Querys.CalcHiddenGradient(prevL))
           return false;
        if(!prevL.GetGradients().SumArray(W0.GetGradients()))
           return false;
      }
  //---
    return true;
   }
```

Обратите внимание, что при построении аналогичных методов в классах реализации механизмов внимания в этом месте мы создавали целую процедуру сложения градиенты ошибок с четырех направлений. Сейчас же благодаря использованию конкатенированного буфера градиентов ошибки мы получили суммарный градиент ошибки сразу с трех направлений выполнением метода только одного нейронного слоя. Складывать градиенты все же нам придется, но только один раз. К полученному градиенту ошибки мы должны прибавить градиент ошибки на уровне результатов блока многоголового внимания. Вы ведь помните, что при прямом проходе мы также складывали исходные данные с тензором результатов работы блока многоголового внимания. Следовательно, градиент ошибки должен пройти все те шаги, которые проходит сигнал при прямом проходе, только в обратном порядке.

На этом заканчиваются операции в теле цикла перебора вложенных нейронных слоев нашего блока *GPT*, как и в целом операции нашего метода. Мы закрываем цикл и выходим из метода.

И еще раз хочу сказать: не забывайте контролировать каждый шаг выполнения операций. Это позволяет минимизировать риск критических ошибок и сделать работу программы более контролируемой и надежной.

Мы рассмотрели с вами организацию метода распределения градиента ошибки на предыдущий слой. Но это лишь один их трех методов обратного прохода который мы должны переопределить для данного класса. Поэтому после распределения градиента ошибки на предыдущий нейронный слой нам необходимо довести градиент ошибки до внутренних матриц весовых коэффициентов, которые содержатся в недрах довольно большого количества внутренних объектов нейронных слоев. В соответствии со структурой методов наших классов данный функционал выполняется в методе *CalcDeltaWeights*.

Для распределения градиента ошибки на матрицу весовых коэффициентов любого из ранее рассмотренных нами нейронного слоя необходимы две вещи:

- градиент ошибки на уровне результатов данного нейронного слоя до функции активации;
- исходные данные, которые предоставляются предыдущим нейронным слоем.

Для организации данного процесса у нас уже есть все необходимые данные. В предыдущем методе мы распределили градиент ошибки до каждого нейронного слоя. Указатель на предыдущий нейронный слой мы получаем в параметрах метода *CNeuronGPT::CalcDeltaWeights*.

В теле метода мы как обычно организовываем контрольный блок по проверке указателей всех используемых внутренних объектов. Надо сказать, что блок контролей должен быть минимальным и достаточным. Нужно исключить избыточные и явно повторяющиеся контроли, так как они не несут ценности для работы программы, но при этом тормозят ее выполнение. При этом каждая операция, в том числе и контрольная, требует ресурсов и времени. Давайте подумаем, матрицы весовых коэффициентов каких объектов мы должны обновить. Это:

- нейронный слой запросов *Query*, который возвращает конкатенированный тензор трех сущностей (Query, Key, Value);
- нейронный слой матрицы W_0;
- два нейронных слоя блока *Feed Forward*.

Все указанные объекты объявлены статичными. В связи с этим, нет необходимости проверять их указатели, так как их наличие контролируется системой. Это позволяет нам исключить блок контролей из данного метода.

Далее все банально и просто. Организуем цикл по перебору всех вложенных нейронных слоев нашего блока *GPT*. В теле блока поочередно извлекаем все объекты указанных выше коллекций. Сначала проверяем указатель на объект, а затем вызываем его метод распределения градиента ошибки до уровня матрицы весовых коэффициентов.

```
bool CNeuronGPT::CalcDeltaWeights(CNeuronBase *prevLayer, bool read)
  {
//--- в цикле вызываем аналогичный метод для каждого внутреннего объекта
   for(int layer = 0; layer < m_iLayers; layer++)
     {
      if(!m_cFF2.At(layer))
         return false;
      CNeuronBase *temp = m_cFF2.At(layer);
      if(!temp.CalcDeltaWeights(m_cFF1.At(layer), false))
         return false;
      temp = m_cFF1.At(layer);
      if(!temp.CalcDeltaWeights(m_cW0.At(layer), false))
         return false;
      temp = m_cW0.At(layer);
      if(!temp.CalcDeltaWeights(m_cAttentionOut.At(layer), false))
         return false;
      temp = m_cQuerys.At(layer);
      if(!temp)
         return false;
      CNeuronBase *prevL = (layer == 0 ? prevLayer : m_cFF2.At(layer - 1));
      if(!temp.CalcDeltaWeights(prevL, (read && layer == m_iLayers - 1)))
         return false;
     }
//---
   return true;
  }
```

Следует сказать немного слов о выстраивании порядка вызова методов внутренних объектов. С точки зрения выполнения математических операций, порядок вызова методов не влияет на конечный результат. Но используемый в теле цикла порядок вызова методов неслучаен. Обратите внимание, что в теле цикла мы явно проверяем указатели только на два объекта, которые не выполняют роль исходных данных других внутренних слоев. Дело в том, что в вызываемых методах нейронных слоев также существует блок контролей, который проверяет поступающие данные, в том числе и получаемые указатели на объекты. Поэтому, чтобы исключить повторные проверки указателей на объекты, мы сначала передаем указатель на объект в качестве исходных данных другого объекта, проверяем результат выполнения операций вызванного метода, который среди прочего подтверждает и действительность переданного указателя, а затем уже смело обращаемся к объекту, так как его указатель был проверен во время работы метода предыдущего

объекта. Таким образом, мы организовываем полную проверку всех указателей на объекты без явного контроля в теле метода и исключаем избыточные повторные проверки указателей, которые бы тормозили работу программы.

Следующим рассмотрим метод обновления параметров модели. Для выполнения этой функции не требуются данные внешних объектов. В параметрах метода нет ни одного указателя на объект, есть только значение параметров для выполнения заданного алгоритма оптимизации параметров.

В теле метода мы также организуем цикл по перебору вложенных нейронных слоев нашего блока *GPT.* В теле цикла будем извлекать по одному объекту из каждой коллекции, проверять действительность указателя и вызывать метод обновления матрицы весов каждого объекта.

```
bool CNeuronGPT::UpdateWeights(int batch_size, TYPE learningRate,
                               VECTOR &Beta, VECTOR &Lambda)
  {
//--- в цикле вызываем аналогичный метод для каждого внутреннего объекта
   for(int layer = 0; layer < m_iLayers; layer++)
     {
      CNeuronBase *temp = m_cFF2.At(layer);
      if(!temp || !temp.UpdateWeights(batch_size, learningRate, Beta, Lambda))
         return false;
      temp = m_cFF1.At(layer);
      if(!temp || !temp.UpdateWeights(batch_size, learningRate, Beta, Lambda))
         return false;
      temp = m_cW0.At(layer);
      if(!temp || !temp.UpdateWeights(batch_size, learningRate, Beta, Lambda))
         return false;
      temp = m_cQuerys.At(layer);
      if(!temp || !temp.UpdateWeights(batch_size, learningRate, Beta, Lambda))
         return false;
     }
//---
   return true;
  }
```

Так как вызываемые методы не обращаются к внешним объектам, то и наш прием с оптимизацией контролей здесь не будет работать ввиду отсутствия явно повторяющихся контролей. Поэтому нам нужно явно проверять каждый указатель на объект перед обращением к его методу.

Мы рассмотрели реализацию трех методов обратного прохода и на этом заканчиваем работу над реализацией алгоритма модели *GPT* в нашем классе *CNeuronGPT*. Для полной реализации функционала стандартными средствами *MQL5* нам остается переопределить методы работы с файлами — мы не раз уже говорили о важности этих методов для эксплуатации моделей нейронных сетей.

5.3.2.3 Методы работы с файлами

Мы продолжаем работу над нашим классом реализации модели *GPT*. Фактически мы уже реализовали функционал данной модели в методах нашего класса *CNeuronGPT*. В предыдущих разделах мы разобрали методы инициализации объекта, а также выстроили процессы прямого и обратного проходов. Указанного функционала вполне достаточно для создания тестовой модели и даже можно провести ряд тестов работоспособности модели.

Но мы уже не раз говорили о важности методов работы с файлами для промышленной эксплуатации любой модели нейронной сети. Основную значимость данному процессу придают стоимость процесса обучения модели, ведь этот процесс требует затрат как времени, так и ресурсов. Часто подобные затраты бывают довольно высокие. Поэтому есть огромное желание обучить модель один раз и в последующем использовать ее с максимальной нагрузкой в минимально возможные сроки.

Конечно, изменчивость финансовых рынков не оставляет нам надежды на бесконечно долгое использование однажды обученной модели. Но и здесь, даже с изменчивостью окружающей среды, дообучение модели в новых условиях потребует меньших затрат ресурсов и времени, чем обучение модели полностью со случайных весовых коэффициентов.

Поэтому давайте продолжим нашу работу и реализуем методы работы с файлами. Как всегда, начнем с метода сохранения данных в файл *CNeuronGPT::Save*.

Приступая к работе над методом сохранения данных, мы как обычно бросаем критический взгляд на структуру нашего класса и оцениваем необходимость сохранения данных каждого объекта.

```
class CNeuronGPT    :  public CNeuronBase
  {
protected:
   CArrayLayers      m_cQuerys;
   CArrayLayers      m_cKeys;
   CArrayLayers      m_cValues;
   CArrayLayers      m_cScores;
   CArrayLayers      m_cAttentionOut;
   CArrayLayers      m_cW0;
   CArrayLayers      m_cFF1;
   CArrayLayers      m_cFF2;
   //---
   int               m_iLayers;
   int               m_iWindow;
   int               m_iUnits;
   int               m_iKeysSize;
   int               m_iHeads;
   CBufferType       m_dStd[];
   int               m_iCurrentPosition;
   int               m_iScoreTemp;

   virtual bool      NormlizeBuffer(CBufferType *buffer, CBufferType *std,
                                                          uint std_shift);
   virtual bool      NormlizeBufferGradient(CBufferType *output,
                     CBufferType *gradient, CBufferType *std, uint std_shift);
public:
                     CNeuronGPT(void);
                    ~CNeuronGPT(void);
   //---
   virtual bool      Init(const CLayerDescription *desc) override;
   virtual bool      SetOpenCL(CMyOpenCL *opencl) override;
   virtual bool      FeedForward(CNeuronBase *prevLayer) override;
   virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer, bool read) override;
   virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                                   VECTOR &Beta, VECTOR &Lambda) override;
   //---
   virtual int       GetUnits(void) const { return m_iUnits;   }
   virtual int       GetLayers(void) const { return m_iLayers; }
   //--- методы работы с файлами
   virtual bool      Save(const int file_handle) override;
   virtual bool      Load(const int file_handle) override;
   //--- метод идентификации объекта
   virtual int       Type(void) override  const { return(defNeuronGPT);  }
  };
```

И тут мы понимаем, что кроме констант наш класс содержит только коллекции объектов. Трудозатраты на повторное создание объектов коллекций с полным описанием их структуры будут гораздо выше возможной экономии ресурсов дискового пространства. Поэтому мы организуем сохранение всех коллекций в файл данных для восстановления модели.

В параметрах данный метод получает хендл файла для сохранения данных. Чтобы избежать дублирующих контролей и сократить общий объем кода программы, мы не будем проверять полученный хендл. Вместо этого мы просто вызовем аналогичный метод родительского класса, в который и передадим полученный хендл. Преимущества такого подхода очевидны. Одной командой мы проверяем полученный хендл и сохраняем данные объектов, унаследованных от родительского класса. Одной проверкой результата выполнения метода родительского класса мы контролируем весь указанный процесс.

```
bool CNeuronGPT::Save(const int file_handle)
  {
//--- вызов метода родительского класса
   if(!CNeuronBase::Save(file_handle))
      return false;
```

После успешного выполнения метода родительского класса мы сохраняем в файл константы нашего метода:

- *m_iLayers* — количество вложенных нейронных слоев блока *GPT;*
- *m_iWindow* — размер окна исходных данных (размер вектора описания одного элемента последовательности исходных данных);
- *m_iKeysSize* — размер вектора описания одного элемента тензора ключей *Keys*;
- *m_iHeads* — количество используемых голов внимания;
- *m_iUnits* — количество элементов в последовательности;
- *m_iCurrentPosition* — позиция текущего анализируемого элемента.

```
//--- сохраняем константы
   if(FileWriteInteger(file_handle, m_iLayers) <= 0)
      return false;
   if(FileWriteInteger(file_handle, m_iWindow) <= 0)
      return false;
   if(FileWriteInteger(file_handle, m_iKeysSize) <= 0)
      return false;
   if(FileWriteInteger(file_handle, m_iHeads) <= 0)
      return false;
   if(FileWriteInteger(file_handle, m_iUnits) <= 0)
      return false;
   if(FileWriteInteger(file_handle, m_iCurrentPosition) <= 0)
      return false;
```

Сохранение позиции текущего анализируемого элемента необходимо для правильной работы стеков ключей *Key* и значений *Value*. Впрочем в условиях промышленной эксплуатации я бы рекомендовал перед использованием модели последовательно подать ей на вход данные в объеме, достаточном для полного заполнения стеков. Такой подход позволит контролировать процесс загрузки данных в модель и исключить риск возможных пропусков, которые потенциально могут отрицательно сказаться на точности работы модели на начальном этапе после загрузки данных. Конечно, работа модели выровняется после полного заполнения стека. Но риск получения убытков до этого момента повышается.

Далее мы поочередно проверяем указатели на объекты всех наших коллекций и вызываем их методы сохранения данных.

```
//--- вызываем аналогичный метод для всех коллекций внутренних слоев
   if(!m_cQuerys.Save(file_handle))
      return false;
   if(!m_cKeys.Save(file_handle))
      return false;
   if(!m_cValues.Save(file_handle))
      return false;
   if(!m_cScores.Save(file_handle))
      return false;
   if(!m_cAttentionOut.Save(file_handle))
      return false;
   if(!m_cW0.Save(file_handle))
      return false;
   if(!m_cFF1.Save(file_handle))
      return false;
   if(!m_cFF2.Save(file_handle))
      return false;
//---
   return true;
  }
```

После этого выходим из метода сохранения данных.

Что ж, метод сохранения объекта нашего класса мы создали. Теперь можем перейти к работе над методом восстановления объекта из данных, записанных в файл. Напомню, что основным требованием к методам восстановления работоспособности объектов из сохраненных данных является считывание данных в строгом соответствии с последовательностью их записи.

Как и метод записи в файл, наш метод загрузки данных *CNeuronGPT::Load* в параметрах получает хендл файла, содержащего данные для считывания. Точно также, как и при записи данных, сначала мы вызываем аналогичный метод родительского класса. Во-первых, мы выполняем чтение данных в строгом соответствии с последовательностью записи. Во-вторых, используем идею, озвученную при изучении метода записи данных, т.е. используем контроли, реализованные в методе родительского класса, и исключаем их дублирование. Конечно, перед тем как двигаться дальше, мы проверяем результат выполнения операций родительского метода.

```
bool CNeuronGPT::Load(const int file_handle)
  {
//--- вызов метода родительского класса
   if(!CNeuronBase::Load(file_handle))
      return false;
```

После успешного выполнения метода родительского класса мы считываем константы параметров работы нашего блока. Считывание их значений осуществляется в порядке их записи. После считывания значения констант не забываем изменить размер динамического массива для записи стандартных отклонений, используемых при нормализации результатов работы нашего блока. Размер массива должен быть достаточным для хранения данных со всех вложенных нейронных слоев. В противном случае мы рискуем получить критическую ошибку выхода за размеры массива в процессе выполнения программы.

```
//--- считываем константы из файла
   m_iLayers = FileReadInteger(file_handle);
   m_iWindow = FileReadInteger(file_handle);
   m_iKeysSize = FileReadInteger(file_handle);
   m_iHeads = FileReadInteger(file_handle);
   m_iUnits = FileReadInteger(file_handle);
   m_iCurrentPosition = FileReadInteger(file_handle);
   if(ArrayResize(m_dStd, m_iLayers) <= 0)
      return false;
   for(int i = 0; i < m_iLayers; i++)
      if(!m_dStd[i].BufferInit(1, 2, 1))
         return false;;
```

Дальше нам остается лишь загрузить данные наших коллекций объектов. Но прежде чем вызвать метод загрузки данных коллекции объектов, нужно убедиться в актуальности указателя на объект коллекции и при необходимости создать новый экземпляр объекта коллекции. Только потом можно вызывать метод загрузки данных. Конечно, не забываем, что порядок загрузки объектов идет в строгом соответствии с порядком их записи. Также контролируем процесс загрузки данных на каждой итерации.

```
//--- вызываем аналогичный метод для всех коллекций внутренних слоев
   if(!m_cQuerys.Load(file_handle))
      return false;
   if(!m_cKeys.Load(file_handle))
      return false;
   if(!m_cValues.Load(file_handle))
      return false;
   if(!m_cScores.Load(file_handle))
      return false;
   if(!m_cAttentionOut.Load(file_handle))
      return false;
   if(!m_cW0.Load(file_handle))
      return false;
   if(!m_cFF1.Load(file_handle))
      return false;
   if(!m_cFF2.Load(file_handle))
      return false;
```

После загрузки всех объектов мы создадим еще один цикл и переформатируем буферы результатов всех созданных объектов. В данном случае мы не осуществляем проверку действительности указателей на объект — в предыдущих итерациях все эти объекты загружали данные с файла, а значит, были созданы и проверены.

```
//--- переформатируем матрицы результатов
   for(int i = 0; i < m_iLayers; i++)
     {
      CNeuronBase* temp = m_cKeys.At(i);
      if(!temp.GetOutputs().Reshape(m_iUnits, m_iKeysSize * m_iHeads))
         return false;
      temp = m_cValues.At(i);
      if(!temp.GetOutputs().Reshape(m_iUnits, m_iKeysSize * m_iHeads))
         return false;
      temp = m_cScores.At(i);
      if(!temp.GetOutputs().Reshape(m_iHeads, m_iUnits))
         return false;
      temp = m_cAttentionOut.At(i);
      if(!temp.GetOutputs().Reshape(m_iHeads, m_iKeysSize))
         return false;
     }
```

В завершение метода мы осуществим подмену буферов и завершаем его работу.

```
//--- осуществляем подмену буферов данных для исключения излишнего копирования
   CNeuronBase *last = m_cFF2.At(m_cFF2.Total() - 1);
   if(!m_cOutputs)
      delete m_cOutputs;
   m_cOutputs = last.GetOutputs();
   if(!m_cGradients)
      delete m_cGradients;
   m_cGradients = last.GetGradients();
//---
   return true;
  }
```

Теперь, когда созданы методы работы с файлами, мы можем двигаться дальше. Дальше у нас по плану создание возможности выполнения параллельных математических операций с использованием технологии *OpenCL*.

5.3.3 Организация параллельных вычислений в модели GPT

Мы продолжаем работу над нашим классом модели *GPT CNeuronGPT*. В прошлых разделах мы уже воссоздали алгоритм модели стандартными средствами *MQL5*. Теперь пришло время дополнить модель возможностью выполнения математических операций в многопоточном режиме с использованием вычислительных мощностей *GPU*. Именно такую возможность предоставляет нам технология *OpenCL*.

Для организации этого процесса нам предстоит выполнить две подзадачи:

- создание программы *OpenCL*,
- организация вызова программы *OpenCL* из основной программы.

Начнем работу с создания выполняемой программы на стороне *OpenCL*. В данной программе нужно реализовать ту часть алгоритма, которая не покрывается использованием методов внутренних объектов. У нас два таких блока: один — в части прямого прохода, а второй, зеркальный первому, — в методе распространения градиента ошибки при выполнении обратного прохода.

Для выполнения алгоритма прямого прохода мы создадим кернел *GPTFeedForward*. Отчасти алгоритм кернела напомнит аналогичный кернел для классов с использованием механизмов внимания. Это и не удивительно — все они используют механизм *Self-Attention.* Но в каждой реализации есть свои нюансы. Если в прошлый раз вместо создания нового кернела для организации многоголового внимания мы смогли довольно быстро модифицировать уже имеющийся кернел алгоритма *Self-Attention*, то сейчас создание нового кернела мне показалось менее затратным по сравнению с попыткой создания универсального кернела для всех задач.

В отличие от реализации механизма *Multi-Heads Self-Attention*, когда мы переводили кернел в двухмерное пространство задач, в данной реализации мы возвращаемся в одно мерное пространство. Это связано с отсутствием возможности деления задачи на параллельные потоки в разрезе элементов последовательности тензора запросов *Query*, потому что в реализации модели *GPT* предусмотрена обработка только одного запроса за итерацию. В этом случае у нас остается деление по потокам только в разрезе голов внимания.

В параметрах кернела прямого прохода *GPTFeedForward* мы, как и прежде, будем передавать указатели на пять буферов данных. Но при этом увеличивается количество переменных: если раньше мы получали размер последовательности из размерности пространства задач, то теперь нам приходится его явно указывать в параметрах кернела. Здесь же добавляется параметр для указания текущего элемента в последовательностях ключей и значений.

```
__kernel void GPTFeedForward(__global TYPE *querys,
                             __global TYPE *keys,
                             __global TYPE *scores,
                             __global TYPE *values,
                             __global TYPE *outputs,
                             int key_size,
                             int units,
                             int current)
  {
```

Как уже было сказано выше, создаваемый кернел будет работать в одномерном пространстве задача в разрезе голов внимания. Поэтому первое, что мы делаем в теле кернела, это определяем активную голову внимания по идентификатору выполняемого потока и общее количество голов внимания по общему количеству запущенных потоков.

```
  const int h = get_global_id(0);
  const int heads = get_global_size(0);
  int shift_query = key_size * h;
  int shift_scores = units * h;
```

Сразу же определяем смещение в тензорах запросов *Query* и матрицы коэффициентов зависимости *Score*.

Далее, согласно выстраиваемому алгоритму *Self-Attention*, определим коэффициенты зависимости между элементами последовательности. Для этого умножим вектор запроса *Query* на тензор ключей *Key*. Для реализации этих операций создадим систему из двух вложенных циклов. Внешний цикл будет перебирать элементы последовательности тензора ключей *Key* и, соответственно, элементы вектора коэффициентов зависимости *Score.* В теле цикла мы определим смещение в тензоре ключей *Key* и подготовим локальную переменную для подсчета промежуточных значений.

После этого организуем вложенный цикл с числом итераций равным размеру вектора описания одного элемента последовательности. В теле данного цикла мы и осуществим операцию

умножения пары векторов. Полученное значение делим на квадратный корень из размерности вектора, берем экспоненту. Результат операции записываем в соответствующий элемент вектора коэффициентов зависимости и прибавляем к накопительной сумме всех элементов векторы коэффициентов зависимости для последующей нормализации данных.

Следует учесть момент конкатенированного буфера результатов слоя Query. Дело в том, что значения вектора ключей и запросов текущего элемента последовательности еще не перенесены в соответствующие буферы. Поэтому мы проверяем элемент, к которому обращаемся в тензоре ключей. Перед обращением к текущему элементу мы сначала копируем данные в буфер. Конечно, для текущих операций мы могли бы взять данные из буфера *querys*. Но эти данные нам потребуются и при последующих итерациях. Следовательно, перенос их в буфер неизбежен.

```
TYPE summ = 0;
for(int s = 0; s < units; s++)
  {
   TYPE score = 0;
   int shift_key = key_size * (s * heads + h);
   for(int k = 0; k < key_size; k ++)
     {
      if(s == current)
         keys[shift_key + k] = querys[shift_query + k + heads * key_size];
      score += querys[shift_query + k] * keys[shift_key + k];
     }
   score = exp(score / sqrt((TYPE)key_size));
   summ += score;
   scores[shift_scores + s] = score;
  }
```

В результате выполнения полного цикла итераций созданной выше системы из двух циклов мы получим вектор коэффициентов зависимости. Согласно алгоритму *Self-Attention*, перед дальнейшим использованием полученных коэффициентов их предстоит нормализовать функцией *Softmax.* При получении экспоненты из произведений векторов мы уже выполнили часть алгоритма указанной функции. Для полного завершения операции нормализации нам остается разделить сохраненные в векторе значения на их общую сумму, которую мы предусмотрительно собрали в локальной переменной *summ*. Поэтому организуем еще один цикл с числом итераций равным размеру вектора коэффициентов зависимости. В теле данного цикла мы разделим все значения вектора на значение локальной переменной *summ.*

```
for(int s = 0; s < units; s++)
   scores[shift_scores + s] /= summ;
```

Таким образом, после завершения итераций цикла в векторе *Score* мы получи нормализованные значения коэффициентов зависимости с общей суммой всех элементов раной единице. Фактически полученные коэффициенты дают нам представление о доле влияния каждого элемента последовательности из тензора значений *Value* на итоговое значение анализируемого элемента последовательности в тензоре результатов текущей головы внимания.

А значит, для получения итоговых значений нам предстоит умножить вектор нормализованных коэффициентов зависимости *Score* на тензор значений *Value.* Для выполнения этой операции нам понадобится еще одна система из двух вложенных циклов. Но перед ее запуском мы определим смещение в тензоре результатов до начала вектора анализируемого элемента последовательности.

Внешний цикл с числом итераций равным размеру вектора описания одного элемента последовательности укажет нам на порядковый номер собираемого элемента в векторе результатов. Вложенный цикл с числом итераций равным числу элементов в последовательности поможет сопоставить векторы тензора значений *Value* и коэффициенты зависимости из вектора *Score.* В теле вложенного цикла мы будем выполнять непосредственно операцию умножения вектора из тензора значений *Value* на соответствующий коэффициент зависимости элементов. Полученные значения произведений мы суммируем в локальную переменную, а после завершения итераций вложенного цикла сохраним полученное значение в буфере результатов работы блока *Self-Attention.*

```
   shift_query = key_size * h;
   for(int i = 0; i < key_size; i++)
     {
      TYPE query = 0;
      for(int v = 0; v < units; v++)
        {
         if(v == current)
            values[key_size * (v * heads + h) + i] =
                                       querys[(2 * heads + h) * key_size + i];
         query += values[key_size * (v * heads + h) + i] *
                                                    scores[shift_scores + v];
        }
      outputs[shift_query + i] = query;
     }
  }
```

В результате выполнения полного цикла итераций системы циклов мы получим вектор описания одного элемента последовательности в тензоре результатов одной головы внимания. Поставленная перед данным кернелом задача выполнена, и мы выходим из него.

На этом мы заканчиваем работу с кернелом прямого прохода и двигаемся дальше. Теперь нам предстоит организовать процесс обратного прохода. Реализацию данной задачи разобьем на два кернела. В кернеле *GPTCalcScoreGradient* мы доведем градиент ошибки до вектора коэффициентов зависимости. Во кернеле *GPTCalcHiddenGradient* продолжим распределение градиента ошибки и доведем его до уровня тензоров запросов *Query* и ключей *Key*.

Но обо всем по порядку. Кернел *GPTCalcScoreGradient* в параметрах получает указатели на шесть буферов данных и три параметра:

- *scores* — буфер вектора коэффициентов зависимости;
- *scores_grad* — буфер вектора градиентов ошибки на уровне коэффициентов зависимости;
- *values* — буфер тензора значений *Value*;
- *values_grad* — буфер тензора градиентов ошибки на уровне *Value*;
- *outputs_grad* — буфер тензора градиентов ошибки на уровне результатов блока *Self-Attention*;
- *scores_temp* — буфер для записи промежуточных значений;
- *window* — размер вектора описания одного элемента последовательности в тензоре значений *Value*;
- *units* — количество элементов в последовательности;
- *current* — порядковый номер текущего элемента в стеке значений *Value*.

```
__kernel void GPTCalcScoreGradient(__global TYPE *scores,
                                   __global TYPE *scores_grad,
                                   __global TYPE *values,
                                   __global TYPE *values_grad,
                                   __global TYPE *outputs_grad,
                                   __global TYPE *scores_temp,
                                   int window,
                                   int units,
                                   int current)
  {
```

Как и при прямом проходе, кернел мы будем запускать в одномерном пространстве по числу используемых голов внимания. В теле кернела мы сразу определяем активную голову внимания по идентификатору потока и общее количество голов внимания по общему количеству запущенных потоков.

```
   const int h = get_global_id(0);
   const int heads = get_global_size(0);
   int shift_value = window * (2 * heads + h);
   int shift_score = units * h;
```

И сразу определяем смещение в тензорах градиентов ошибки на уровне значений *Value* и в векторе коэффициентов зависимости. Обратите внимание, что смещение в тензоре градиентов ошибки значений Value и в самом тензоре значений *Value* в данном случае будет различным.

Напомню, что в данной реализации модели *GPT* мы использовали один внутренний нейронный слой для генерации конкатенированного тензора, содержащего сразу значение тензоров *Query, Key* и *Value* для всех голов внимания. Соответственно, и градиент ошибки мы собираем в аналогичный конкатенированный тензор градиентов ошибки указанного нейронного слоя. Но данный тензор содержит только текущий элемент последовательности. В то же время в тензоре стека *Value* содержится полная информация обо всей последовательности, но только тензора значений *Value*.

После проведения подготовительной работы мы распределим градиент ошибки на тензор значений *Value*. Как уже было сказано выше, мы распределяем градиент ошибки только для текущего элемента последовательности. Для этого мы организуем цикл с числом итераций равным размеру вектора описания одного элемента последовательности в тензоре значений *Value*. В теле цикла мы умножим вектор градиентов ошибки на уровне результатов блока *Self-Attention* на соответствующий коэффициент зависимости из вектора *Score*. Полученные значения сохраняем в буфер конкатенированного тензора градиентов ошибки.

```
//--- Распределение градиента на Values
   for(int i = 0; i < window; i ++)
      values_grad[shift_value + i] = scores[units * h + current] *
                                                    outputs_grad[window * h + i];
```

После расчета градиента ошибки на тензоре значений *Value* определим значение градиента ошибки на уровне вектора коэффициентов зависимости. Для выполнения этой операции нам понадобится система из двух циклов: внешнего с числом итераций равным числу элементов последовательности и вложенным с числом итераций равным размеру вектора описания одного элемента последовательности в тензоре значений *Value*. В теле вложенного цикла мы будем осуществлять произведение 2-х векторов (*Value* на градиент ошибки). А полученное значение сохраняем в буфер временных данных.

```
//--- Распределение градиента на Score
   for(int k = 0; k < units; k++)
     {
      TYPE grad = 0;
      for(int i = 0; i < window; i++)
         grad += outputs_grad[shift_value + i] *
                                        values[window * (k * heads + h) + i];
      scores_temp[shift_score + k] = grad;
     }
```

В результате выполнения полного цикла итераций системы циклов во временном буфере мы получим полностью заполненный вектор градиентов ошибки для вектора коэффициентов зависимости. Но для распределения градиента ошибки дальше, нам предварительно надо его скорректировать на производную функции *Softmax.*

Организуем еще одну систему из двух вложенных циклов. Оба цикла будут содержать количество итераций равное числу элементов в последовательности. В теле вложенного цикла мы посчитаем производную функции по формуле.

$$(Softmax(x_i))' = \begin{cases} i = j & y_i(1 - y_i) \\ i \neq j & -y_i y_j \end{cases}$$

```
//--- Корректируем на производную Softmax
   for(int k = 0; k < units; k++)
     {
      TYPE grad = 0;
      TYPE score = scores[shift_score + k];
      for(int i = 0; i < units; i++)
         grad += scores[shift_score + i] * ((int)(i == k) - score) *
                                                scores_temp[shift_score + i];
      scores_grad[shift_score + k] = grad;
     }
  }
```

Полученные значения сохраним в буфер градиентов ошибки на уровне вектора коэффициентов зависимости. На данном этапе мы завершаем работу первого кернела и переходим к работе над вторым.

Во втором кернеле организации процесса обратного прохода *GPTCalcHiddenGradient* нам предстоит распределить градиент ошибки дальше и довести его до уровня тензоров запросов *Query* и ключей *Key*.

В параметрах кернел *GPTCalcHiddenGradient* получает указатели на 4 буфера данных и 3 параметра.

```
__kernel void GPTCalcHiddenGradient(__global TYPE *querys,
                                    __global TYPE *querys_grad,
                                    __global TYPE *keys,
                                    __global TYPE *scores_grad,
                                    int key_size,
                                    int units,
                                    int current)
  {
```

Обратите внимание, что мы говорили о распределении градиента на тензоры *Query* и *Key*. А в параметрах кернела есть указатель только на буфер градиентов ошибки *Query*. Такая ситуация стала возможной благодаря использованию конкатенированного буфера, в который мы уже сохранили градиент ошибки на уровне тензора значений *Value*. И теперь в тот же буфер мы добавим градиенты ошибок на уровне тензоров *Query* и *Key*.

В теле кернела мы определяем порядковый номер анализируемой головы внимания по идентификатору потока и количество используемых голов внимания по общему количеству запущенного пула задач.

```
   const int h = get_global_id(0);
   const int heads = get_global_size(0);
   int shift_query = key_size * h;
   int shift_key = key_size * (heads + h);
   int shift_score = units * h;
```

Здесь же мы определяем смещения в буферах данных до начала анализируемых векторов.

Далее мы организуем систему из двух вложенных циклов, в теле которой определим градиенты ошибок на уровне искомых тензоров. Для этого умножим градиент ошибки на уровне вектора коэффициентов зависимости на противоположный тензор.

```
//--- Распределение градиента на Querys и Keys
   const TYPE k = 1 / sqrt((TYPE)key_size);
//---
   for(int i = 0; i < key_size; i++)
     {
      TYPE grad_q = 0;
      TYPE grad_k = 0;
      for(int s = 0; s < units; s++)
        {
         grad_q += keys[key_size * (s * heads + h) + i] *
                                                 scores_grad[shift_score + s];
         if(s == current)
            grad_k += querys[key_size * h + i] *
                                              scores_grad[units * h + current];
        }
      querys_grad[shift_query + i] = grad_q * k;
      querys_grad[shift_key + i] = grad_k * k;
     }
  }
```

Стоит отметить, что мы определяем градиент ошибки только для текущего элемента последовательности и записываем полученные значения в соответствующие элементы буфера градиентов ошибок.

В результате выполнения всех итераций нашей системы циклов мы получаем полностью заполненный конкатенированный тензор градиентов ошибок всех трех сущностей (*Query, Key, Value*). Завершаем работу по построению программы *OpenCL* и переходим к построению функционала на стороне основной программы.

Чтобы обслуживать построенные кернелы на стороне основной программы было удобнее, создадим именованные константы вызова кернелов и доступа к его элементам. Для этого открываем наш файл констант *defines.mqh* и создаем в нем константы обращения к кернелам.

```
#define def_k_GPTFeedForward          34
#define def_k_GPTScoreGradients       35
#define def_k_GPTHiddenGradients      36
```

Добавляем константы обращения к параметрам кернелов.

```
//--- прямой проход GPT
#define def_gptff_querys              0
#define def_gptff_keys                1
#define def_gptff_scores              2
#define def_gptff_values              3
#define def_gptff_outputs             4
#define def_gptff_key_size            5
#define def_gptff_units               6
#define def_gptff_current             7

//--- определение градиента на матрице коэффициентов зависимости GPT
#define def_gptscr_scores             0
#define def_gptscr_scores_grad        1
#define def_gptscr_values             2
#define def_gptscr_values_grad        3
#define def_gptscr_outputs_grad       4
#define def_gptscr_scores_temp        5
#define def_gptscr_window             6
#define def_gptscr_units              7
#define def_gptscr_current            8

//--- распределение градиента через GPT
#define def_gpthgr_querys             0
#define def_gpthgr_querys_grad        1
#define def_gpthgr_keys               2
#define def_gpthgr_scores_grad        3
#define def_gpthgr_key_size           4
#define def_gpthgr_units              5
#define def_gpthgr_current            6
```

После этого переходим к диспетчерскому классу обслуживания модели нейронной сети *CNet* и в методе инициализации контекста *OpenCL InitOpenCL* изменяем общее количество кернелов в нашей программе. Затем инициализируем создание новых кернелов в контексте *OpenCL*.

```
bool CNet::InitOpenCL(void)
  {
   ......
   if(!m_cOpenCL.SetKernelsCount(37))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
   ......
   if(!m_cOpenCL.KernelCreate(def_k_GPTFeedForward, "GPTFeedForward"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
   if(!m_cOpenCL.KernelCreate(def_k_GPTScoreGradients, "GPTCalcScoreGradient"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
   if(!m_cOpenCL.KernelCreate(def_k_GPTHiddenGradients, "GPTCalcHiddenGradient"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
//---
   return true;
  }
```

На этом мы завершаем подготовительную работу и переходим непосредственно к методам нашего класса *CNeuronGPT*. В них нам предстоит выполнить три этапа работы для вызова каждого кернела:

- подготовка исходных данных и передача их в память контекста *OpenCL*;
- постановка кернела в очередь выполнения;
- загрузка результатов выполнения программы в память основной программы.

Первым мы модифицируем метод прямого прохода *CNeuronGPT::FeedForward*. В блоке организации многопоточных вычислений с использованием технологии *OpenCL* мы сначала проверяем наличие уже созданного буфера в памяти контекста *OpenCL*.

```
bool CNeuronGPT::FeedForward(CNeuronBase *prevLayer)
  {
   ......
   for(int layer = 0; layer < m_iLayers; layer++)
     {
   ......
      //--- разветвление алгоритма по вычислительному устройству
      if(!m_cOpenCL)
        {
         // Блок программы стандартными средствами MQL5
   ......
        }
      else // блок OpenCL
        {
         //--- проверка буферов данных
         if(Querys.GetOutputs().GetIndex() < 0)
            return false;
         if(Keys.GetOutputs().GetIndex() < 0)
            return false;
         if(Values.GetOutputs().GetIndex() < 0)
            return false;
         if(Scores.GetOutputs().GetIndex() < 0)
            return false;
         if(AttentionOut.GetOutputs().GetIndex() < 0)
            return false;
```

Когда все буферы созданы, а те из них, что необходимы для работы кернела, переданы в память контекста *OpenCL*, мы предаем в параметры кернела указатели на используемые буферы данных и необходимые константы.

```
   //--- передача параметров кернелу
   if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTFeedForward,
                           def_gptff_keys, Keys.GetOutputs().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTFeedForward,
               def_gptff_outputs, AttentionOut.GetOutputs().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTFeedForward,
                       def_gptff_querys, Querys.GetOutputs().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTFeedForward,
                       def_gptff_scores, Scores.GetOutputs().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTFeedForward,
                       def_gptff_values, Values.GetOutputs().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgument(def_k_GPTFeedForward,
                                        def_gptff_key_size, m_iKeysSize))
      return false;
   if(!m_cOpenCL.SetArgument(def_k_GPTFeedForward,
                                              def_gptff_units, m_iUnits))
      return false;
   if(!m_cOpenCL.SetArgument(def_k_GPTFeedForward,
                                    def_gptff_current, m_iCurrentPosition))
      return false;
```

На этом этап подготовительной работы завершен. Переходим к этапу постановки кернела в очередь выполнения. Здесь мы сначала создаем два динамических массива, в которых укажем смещение и количество выполняемых потоков в каждом подпространстве задач. Затем вызовем метод постановки кернела в очередь выполнения *m_cOpenCL.Execute*.

```
   //--- постановка кернела в очередь выполнения
   int off_set[] = {0};
   int NDRange[] = {m_iHeads};
   if(!m_cOpenCL.Execute(def_k_GPTFeedForward, 1, off_set, NDRange))
      return false;
}
```

На этом мы завершаем работу с методом прямого прохода *CNeuronGPT::FeedForward.* Но нам еще предстоит выполнить аналогичную работу в методе алгоритма обратного прохода *CNeuronGPT::CalcHiddenGradient*.

Напомню, что для реализации метода обратного прохода мы создали два кернела, которые будем вызывать последовательно друг за другом. Следовательно, и работу по обслуживанию кернела нужно повторить для каждого из них.

Сначала создадим буферы данных для первого кернела.

```
bool CNeuronGPT::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
   ......
   for(int layer = m_iLayers - 1; layer >= 0; layer--)
     {
   ......
      //--- разветвление алгоритма по вычислительному устройству
      attention_grad = AttentionOut.GetGradients();
      if(!m_cOpenCL)
        {
         // Блок программы стандартными средствами MQL5
   ......
        }
      else // блок OpenCL
        {
         //--- проверка буферов данных
         if(Values.GetOutputs().GetIndex() < 0)
            return false;
         if(Querys.GetGradients().GetIndex() < 0)
            return false;
         if(Scores.GetOutputs().GetIndex() < 0)
            return false;
         if(attention_grad.GetIndex() < 0)
            return false;
         if(Scores.GetGradients().GetIndex() < 0)
            return false;
         if(m_iScoreTemp < 0)
            return false;
```

Следуя нашему алгоритму работы с контекстом *OpenCL*, после создания буферов данных и передачи всей необходимой информации в память контекста мы передаем в параметры запускаемого кернела указатели на используемые буферы данных и константы для выполнения алгоритма программы.

```
//--- передача параметров кернелу
if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTScoreGradients,
                  def_gptscr_outputs_grad, attention_grad.GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTScoreGradients,
                   def_gptscr_scores, Scores.GetOutputs().GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTScoreGradients,
            def_gptscr_scores_grad, Scores.GetGradients().GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTScoreGradients,
                                def_gptscr_scores_temp, m_iScoreTemp))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTScoreGradients,
                   def_gptscr_values, Values.GetOutputs().GetIndex()))
   return false;
if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTScoreGradients,
            def_gptscr_values_grad, Querys.GetGradients().GetIndex()))
   return false;
if(!m_cOpenCL.SetArgument(def_k_GPTScoreGradients,
                                      def_gptscr_window, m_iKeysSize))
   return false;
if(!m_cOpenCL.SetArgument(def_k_GPTScoreGradients,
                                           def_gptscr_units, m_iUnits))
   return false;
if(!m_cOpenCL.SetArgument(def_k_GPTScoreGradients,
                                def_gptscr_current, m_iCurrentPosition))
   return false;
```

Обратите внимание, что вместо буфера градиентов ошибки тензора значений *Value* мы передаем указатель на буфер градиентов внутреннего нейронного слоя *Querys*. Это вызвано использованием конкатенированного буфера градиентов ошибки для всех трех тензоров. Чтобы исключить последующую операцию копирования данных, мы будем сразу записывать данные в конкатенированный буфер.

После этого мы выполняем операцию постановки кернела в очередь. Напомню, что запускаем мы кернел на выполнение в одномерном пространстве задач в разрезе голов внимания.

Укажем смещение в пространстве задач и количество запускаемых потоков в соответствующих динамических массивах. После этого вызовем метод постановки нашего кернела в очередь задач.

```
//--- постановка кернела в очередь выполнения
int off_set[] = {0};
int NDRange[] = {m_iHeads};
if(!m_cOpenCL.Execute(def_k_GPTScoreGradients, 1, off_set, NDRange))
   return false;
```

На этом работа над первым кернелом завершена, и мы переходим к построению аналогичного алгоритма для второго кернела обратного прохода.

Проверяем дополнительные буферы в памяти контекста *OpenCL*.

```
   if(Querys.GetOutputs().GetIndex() < 0)
      return false;
   if(Keys.GetOutputs().GetIndex() < 0)
      return false;
```

Передаем параметры кернелу.

```
   if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTHiddenGradients,
                           def_gpthgr_keys, Keys.GetOutputs().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTHiddenGradients,
                         def_gpthgr_querys, Querys.GetOutputs().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTHiddenGradients,
                def_gpthgr_querys_grad, Querys.GetGradients().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgumentBuffer(def_k_GPTHiddenGradients,
                def_gpthgr_scores_grad, Scores.GetGradients().GetIndex()))
      return false;
   if(!m_cOpenCL.SetArgument(def_k_GPTHiddenGradients,
                                        def_gpthgr_key_size, m_iKeysSize))
      return false;
   if(!m_cOpenCL.SetArgument(def_k_GPTHiddenGradients,
                                              def_gpthgr_units, m_iUnits))
      return false;
   if(!m_cOpenCL.SetArgument(def_k_GPTHiddenGradients,
                                    def_gpthgr_current, m_iCurrentPosition))
      return false;
```

После этого мы ставим кернел в очередь выполнения. Обратите внимание, что на это раз мы не создаем массивы смещения и размерности пространства задач. Мы просто используем без изменений массивы, созданные при выполнении предыдущего кернела.

```
   if(!m_cOpenCL.Execute(def_k_GPTHiddenGradients, 1, off_set, NDRange))
      return false;
}
```

На этом мы завершаем работу по построению класса модели *GPT* и можем перейти к оценке результатов проделанной работы.

5.3.4 Сравнительное тестирование реализаций

Мы завершили работу еще над одним классом нейронного слоя с использованием механизмов внимания *CNeuronGPT*. В данном классе мы попытались воссоздать модель *GPT* (*Generative Pre-trained Transformer*), предложенную командой *OpenAI* в 2018 году. Данная модель была разработана для решения языковых задач, но позже продемонстрировала довольно высокие результаты и для решения других задач. Третье поколение данной модели (*GPT-3*) на момент написания книги является самой продвинутой языковой моделью.

Отличительной особенностью данной модели от других вариаций модели Трансформер является ее авторегрессионный алгоритм работы. При этом на вход модели подается не весь объем данных, описывающих текущее состояние, а только изменение состояния. В примерах решения языковых задач на вход модели мы можем подавать не весь текст сразу, а по одному слову. При

этом полученное на выходе из модели сгенерированное новое слово является продолжением предложения. Данное слово мы опять подаем на вход модели без повторения предыдущей фразы. А модель сопоставляет его с сохраненными предыдущими состояниями и генерирует новое слово. Такая авторегрессионная модель на практике позволяет генерировать связанные тексты. При этом за счет исключения повторной обработки предыдущих состояний значительно сокращается объем операций в модели без потери качества работы модели.

Мы не будем ставить задачу генерирования новой свечи графика. Для сравнительного анализа работы модели с ранее рассмотренными архитектурными решениями мы оставим прежнюю задачу и ранее используемую обучающую выборку. Но при этом мы усложним задачу для данной модели и на вход будем подавать не весь паттерн как ранее, а лишь небольшую его часть из пяти последних свечей. Для этого немного изменим наш скрипт для тестирования.

Скрипт для данного тестирования мы запишем в файл *gpt_test.mq5*. В качестве шаблона возьмем один из скриптов тестирования предыдущих моделей внимания — *attention_test.mq5*. В начале скрипта создадим константу указания размера паттерна в файле обучающей выборки и внешние параметры настройки скрипта.

```
#define GPT_InputBars         5
#define HistoryBars           40
//+------------------------------------------------------------------+
//| Внешние параметры для работы скрипта                             |
//+------------------------------------------------------------------+
// Имя файла с обучающей выборкой
input string   StudyFileName = "study_data.csv";
// Имя файла для записи динамики ошибки
input string   OutputFileName = "loss_study_gpt.csv";
// Количество исторических баров в одном паттерне
input int      BarsToLine     = 40;
// Количество нейронов входного слоя на 1 бар
input int      NeuronsToBar   = 4;
// Использовать OpenCL
input bool     UseOpenCL      = false;
// Размер пакета для обновления матрицы весов
input int      BatchSize      = 10000;
// Коэффициент обучения
input double   LearningRate   = 0.00003;
// Количество скрытых слоев
input int      HiddenLayers   =  3;
// Количество нейронов в одном скрытом слое
input int      HiddenLayer    =  40;
// Количество циклов обновления матрицы весов
input int      Epochs         =  1000;
```

Как можно заметить, все внешние параметры скрипта перешли из тестовых скриптов прежних моделей. Константа размера паттерна в обучающей выборке нам потребуется для организации правильной загрузки данных, ведь в данной реализации размер передаваемых данных в модель будет сильно отличаться от размера паттерна в обучающей выборке. Я не стал выносить данную константу во внешние параметры, так как мы используем одну обучающую выборку, в связи с чем нам не потребуется изменять параметр при тестировании. В то же время появление дополнительного внешнего параметра может добавить путаницы для пользователя.

После объявления внешних параметров создаваемого тестового скрипта мы подключаем нашу библиотеку для создания моделей нейронных сетей.

```
//+------------------------------------------------------------------+
//| Подключаем библиотеку нейронной сети                             |
//+------------------------------------------------------------------+
#include "..\..\..\Include\NeuroNetworksBook\realization\neuronnet.mqh"
```

На этом мы завершаем создание глобальных переменных и переходим к работе непосредственно над скриптом.

В теле скрипта нам необходимо внести изменения в две функции. Первые изменения мы внесем в функцию описания архитектуры модели *CreateLayersDesc*. Как уже было сказано выше, на вход модели мы будем подавать информацию только о пяти последних свечах. Значит, мы уменьшаем размер слоя исходных данных до 20 нейронов. Но мы сделаем гибкую архитектуру скрипта и укажем размер слоя исходных данных как произведение внешнего параметра количества нейронов на описание одной свечи *NeuronsToBar* и константы количества свечей для загрузки *GPT_InputBars*.

```
bool CreateLayersDesc(CArrayObj &layers)
  {
   CLayerDescription *descr;
//--- создаем слой исходных данных
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   descr.type         = defNeuronBase;
   int prev_count = descr.count = NeuronsToBar * GPT_InputBars;
   descr.window       = 0;
   descr.activation   = AF_NONE;
   descr.optimization = None;
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
```

Обратите внимание, что в случае возникновения ошибки при добавлении объекта в динамический массив мы выводим сообщение для пользователя в журнал и ***обязательно*** удаляем созданные нами объекты перед завершением работы скрипта. Для вас должно стать хорошей практикой обязательная очистка памяти перед завершением программы в любой его точке, будь то нормальное завершение или вследствие ошибки.

После добавления нейронного слоя в динамический массив описаний мы переходим к работе над следующим нейронным слоем. Создаем новый экземпляр объекта для описания нейронного слоя. Мы не можем воспользоваться ранее созданным экземпляром, так как в переменной хранится лишь указатель на объект. Этот же указатель мы передали в динамический массив указателей объектов описания нейронных слоев. А следовательно, при внесении изменений в объект по указателю в локальной переменной, все новые данные отразятся и при обращении к объекту по указателю из динамического массива. Таким образом, используя один указатель, мы

получим в динамическом массиве лишь копии одного указателя, и программа создаст нам модель из одинаковых нейронных слоев, а не желаемую нами архитектуру.

```
//--- блок GPT
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      delete descr;
      return false;
     }
```

Вторым слоем мы создадим блок *GPT*. Об этом модели подскажет константа *defNeuronGPT* в поле типа создаваемого нейронного слоя *type*.

В поле *count* мы укажем размера стека для хранения информации о паттерне. Его значение определит размер буферов для тензоров *Key* и *Value*, а также повлияет на размер вектора коэффициентов зависимости *Score*.

Размер окна исходных данных мы установим на уровне количества элементов в предыдущем слое, которое мы предусмотрительно сохранили в локальную переменную.

Размер вектора описания одного элемента в тензоре *Key* сделаем равным количеству элементов описания одной свечи. Напомню, именно такое значение мы использовали при выполнении предыдущих тестов с моделями внимания. Такой поход поможет нам сделать больший акцент на влиянии самой архитектуры решения, а не используемых параметров.

Остальные параметры мы также перенесем без изменений из скриптов прошлых тестов с моделями внимания. Среди них количество используемых голов внимания и функция оптимизации параметров. Напомню, что функции активации всех внутренних нейронных слоев определены архитектурой Трансформера, поэтому здесь не требуется дополнительная функция для нейронного слоя.

```
      descr.type = defNeuronGPT;
      descr.count = BarsToLine;
      descr.window = prev_count;
      descr.window_out = NeuronsToBar; // Размер вектора Key
      descr.step = 8;                  // Голов внимания
      descr.layers = 4;
      descr.activation = AF_NONE;
      descr.optimization = Adam;
```

Кроме того, при тестировании архитектуры *Multi-Head Self-Attention* мы создавали четыре одинаковых нейронных слоя. Сейчас же для создания подобной архитектуры нам достаточно создать одно описания нейронного слоя и указать в параметре *layers* количество идентичных нейронных слоев.

Созданное описание нейронного слоя добавляем в нашу коллекцию описания архитектуры создаваемой модели.

```
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
```

Далее следует блок скрытых полносвязных нейронных слоев, перенесенный без изменений из скриптов предшествующих тестов, как впрочем и слой результатов. На выходе нашей модели будет слой результатов — полносвязный нейронный слой из двух элементов и линейной функцией активации.

Следующий блок, в который внесем изменения, — функция загрузка обучающей выборки *LoadTrainingData*.

Вначале мы создаем два динамических объекта буферов данных. Один будет использоваться для загрузки описаний паттернов, второй — для целевых значений.

```
bool LoadTrainingData(string path, CArrayObj &data, CArrayObj &result)
  {
   CBufferType *pattern;
   CBufferType *target;
```

После этого открываем для чтения файл с обучающей выборкой. При открытии файла используем флаг *FILE_SHARE_READ*, что позволяет не блокировать файл для чтения другими программами.

```
//--- открываем файл с обучающей выборкой
   int handle = FileOpen(path, FILE_READ | FILE_CSV | FILE_ANSI |
                               FILE_SHARE_READ, ",", CP_UTF8);
   if(handle == INVALID_HANDLE)
     {
      PrintFormat("Error opening study data file: %d", GetLastError());
      return false;
     }
```

Проверяем полученный в результате операции хендл файла.

После успешного открытия файла с обучающей выборкой мы создаем цикл чтения данных до конца файла. Для возможности принудительной остановки скрипта добавим функцию проверки прерывания закрытия программы *IsStopped*.

```
//--- выводим прогресс загрузки данных обучения в комментарий чарта
   uint next_comment_time = 0;
   uint OutputTimeout = 250; // не чаще 1 раза в 250 миллисекунд
//--- организовываем цикл загрузки обучающей выборки
   while(!FileIsEnding(handle) && !IsStopped())
     {
      if(!(pattern = new CBufferType()))
        {
         PrintFormat("Error creating Pattern data array: %d", GetLastError());
         return false;
        }
      if(!pattern.BufferInit(1, NeuronsToBar * GPT_InputBars))
         return false;
      if(!(target = new CBufferType()))
        {
         PrintFormat("Error creating Pattern Target array: %d", GetLastError());
         return false;
        }
      if(!target.BufferInit(1, 2))
         return false;
```

В теле цикла мы создаем новые экземпляры объектов буферов данных для записи отдельных паттернов и их целевых значений, локальные переменные для указателей которых мы уже объявили раньше. И, конечно, не забываем контролировать процесс создание объектов. Иначе увеличивается риск получить критическую ошибку при последующем обращении к создаваемому объекту.

Хочется обратить внимание, что мы будем создавать новые объекты на каждой итерации цикла. Это связано с принципами работы с указателями на экземпляры объектов, описанными немного выше при создании описания модели.

После успешного создания объектов переходим непосредственно к чтению данных. При создании файла обучающей выборки мы сначала записывали описание 40 свечей паттерна и за ними 2 элемента целевых значений. Чтение данных будем осуществлять в той же последовательности. Сначала организуем цикл по чтению вектора описания паттерна. Считывать из файла будем по одному значению в локальную переменную, при этом будем проверять позицию загруженного элемента. Сохранять в буфер данных будем только те элементы, которые попадают в размер нашего окна анализа.

```
      int skip = (HistoryBars - GPT_InputBars) * NeuronsToBar;
      for(int i = 0; i < NeuronsToBar * HistoryBars; i++)
        {
         TYPE temp = (TYPE)FileReadNumber(handle);
         if(i < skip)
            continue;
         pattern.m_mMatrix[0, i - skip] = temp;
        }
```

Аналогичным образом считываем целевые значения, только здесь мы оставляем оба значения.

```
      for(int i = 0; i < 2; i++)
         target.m_mMatrix[0, i] = (TYPE)FileReadNumber(handle);
```

После успешного считывания из файла информации об одном паттерне сохраняем загруженную информацию в динамические массивы нашей базы данных. Информацию о паттерне сохраняем в динамический массив *data*, а целевые значения — в массив *result*.

```
      if(!data.Add(pattern))
        {
         PrintFormat("Error adding study data to array: %d", GetLastError());
         return false;
        }

      if(!result.Add(target))
        {
         PrintFormat("Error adding study data to array: %d", GetLastError());
         return false;
        }
```

Не забываем контролировать процесс выполнения операций.

На данном этапе мы полностью загрузили и сохранили информацию об одном паттерне. Но прежде чем перейти к загрузке информации о следующем паттерне, выведем на график инструмента количество загруженных паттернов для визуального контроля процесса пользователем.

Переходим к следующей итерации цикла.

```
      //--- выводим прогресс загрузки в комментарий чарта
      //--- (не чаще 1 раза в 250 миллисекунд)
      if(next_comment_time < GetTickCount())
        {
         Comment(StringFormat("Patterns loaded: %d", data.Total()));
         next_comment_time = GetTickCount() + OutputTimeout;
        }
     }
   FileClose(handle);
   return(true);
  }
```

После завершения всех итераций цикла два динамических массивах (*data* и *result*) будут содержать всю информацию об обучающей выборке. Можем закрыть файл, а вместе с тем и завершить блок загрузки данных. Завершаем работу функции.

GPT является регрессионной моделью. А значит она чувствительна к последовательности подаваемых на вход элементов. Чтобы удовлетворить такому требованию модели к циклу обучения, применим наработки рекуррентного алгоритма. Случайным образом выбираем лишь первый элемент обучающего батча, а в промежутке между обновлениями параметров модели на вход подаем последовательные паттерны.

```
bool NetworkFit(CNet &net, const CArrayObj &data, const CArrayObj &target,
                                                    VECTOR &loss_history)
  {
//--- обучение
   int patterns = data.Total();
//--- цикл по эпохам
   for(int epoch = 0; epoch < Epochs; epoch++)
     {
      ulong ticks = GetTickCount64();
      //--- обучаем батчами
      //--- выбор случайного паттерна
      int k = (int)((double)(MathRand() * MathRand()) / MathPow(32767.0, 2) *
                                                    (patterns - BarsToLine-1));
      k = fmax(k, 0);
      for(int i = 0; (i < (BatchSize + BarsToLine) && (k + i) < patterns); i++)
        {
         //--- проверим на остановку обучения
         if(IsStopped())
           {
            Print("Network fitting stopped by user");
            return true;
           }
         if(!net.FeedForward(data.At(k + i)))
           {
            PrintFormat("Error in FeedForward: %d", GetLastError());
            return false;
           }
         if(i < BarsToLine)
            continue;
         if(!net.Backpropagation(target.At(k + i)))
           {
            PrintFormat("Error in Backpropagation: %d", GetLastError());
            return false;
           }
        }
      //--- перенастраиваем веса сети
      net.UpdateWeights(BatchSize);
      printf("Use OpenCL %s, epoch %d, time %.5f sec",
                (string)UseOpenCL, epoch, (GetTickCount64() - ticks) / 1000.0);
      //--- сообщим о прошедшей эпохе
      TYPE loss = net.GetRecentAverageLoss();
      Comment(StringFormat("Epoch %d, error %.5f", epoch, loss));
      //--- запомним ошибку эпохи для сохранения в файл
      loss_history[epoch] = loss;
     }
   return true;
  }
```

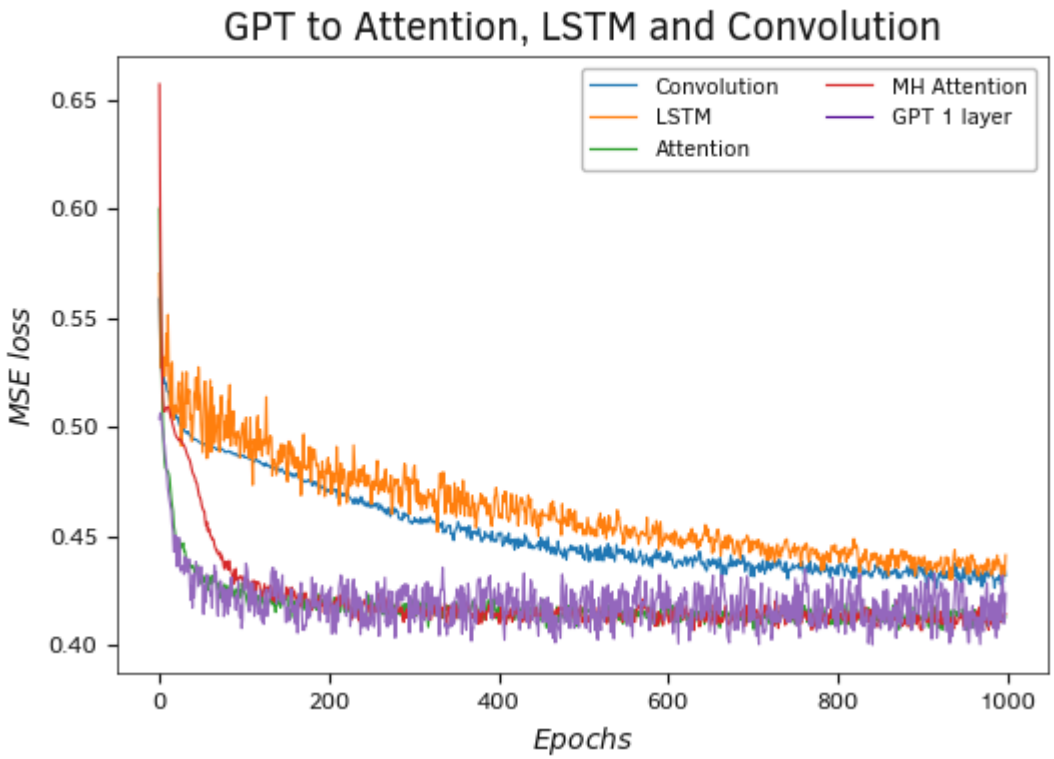
Не забываем контролировать процесс выполнения операций.

Дальнейший код скрипта перенесен без изменений.

Теперь запустим скрипт и сравним результаты с полученными ранее при тестировании предыдущих моделей.

Первое тестирование мы проводили с сохранением всех параметров обучения и одним слоем в блоке *GPT*. График ошибки модели в динамике обучения имеет относительно большие колебания. Что может быть вызвано как неравномерностью данных выборки между обновлениями матриц весов в следствии отсутствия перемешивания данных, так и снижением подаваемых на вход модели данных, что приводит к меньшей передаче градиента ошибки до матрицы весов на каждой итерации прямого прохода. Напомню, что при реализации модели мы обсуждали вопрос передачи градиента ошибки только в рамках текущего состояния.

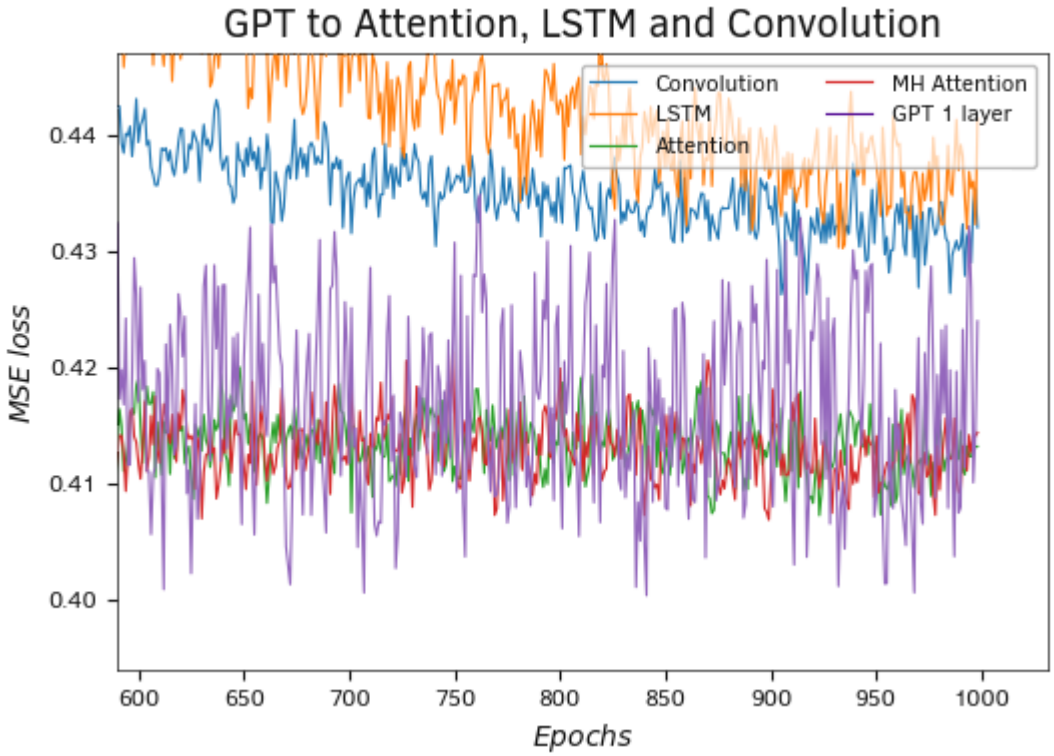
В то же время, несмотря на большой шум, предложенная архитектура повышает планку качества работы модели. Именно она демонстрирует наиболее высокие показатели среди всех рассмотренных моделей.



Тестирование модели GPT

Повышение масштаба графика демонстрирует, насколько хорошо модель понижает планку минимальной ошибки.

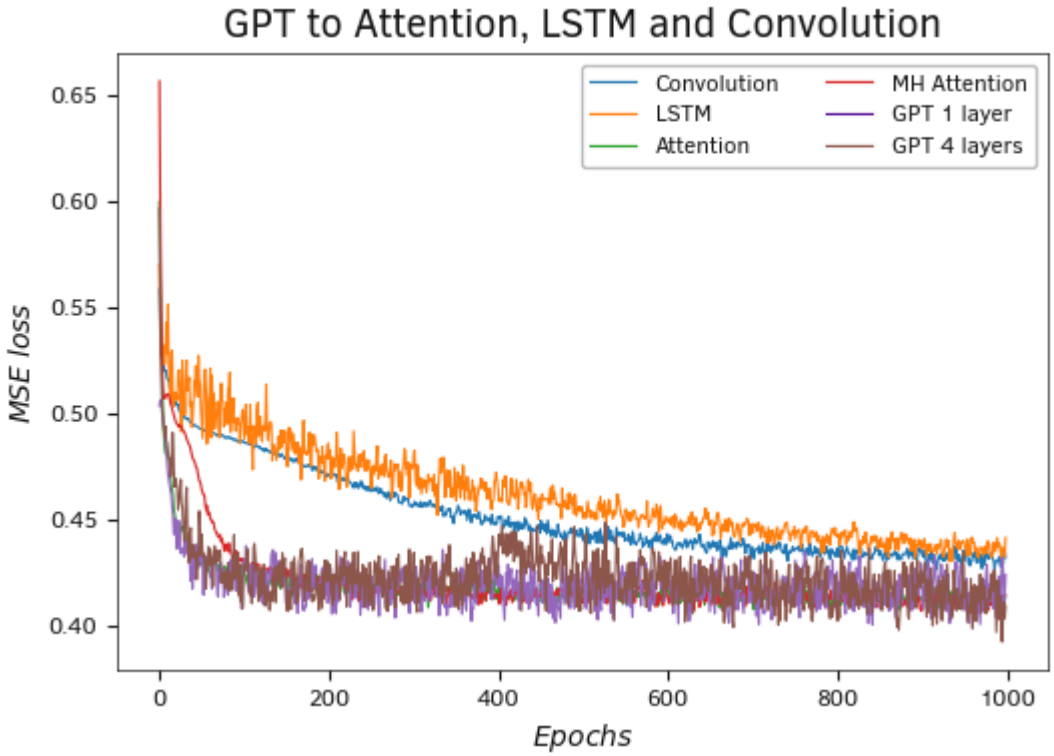
Здесь надо добавить, что при тестировании мы обучали нашу модель «с чистого листа». Авторы архитектуры предлагают проводить предварительное обучение блока *GPT* без учителя на большом объеме данных и только предварительно обученную модель подстраивать под решение конкретных задач в процессе обучения с учителем.



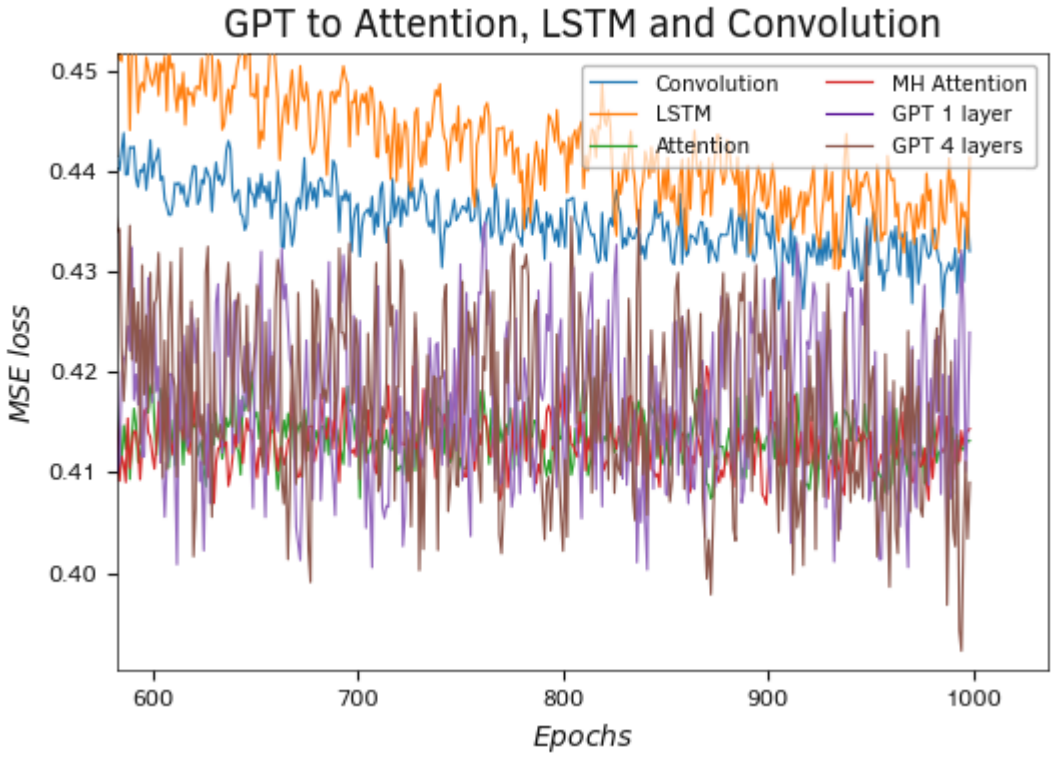
Тестирование модели GPT

Но мы продвигаемся дальше в процессе тестирования нашей реализации. Все известные реализации архитектуры *GPT* используют несколько блоков данной архитектуры. Для следующего тестирования мы увеличили число слоев в блоке *GPT* до четырех. Остальные параметры скрипта мы оставили без изменений.

Результаты тестирования были ожидаемыми. Увеличение числа нейронных слоев неизменно ведет к росту общего числа параметров модели. Большему количеству параметров требуется большее число итераций обновления для достижения оптимального результата. При этом модель учится больше разделять отельные паттерны и более склонна к переобучению. Это и продемонстрировали результаты обучения модели. Мы видим тот же шум на графике ошибки. При этом мы наблюдаем еще большее снижение минимальных показателей ошибки модели.



Тестирование модели GPT 4 слоя



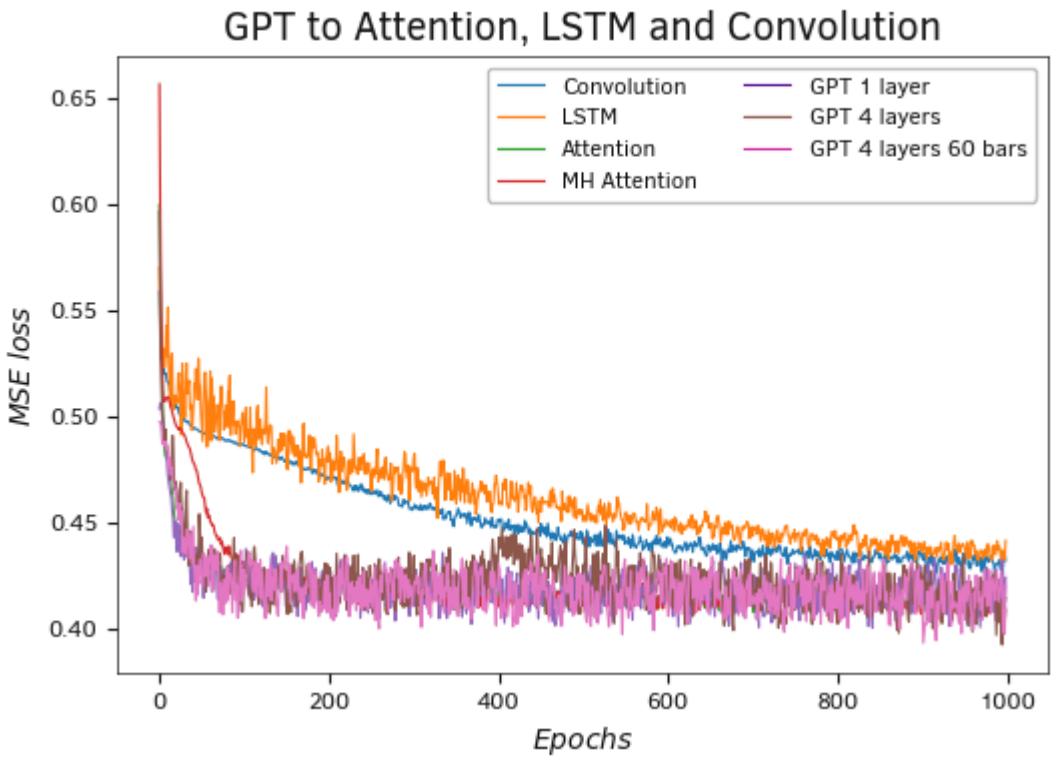
Тестирование модели GPT 4 слоя

Добавим, что из практики использования моделей внимания наиболее четко проявляются их преимущества при использовании длительных последовательностей. И *GPT* не исключение. Скорее даже наоборот. Так как модель пересчитывает только текущее состояние и использует

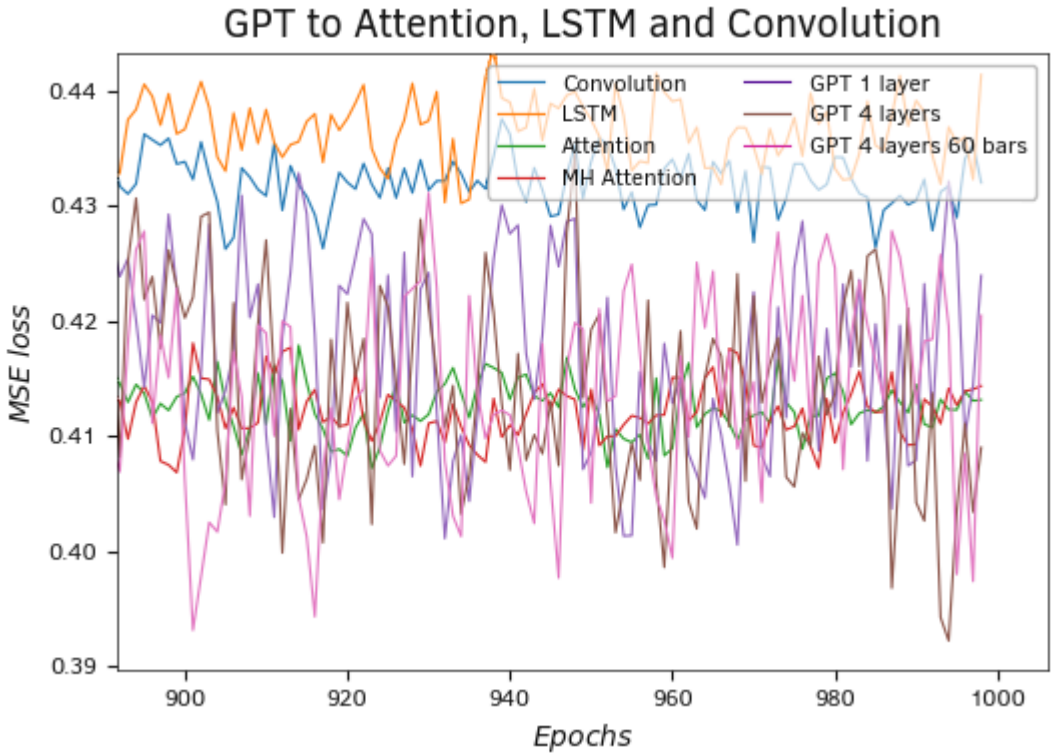
архивные копии предыдущих состояний, это позволяет значительно сократить количество операций при анализе больших последовательностей. А со снижением количества итераций растет и скорость работы всей модели.

Для следующего теста мы увеличили величину стека до 60 свечей. Напомню, что благодаря архитектурному решению *GPT* мы можем увеличить длину анализируемой последовательности только увеличив один внешний параметр без изменения кода программы. В том числе нам не нужно изменять объем данных, подаваемых на вход модели. Надо сказать, что изменения величины стека не ведет к изменению числа параметров модели. Да, увеличение стека тензоров ключей *Key* и значений *Value* влечет за собой увеличение вектора коэффициентов зависимости *Score*. Но при этом абсолютно не меняется ни одна матрица весов внутренних нейронных слоев.

Результаты тестирования продемонстрировали снижение ошибки работы модели. При этом общая тенденция подсказывает, что с большой долей вероятности мы увидим улучшение результатов работы модели в процессе дальнейшего обучения.



Тестирование модели GPT с увеличенным стеком



Тестирование модели GPT с увеличенным стеком

Мы с вами построили еще одну архитектурную модель нейронного слоя. Результаты тестирования модели с использованием нового архитектурного решения демонстрируют значительный потенциал в использовании данного решения. В то же время мы использовали небольшие модели с довольно коротким периодом обучения. Этого достаточно для демонстрации работы архитектурных решений, но не достаточно для использования на реальных данных. Как показывает практика, для получения наилучших результатов требуются различные эксперименты и нестандартные подходы. В большинстве случаев наилучшие результаты кроются за смешиванием различных архитектурных решений.

6. Архитектурные решения повышения сходимости моделей

Мы с вами рассмотрели несколько различных архитектурных решений нейронных слоев. Мы создали классы для реализации рассмотренных архитектурных решений и небольшие модели с их использованием. Но в процессе изучения нейронных сетей мы не можем обойти стороной вопрос повышения сходимости нейронных сетей. Мы рассматривали теоретические аспекты подобных практик, но пока не реализовали их ни в одной модели.

В этой главе мы углубимся в разработку и применение архитектурных решений для повышения сходимости нейронных сетей, акцентируя внимание на пакетной нормализации и методике Dropout. Освещая тему пакетной нормализации, мы начнем с изучения её основных принципов, перейдем к детальному рассмотрению создания класса пакетной нормализации с использованием языка программирования MQL5, включая методы прямого и обратного прохода, а также методы работы с файлами. Также будет затронута тема многопоточных вычислений в контексте пакетной нормализации и представлена реализация данного подхода на языке Python, включая подготовку скрипта для тестирования. Важной частью обсуждения станет сравнительное тестирование моделей с использованием пакетной нормализации, что позволит нам увидеть практическую эффективность рассмотренных подходов.

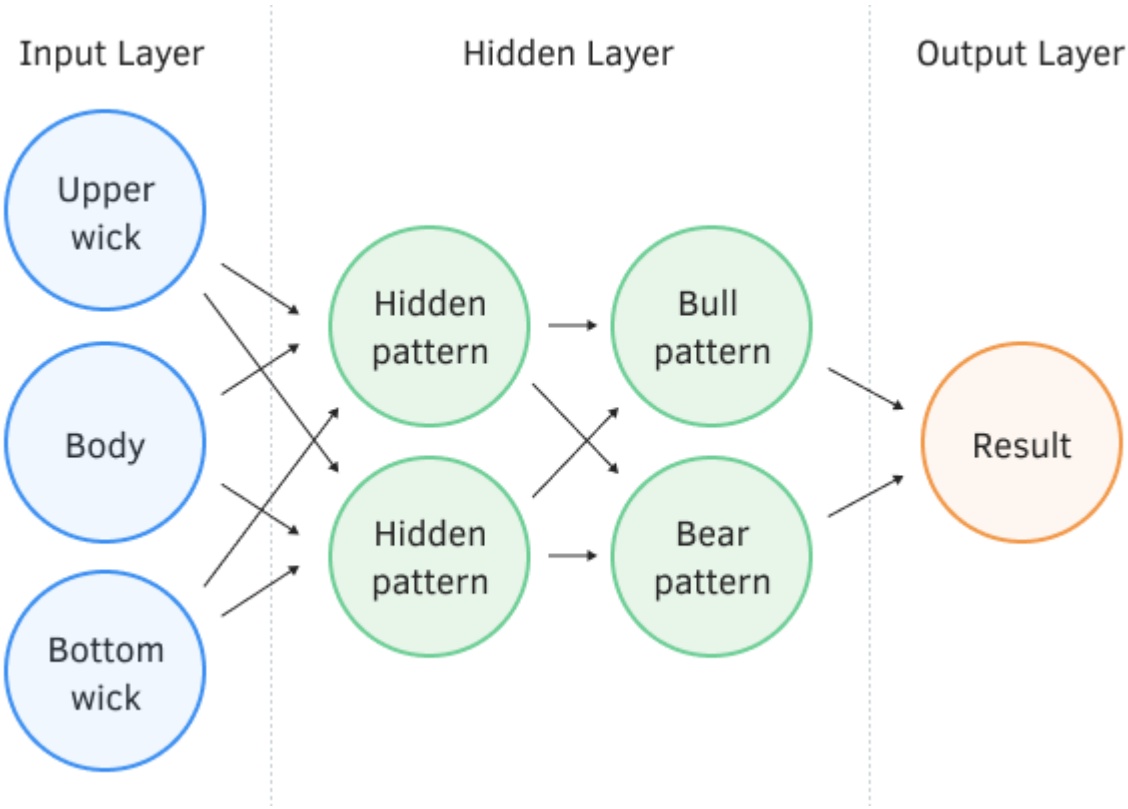
Переходя к теме Dropout, мы подробно рассмотрим его реализацию на MQL5, включая методы прямого прохода, обратного прохода и работы с файлами. Будет затронута тема многопоточных операций при использовании Dropout, а также его реализация на языке Python. Завершая главу, мы проведем сравнительное тестирование моделей с применением Dropout, что позволит нам оценить влияние данной техники на сходимость и эффективность нейронных сетей. Таким образом, мы не только изучим теоретические аспекты этих методик, но и практически применим их для улучшения производительности и сходимости моделей.

6.1 Пакетная нормализация

Одной из таких практик является пакетная нормализация данных. Надо сказать, что нормализация данных довольно часто встречается в моделях нейронных сетей в различных вариантах. Вспомните, когда мы создали свою первую модель полносвязного перцептрона, одним из тестов было сравнение работы модели на обучающей выборке нормализованных и не нормализованных данных. Тестирование показало преимущество использования нормализованных данных.

Еще мы встретили нормализацию данных при изучении моделей внимания. В механизме *Self-Attention* используется нормализация данных на выходе блока внимания и на выходе блока *Feed Forward*. Отличие от предыдущей нормализации заключается в области нормализации данных. Если в первом случае мы брали каждый отдельный параметр и нормализовали его значения в разрезе исторических данных, то во втором случае мы смотрели не на историю значений одного показателя, а наоборот брали все показатели на текущий момент и нормализовали их значения в рамках текущего состояния. Можно сказать, проводили нормализацию данных вдоль временного отрезка и поперек. Первый вариант относится к пакетной нормализации данных, а второй называется нормализацией данных нейронного слоя *Layer Normalization*.

Но возможны и другие точки использования нормализации данных. Напомню основную проблему, решаемую нормализацией данных. Рассмотрим полносвязный перцептрон с двумя скрытыми слоями. При прямом проходе каждый слой генерирует некую совокупность данных, которые служат обучающей выборкой для последующего слоя. Результат работы выходного слоя сравнивается с эталонными данными и на обратном проходе распространяется градиент ошибки от выходного слоя через скрытые слои к исходным данным. Получив на каждом нейроне свой градиент ошибки, мы обновляем весовые коэффициенты, подстраивая нашу нейронную сеть под обучающие выборки последнего прямого прохода. Здесь возникает конфликт: мы подстраиваем второй скрытый слой под выборку данных на выходе первого скрытого слоя, в то время как, изменив параметры первого скрытого слоя, мы уже изменили массив данных. Т. е. мы подстраиваем второй скрытый слой под уже несуществующую выборку данных. Аналогичная ситуация возникает и с выходным слоем, который подстраивается под уже измененный выход второго скрытого слоя. А если еще учесть искажение между первым и вторым скрытыми слоями, то масштабы ошибки увеличиваются. И чем глубже нейронная сеть, тем сильнее проявление этого эффекта. Это явление было названо внутренним ковариационным сдвигом.



В классических нейронных сетях указанная проблема частично решалась уменьшением коэффициента обучения. Небольшие изменения весовых коэффициентов не сильно изменяют распределение выборки на выходе нейронного слоя. Но такой подход не решает проблемы масштабирования с ростом количества слоев нейронной сети и снижает скорость обучения. Еще одна проблема маленького коэффициента обучения — застревание в локальных минимумах.

В феврале 2015 года Сергей Иоффе (*Sergey Ioffe)* и Кристиан Сзегеди (*Christian Szegedy)* предложили метод пакетной нормализации данных (*Batch Normalization*) для решения проблемы внутреннего ковариационного сдвига. Суть метода заключалась в нормализации каждого отдельного нейрона на некоем временном интервале со смещением медианы выборки к нулю и приведением дисперсии выборки к единице.

Эксперименты, проведенные авторами метода, показывают, что применение метода *Batch Normalization* выступает и в роли регуляризатора. Это позволяет отказаться от использования других методов регуляризации, в частности от *Dropout*. Более того, есть более поздние работы, в которых показано, что совместное использование *Dropout* и *Batch Normalization* отрицательно сказывается на результатах обучения нейронной сети.

В современных архитектурах нейронных сетей предложенный алгоритм нормализации можно встретить в различных вариациях. Авторы предлагают использовать *Batch Normalization* непосредственно перед нелинейностью (формулой активации).

6.1.1 Принципы реализации пакетной нормализации

Авторы метода предложили следующий алгоритм проведения нормализации. Вначале по выборке данных считаем среднее значение.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

где:

- μ_B — среднее арифметическое признака по выборке;
- m — размер выборки (batch).

Затем считаем дисперсию исходной выборки.

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)$$

Далее нормализуем данные выборки, приведя выборку к нулевому среднему и единичной дисперсии.

$$\hat{x} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

Обратите внимание, что в знаменателе к дисперсии выборки прибавляется константа ε — небольшое положительное число с целью исключить деление на ноль.

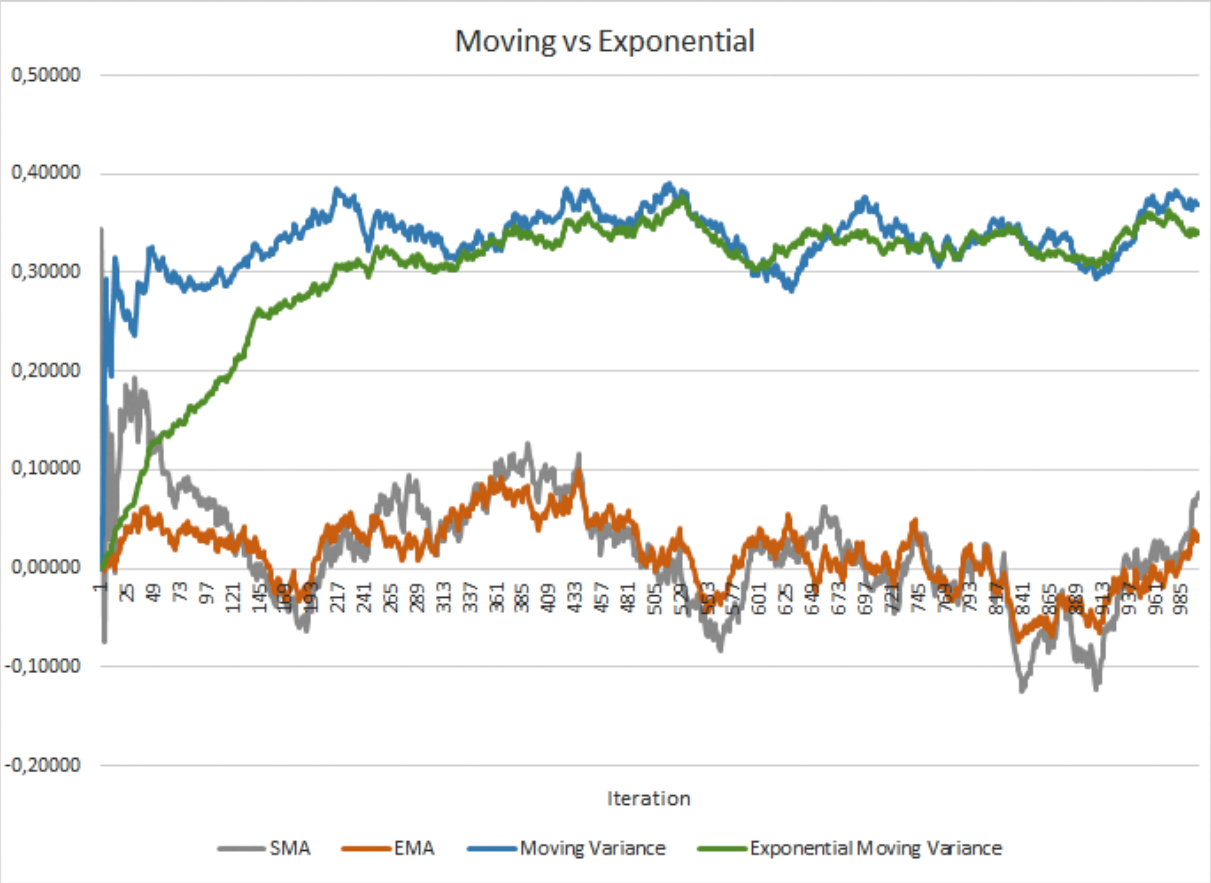
Но как оказалось, такая нормализация может исказить влияние исходных данных. Поэтому авторы метода добавили еще один шаг — масштабирование и смещение. Были введены переменные γ и β, которые обучаются вместе с нейронной сетью методом градиентного спуска.

$$y_i = \gamma\hat{x}_i + \beta \equiv BN_{\gamma\beta}(x_i)$$

Применение данного метода позволяет на каждом шаге обучения получать выборку данных с одинаковым распределением, что на практике делает обучение нейронной сети более стабильным и позволяет увеличить коэффициент обучения. В целом это позволит повысить качество обучения при меньших затратах времени на обучение нейронной сети.

Но в тоже время возрастают затраты на хранение дополнительных коэффициентов. Кроме того, для расчета скользящих средних и дисперсии требуется дополнительное выделение памяти на хранение исторических данных каждого нейрона на весь размер пакета. Тут можно посмотреть в сторону экспоненциальной средней. Для расчета *EMA* достаточно предыдущего значения функции и текущего элемента последовательности.

На рисунке ниже наглядно представлены графики скользящей средней и скользящей дисперсии на 100 элементов в сравнении с экспоненциальной скользящей средней и экспоненциальной скользящей дисперсии на те же 100 элементов. График был построен для 1000 случайных элементов из диапазона от -1,0 до 1,0.



Сравнение графиков скользящей и экспоненциальной средних

Как видно на графике, скользящая средняя и экспоненциальная скользящая средняя сближаются после 120-130 итераций и дальше есть минимальное отклонение, которым можно пренебречь. К тому же график экспоненциальной скользящей средней имеет более сглаженный вид. А вот для расчета *EMA* достаточно предыдущего значения функции и текущего элемента последовательности. Напомню формулу экспоненциальной скользящей средней.

$$\mu_i = \frac{m-1}{m}\mu_{i-1} + \frac{1}{m}x_i = \frac{\mu_{i-1}(m-1) + x_i}{m}$$

где:

- $\boldsymbol{\mu_i}$ — экспоненциальное среднее признака по выборке на *i*-том шаге;
- $\boldsymbol{m}$ — размер выборки (batch);
- $\boldsymbol{x_i}$ — текущее значение показателя.

Для сближения графиков скользящей дисперсии и экспоненциальной скользящей дисперсии потребовалось чуть больше итераций (310–320), но в целом картина похожая. В случае с дисперсией применение экспоненциальной средней дает не только экономию памяти, но и значительно снижает количество вычислений, т.к. для скользящей дисперсии мы бы пересчитывали отклонение от средней для всего пакета исторических данных.

На мой взгляд, использование подобного решения значительно снижает использование ресурсов памяти и затраты времени на проведение математических операций на каждой итерации прямого прохода.

6.1.2 Построение класса пакетной нормализации средствами MQL5

После рассмотрения теоретических аспектов метода нормализации перейдем к практической реализации в рамках нашей библиотеке. Для этого мы создадим новый класс *CNeuronBatchNorm* наследником базового класса полносвязного нейронного слоя *CNeuronBase*.

Для полноценного функционирования нашего класса добавить нужно немного. Добавим всего лишь один буфер для записи параметров нормализации для каждого элемента последовательности и переменную для хранения размера пакета нормализации. В остальном будем использовать буферы базового класса, хотя и с небольшими поправками. Но о них расскажем в ходе реализации методов.

```
class CNeuronBatchNorm    :  public CNeuronBase
  {
protected:
   CBufferType       m_cBatchOptions;
   uint              m_iBatchSize;       // размер пакета

public:
                     CNeuronBatchNorm(void);
                    ~CNeuronBatchNorm(void);
   //---
   virtual bool      Init(const CLayerDescription* description) override;
   virtual bool      SetOpenCL(CMyOpenCL *opencl) override;
   virtual bool      FeedForward(CNeuronBase* prevLayer) override;
   virtual bool      CalcHiddenGradient(CNeuronBase* prevLayer) override;
   virtual bool      CalcDeltaWeights(CNeuronBase* prevLayer, bool read) override;
   //--- методы работы с файлами
   virtual bool      Save(const int file_handle) override;
   virtual bool      Load(const int file_handle) override;
   //--- метод идентификации объекта
   virtual int       Type(void)  override   const { return(defNeuronBatchNorm); }
  };
```

Переопределять мы будем все тот же набор основных методов:

- *Init* — метод инициализации экземпляра класса;
- *FeedForward* — метод прямого прохода;
- *CalcHiddenGradient* — метод распределения градиентов ошибки через скрытый слой;
- *CalcDeltaWeights* — метод распределения градиентов ошибки до матрицы весовых коэффициентов;
- *Save* — метод сохранения параметров нейронного слоя;
- *Load* — метод восстановления работоспособности нейронного слоя из сохраненных данных.

Начнем работу над классом с его конструктора. В данном методе мы лишь задаем начальное значение для размера пакета нормализации. Деструктор класса остается пустым.

```
CNeuronBatchNorm::CNeuronBatchNorm(void)  :  m_iBatchSize(1)
  {
  }
```

После этого переходим к работе над методом инициализации класса. Но прежде чем приступить к реализации этого метода, давайте обратим внимание на нюансы нашей реализации.

Прежде всего, метод нормализации не предусматривает изменение числа элементов. На выходе нейронного слоя будет столько же нейронов, сколько и на входе. Поэтому размер окна исходных данных должен быть равен количеству нейронов в создаваемом слое. Конечно, мы можем проигнорировать параметр размера окна исходных данных и использовать только параметр количества нейронов в слое. Но тогда мы лишаемся дополнительного контроля на стадии инициализации нейронного слоя и должны будем постоянно повторять проверку соответствия количества нейронов при каждом прямом и обратном проходе.

Второй момент связан с отсутствием матрицы весовых коэффициентов в привычном для нас виде. Посмотрим еще раз на математические формулы.

$$\hat{x} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta \equiv BN_{\gamma\beta}(x_i)$$

При расчете нормируемой величины используются только среднее значение и среднеквадратичное отклонение, которые рассчитываются по выборке и не имеют настраиваемые параметры. У нас появляются только два настраиваемых параметра в процессе сдвига и масштабирования значения γ и β. Оба параметра подбираются индивидуально для каждого значения из тензора исходных данных.

А теперь давайте вспомним математическую формулу нейрона со смещением.

$$Output = \sum_{i=1}^{N} w_i x_i + \beta$$

Не кажется ли вам, что при $N = 1$ формулы примут идентичный вид? Этим сходством мы и воспользуемся.

$$Output = wx + \beta \leftrightarrow \gamma \hat{x} + \beta = BN_{\gamma\beta}(x)$$

А теперь вернемся к нашему методу инициализации экземпляра объекта. Данный метод виртуальный, он наследуется от родительского класса. По правилам наследования, данный метод сохраняет тип возвращаемого значения и список параметров метода. В параметрах нашего метода есть только один указатель на объект описания создаваемого нейронного слоя.

В теле метода сразу проверяем полученный указатель на объект описания создаваемого нейронного слоя, а также одновременно проверяем соответствие размера окна исходных данных и количества нейронов в создаваемом слое. Этот момент мы обсуждали немного выше.

После успешного прохождения проверки полученного объекта мы изменим размер окна исходных данных на единицу в соответствии с выведенным выше сходством. Вызовем метод инициализации родительского класса, не забывая проверять результат выполнения операций.

```
bool CNeuronBatchNorm::Init(const CLayerDescription *description)
  {
   if(!description ||
      description.window != description.count)
      return false;
   CLayerDescription *temp = new CLayerDescription();
   if(!temp || !temp.Copy(description))
      return false;
   temp.window = 1;
   if(!CNeuronBase::Init(temp))
      return false;
   delete temp;
```

Здесь надо сказать, что в процессе инициализация родительского класса матрица весовых коэффициентов инициализируется случайными значениями. Однако для пакетной нормализации рекомендуются начальные значение 1 для коэффициента масштабирования γ и 0 для смещения β. В качестве эксперимента мы можем оставить как есть, а можем сейчас заполнить буфер матрицы весов.

```
//--- инициализируем буфер обучаемых параметров
   if(!m_cWeights.m_mMatrix.Fill(0))
      return false;
   if(!m_cWeights.m_mMatrix.Col(VECTOR::Ones(description.count), 0))
      return false;
```

После успешной инициализации объектов родительского класса переходим к созданию объектов и указанию начальных значений переменных и констант нового класса.

Вначале мы инициализируем буфер параметров нормализации. В данном буфере нам потребуется по три элемента для каждого элемента последовательности. В нем мы будем сохранять:

0. μ — среднее значение от предыдущих итераций прямого прохода.
1. σ^2 — дисперсия выборки за предыдущие итерации прямого прохода.
2. $\hat{x}$ — нормализованная величина до масштабирования и сдвига.

Я не случайно привел нумерацию показателей с 0. Именно такую индексацию получат значения в нашем буфере данных. На начальном этапе мы инициализируем весь буфер нулевыми значениями и проверим результат выполнения операций.

```
//--- инициализируем буфер параметров нормализации
   if(!m_cBatchOptions.BufferInit(description.count, 3, 0))
      return false;
   if(!m_cBatchOptions.Col(VECTOR::Ones(description.count), 1))
      return false;
```

В заключение метода инициализации нашего класса сохраним размер пакета нормализации в специально созданную переменную. Выходим из метода с положительным результатом.

```
      m_iBatchSize = description.batch;
   //---
      return true;
     }
```

На этом мы завершаем работу со вспомогательными методами инициализации объекта класса и переходим к построению алгоритмов работы класса. Как всегда, начнем мы эту работу с построения метода прямого прохода.

6.1.2.1 Метод прямого прохода пакетной нормализации

Мы продолжаем движение вперед по пути построения класса пакетной нормализации, а вместе с тем и по пути познания строения и методов организации нейронных сетей. Ранее мы с вами рассматривали различные архитектуры построения нейронных слоев для решения каких-либо практических задач. Работа же слоя пакетной нормализации не менее важна в организации работы нейронной сети, хотя и решаемая им задача не лежит на поверхности. Скорее она скрыта внутри организации процессов самой нейронной сети и служит больше для стабильности работы нашей модели.

Мы уже построили методы инициализации класса, теперь пришло время выстраивать непосредственно алгоритм работы метода. Начинаем мы этот процесс с метода прямого прохода *FeedForward*. Данный метод объявлен виртуальным в базовом классе нейронных слоев нашей библиотеки *CNeuronBase* и переопределяется в каждом новом классе.

Напомню, такой подход позволяет исключить использование диспетчерских методов и функций по перераспределению информационных потоков и вызова различных методов в зависимости от используемого класса объекта. Практически мы можем просто передать указатель на любой объект-наследник в локальную переменную базового класса нейронного слоя и вызвать метод, объявленный в базовом классе. При этом весь диспетчерский функционал система выполнит без нашего участия. Она вызовет метод, относящийся к фактическому типу объекта.

Именно это свойство мы и эксплуатируем, когда в параметрах метода ожидаем получить указатель на объект базового класса нейронного слоя. В то же время в параметрах может быть передан указатель на любой из объектов нейронных слоев нашей библиотеки. Можем с ним работать благодаря использованию переопределенных виртуальных функций.

Работа самого метода прямого прохода начинается с контрольного блока проверки указателей на используемые методом объекты. Здесь мы проверяем как указатель на объект предыдущего слоя, полученный в параметрах, так и указатели на внутренние объекты.

```
bool CNeuronBatchNorm::FeedForward(CNeuronBase *prevLayer)
  {
//--- блок контролей
   if(!prevLayer || !prevLayer.GetOutputs() || !m_cOutputs ||
      !m_cWeights || !m_cActivation)
      return false;
```

Обратите внимание, что наряду с другими объектами мы проверяем и указатель на объект функции активации. Хотя алгоритмом пакетной нормализации не предусматривается использование функции активации. Тем не менее, мы не будем ограничивать возможности пользователя и предоставим ему возможность использования функции активации на его усмотрение. Тем более существуют практические кейсы с применением функции активации после нормализации данных. К примеру, авторы метода рекомендуют использовать нормализацию данных непосредственно перед применением функции активации. На первый взгляд, для применения такого подхода требуется внести изменения в каждый ранее рассмотренный класс. Но мы можем реализовать тот же функционал и без внесения изменений в написанные классы. Нам всего лишь надо объявить требуемый нейронный слой без функции активации, а за ним поставить слой нормализации с требуемой функцией активации. Поэтому использование функции активации в нашем классе считаю вполне обоснованным.

Далее мы сделаем ответвление алгоритма для размера пакета нормализации равного 1 и менее. Надо понимать, что при пакете равном 1 нормализация не осуществляется, и мы просто

передаем тензор исходных данных на выход нейронного слоя. После завершения копирования данных буфера мы взвываем метод активации и выходим из метода, предварительно проверив результаты выполнения операций.

```
//--- проверка размера пакета нормализации
   if(m_iBatchSize <= 1)
     {
      m_cOutputs.m_mMatrix = prevLayer.GetOutputs().m_mMatrix;
      if(m_cOpenCL && !m_cOutputs.BufferWrite())
         return false;
      if(!m_cActivation.Activation(m_cOutputs))
         return false;
      return true;
     }
```

Далее нам предстоит построить алгоритм работы метода. В соответствии с принятой нами концепцией мы будем создавать два варианта реализации алгоритма: стандартными средствами *MQL5* и в режиме многопоточных вычислений с использованием технологии *OpenCL*. Поэтому дальше мы создаем еще одно разветвление алгоритма в зависимости от выбора вычислительного устройства пользователем. В данном разделе мы рассмотрим построение алгоритма стандартными средствами *MQL5*, а к построению алгоритма средствами *OpenCL* мы вернемся в следующих разделах нашей книги.

```
//--- разветвление алгоритма по вычислительному устройству
   if(!m_cOpenCL)
     {
```

Блок операций средствами *MQL5* мы начинаем с небольшой подготовительной работы. Для упрощения процесса доступа к данным мы сохраним в локальную матрицу последовательность исходных данных.

```
      MATRIX inputs = prevLayer.GetOutputs().m_mMatrix;
      if(!inputs.Reshape(1, prevLayer.Total()))
         return false;
```

Согласно алгоритму нормализации данных мы находим среднее значение. При рассмотрении архитектуры нашего решения мы определились использовать экспоненциальное среднее значение, которое определяется по формуле.

$$\mu_i = \frac{\mu_{i-1}(m-1) + x_i}{m}$$

```
      VECTOR mean = (m_cBatchOptions.Col(0) * ((TYPE)m_iBatchSize - 1.0) +
                     inputs.Row(0)) / (TYPE)m_iBatchSize;
```

После определения скользящей средней находим среднюю дисперсию.

```
      VECTOR delt = inputs.Row(0) - mean;
      VECTOR variance = (m_cBatchOptions.Col(1) * ((TYPE)m_iBatchSize - 1.0) +
                         MathPow(delt, 2)) / (TYPE)m_iBatchSize;
```

Когда значения средней и дисперсии найдены, довольно легко вычислить нормализованное значение текущего элемента последовательности.

$$\hat{x} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

```
      VECTOR std = sqrt(variance) + 1e-32;
      VECTOR nx = delt / std;
```

Обратите внимание, что к дисперсии мы прибавляем небольшую константу для исключения ошибки деления на ноль.

Следующий шаг алгоритма пакетной нормализации — сдвиг и масштабирование.

```
      VECTOR res = m_cWeights.Col(0) * nx + m_cWeights.Col(1);
```

После этого нам остается лишь сохранить полученные значения в соответствующие элементы буферов. Обращаю ваше внимание на то, что мы сохраняем не только результат выполнения операций алгоритма в буфер результатов, но и наши промежуточные значения в буфер параметров нормализации. Они нам потребуются при последующих итерациях алгоритма. Не забываем проверить результат выполнения операций.

```
      if(!m_cOutputs.Row(res, 0) ||
         !m_cBatchOptions.Col(mean, 0) ||
         !m_cBatchOptions.Col(variance, 1) ||
         !m_cBatchOptions.Col(nx, 2))
         return false;
      }
    else  // Блок OpenCL
      {
       return false;
      }
```

На этом мы завершаем блок разделения алгоритма в зависимости от используемого вычислительного устройства. Как всегда, для блока *OpenCL* мы установим временную заглушку в виде возврата ложного значения. Вернемся к этому позже.

А сейчас, перед выходом из метода, проведем активацию значений в буфере результатов нашего класса. Для этого достаточно вызвать метод *Activation* нашего специального объекта для работы с функцией активации *m_cActivation*. После проверки результата выполнения операции завершаем работу метода.

```
    if(!m_cActivation.Activation(m_cOutputs))
       return false;
//---
    return true;
   }
```

На этом мы завершаем работу с методом прямого прохода класса пакетной нормализации *CNeuronBatchNorm*. Надеюсь, понимание логики его построения не вызвало у вас сложности. Переходим к построению методов обратного прохода.

6.1.2.2 Методы обратного прохода класса пакетной нормализации

В предыдущих разделах мы начали изучение алгоритма метода пакетной нормализации. Для его реализации в нашей библиотеке мы создали отдельный нейронный слой в виде класса *CNeuronBatchNorm* и даже уже построили методы инициализации класса алгоритма прямого прохода. Теперь пришло время перейти к построению алгоритма обратного прохода для нашего класса. Напомню, что алгоритм обратного прохода во всех нейронных слоях нашей библиотеки представлен четырьмя виртуальными методами:

- метод расчета градиента ошибки на выходе нейронной сети ***CalcOutputGradient***;
- метод распространения градиента через скрытый слой ***CalcHiddenGradient***;
- метод необходимого расчета корректирующих значений для весовых коэффициентов ***CalcDeltaWeights***;
- метод обновления матрицы весовых коэффициентов ***UpdateWeights***.

Все они были объявлены в нашем базовом классе нейронного слоя *CNeuronBase*. Они переопределяются в каждом новом классе по мере необходимости.

В данном классе мы переопределим только два метода: распределения градиента ошибки через скрытый слой и расчета корректирующих значений для весовых коэффициентов.

Мы не будем переопределять метод градиента ошибки на выходе нейронной сети, так как я не знаю сценария, при котором нужно было бы использовать слой пакетной нормализации в качестве последнего слоя нейронной сети. Боле того, эксперименты показывают, что использование пакетной нормализации непосредственно перед слоем результатов нейронной сети может сказаться отрицательно на результатах работы модели.

Что касается метода обновления матрицы весовых коэффициентов, здесь мы специально построили работу буфера матрицы обучаемых параметров таким образом, что для обновления ее параметров стало возможным использование метода родительского класса.

А теперь давайте перейдем к практической части и посмотрим на реализацию указанных выше методов обратного прохода *CalcHiddenGradient*. Это виртуальный метод, который был определен в базовом классе нейронного слоя *CNeuronBase*. Метод переопределяется в каждом новом классе нейронного слоя для реализации конкретного алгоритма. В параметрах метод получает указатель на объект предыдущего нейронного слоя и возвращает логический результат выполнения операций.

В теле метода мы сразу организуем блок контролей, в котором проверим действительность указателей как на объект предыдущего слоя, полученный в параметрах, так и на внутренние объекты, используемые в работе метода. Мы уже не раз говорили о важности такого процесса, так как обращение к объекту по недействительному указателю приводит к критической ошибке и полной остановке выполнения программы.

```
bool CNeuronBatchNorm::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
//--- блок контролей
   if(!prevLayer || !prevLayer.GetOutputs() || !prevLayer.GetGradients() ||
      !m_cActivation || !m_cWeights)
      return false;
```

Далее нам предстоит скорректировать градиент ошибки, полученный от последующего слоя, на производную функции активации нашего слоя. Благодаря тому, что в базовом классе мы

вынесли всю работу с функцией активации в отдельный объект класса *CActivation*, сейчас для корректировки градиента ошибки нам достаточно вызвать соответствующий метод данного класса и указать в его параметрах указатель на буфер градиентов ошибки нашего класса. Как всегда, не забудем проверить результат выполнения операции.

```
//--- корректировка градиента ошибки на производную функции активации
   if(!m_cActivation.Derivative(m_cGradients))
      return false;
```

После этого проверим размер заданного пакета нормализации. Если он не больше единицы, просто копируем данные буфера градиента текущего слоя в буфер предыдущего слоя. Затем выходим из метода с результатом выполнения копирования данных.

```
//--- проверка размера пакета нормализации
   if(m_iBatchSize <= 1)
     {
      prevLayer.GetGradients().m_mMatrix = m_cGradients.m_mMatrix;
      if(m_cOpenCL && !prevLayer.GetGradients().BufferWrite())
         return false;
      return true;
     }
```

Далее последовательно посчитаем градиенты по всем функциям алгоритма.

$$\Delta\hat{x} \rightarrow \Delta\sigma^2 \rightarrow \Delta\mu \rightarrow \Delta x_i$$

Предлагаю пройти по процессу и посмотреть на математические формулы распространения градиента ошибки. На начальном этапе у нас есть градиент ошибки для результатов нашего слоя нормализации, то есть для значения функции масштабирования и сдвига. Напомню формулу:

$$BN_{\gamma\beta}(x_i) = \gamma\hat{x}_i + \beta$$

Для корректировки градиента ошибки нужно умножить его на производную функции. Согласно правилам вычисления производной для $\hat{x}_i$, смещение β выступает в качестве константы, и ее производная равна нулю. А производная от произведения равна второму множителю. Таким образом, наша производная будет равна коэффициенту масштабирования γ.

$$\left(BN_{\gamma\beta}(x_i)\right)' = \gamma$$

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial\hat{x}_i} = \gamma G_i$$

где G_i — градиент i-го элемента на выходе функции масштабирования и сдвига.

В коде метода эта операция будет выражена следующими строками.

```
//--- разветвление алгоритма по вычислительному устройству
   if(!m_cOpenCL)
     {
      MATRIX mat_inputs = prevLayer.GetOutputs().m_mMatrix;
      if(!mat_inputs.Reshape(1, prevLayer.Total()))
         return false;
      VECTOR inputs = mat_inputs.Row(0);
      CBufferType *inputs_grad = prevLayer.GetGradients();
      ulong total = m_cOutputs.Total();
      VECTOR gnx = m_cGradients.Row(0) * m_cWeights.Col(0);
```

Двигаемся дальше. Нормированную величину определяем по формуле.

$$\hat{x} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

Отсюда нам надо распределить градиент ошибки на каждое из составляющих. Я не буду долго выводить формулы частных производных. Приведу лишь готовый формулы расчета градиента ошибки, представленные авторами метода в статье *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*.

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \sigma^2} = \sum_{i=1}^{m} \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \hat{x}_i} * (x_i - \mu) * \frac{-1}{2}(\sigma^2 + \varepsilon)^{-\frac{3}{2}}$$

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \mu} = \left(\sum_{i=1}^{m} \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \hat{x}_i} * \frac{-1}{\sqrt{\sigma^2 + \varepsilon}}\right) + \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \sigma^2} * \frac{\sum_{i=1}^{m} -2(x_i - \mu)}{m}$$

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial x_i} = \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \hat{x}_i} * \frac{1}{\sqrt{\sigma^2 + \varepsilon}} + \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \sigma^2} * \frac{2(x_i - \mu)}{m} + \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \mu} * \frac{1}{m}$$

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \gamma} = \sum_{i=1}^{m} \partial G_i * \hat{x}_i$$

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \beta} = \sum_{i=1}^{m} \partial G_i$$

Последние две формулы нам понадобятся для следующего метода, в котором мы будем распределять градиент ошибки до уровня матрицы обучаемых параметров. Поэтому в коде данного метода мы реализуем только верхние формулы.

```
      VECTOR temp = MathPow(MathSqrt(m_cBatchOptions.Col(1) + 1e-32), -1);
      VECTOR gvar = (inputs - m_cBatchOptions.Col(0)) /
                    (-2 * pow(m_cBatchOptions.Col(1) + 1.0e-32, 3.0 / 2.0)) * gnx;
      VECTOR gmu = temp * (-1) * gnx - gvar * 2 *
                           (inputs - m_cBatchOptions.Col(0)) / (TYPE)m_iBatchSize;
      VECTOR gx = temp * gnx + gmu / (TYPE)m_iBatchSize + gvar * 2 *
                           (inputs - m_cBatchOptions.Col(0)) / (TYPE)m_iBatchSize;
```

Обратите внимание, что в формулах стоят суммы значений по всей выборке нормализации. Мы же выполняем расчеты только для текущего значения. Тем не менее, мы не отступаем от приведенных формул. Дело в том, что наша выборка растянута во времени, а мы возвращаем градиент ошибки на каждом шаге. В период между обновлениями обучаемых параметров класса мы накапливаем на них градиент ошибки и тем самым суммируем его на всем протяжении нашей выборки нормализации, растянутой по временной шкале.

Нам остается лишь сохранить в соответствующий элемент буфера полученный градиент ошибки и проверить результат выполнения операции.

```
      if(!inputs_grad.Row(gx, 0))
         return false;
      if(!inputs_grad.Reshape(prevLayer.Rows(), prevLayer.Cols()))
         return false;
     }
   else  // Блок OpenCL
     {
      return false;
     }
//---
   return true;
  }
```

В результате выполнения операций мы получили заполненный буфер тензора градиентов предыдущего слоя. Значит задача, поставленная перед данным методом, решена, и мы можем завершить блок разделения алгоритма в зависимости от используемого устройства. Для блока организации многопоточных вычислений с использованием технологии *OpenCL* мы временно установим заглушку, как и в аналогичных случаях при работе с другими методами. Завершим работу над нашим методом *CNeuronBatchNorm::CalcHiddenGradient* на данном этапе.

Продолжим организовывать процесс обратного прохода. Переходим к следующему методу — *CNeuronBatchNorm::CalcDeltaWeights*. Обычно этот метод отвечает за распределение градиента ошибки до уровня матрицы весовых коэффициентов. Но в нашем случае мы имеем немного другие обучаемые параметры, на которых и будем распределять градиент ошибки.

Метод *CalcDeltaWeights*, как и предыдущий, в параметрах получает указатель на объект предыдущего слоя. Но в данном случае это скорее выполнение требования наследования методов, чем функциональная необходимость. Выше уже приводились формулы распределения градиента ошибки на обучаемые переменные, но приведу еще раз.

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \gamma} = \sum_{i=1}^{m} \partial G_i * \hat{x}_i$$

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial\beta} = \sum_{i=1}^{m} \partial G_i$$

Как видно из приведенных формул, градиент ошибки параметров не зависит от значений предыдущего слоя. Градиент коэффициента масштабирования зависит от нормализованной величины, а градиент смещения и вовсе равен градиенту ошибки на выходе слоя пакетной нормализации. Конечно, сама нормализованная величина зависит от значений предыдущего слоя. Но для исключения ее повторного вычисления мы просто сохранили значения нормализованных величин в буфер при прямом проходе. Поэтому в теле данного метода мы не будем обращаться к элементам предыдущего слоя. Следовательно, нет смысла тратить время на проверку полученного указателя на предыдущий слой. В то же время мы не будем полностью исключать блок контролей — в нем мы проверяем не только внешние указатели, но и указатели на внутренние объекты.

```
bool CNeuronBatchNorm::CalcDeltaWeights(CNeuronBase *prevLayer, bool read)
  {
//--- блок контролей
   if(!m_cGradients || !m_cDeltaWeights)
      return false;
```

После успешного прохождения блока контролей проверим величину пакета нормализации. Он должен быть как минимум больше единицы. В противном случае выходим из метода.

```
//--- проверка размера пакета нормализации
   if(m_iBatchSize <= 1)
      return true;
```

После успешного прохождения всех контролей мы переходим к непосредственной реализации алгоритма метода. Мы всегда реализовываем алгоритм в двух вариантах: стандартными средствами *MQL5* и с использованием технологии многопоточных вычислений средствами *OpenCL*. Поэтому перед продолжением операций создадим разветвление алгоритма в зависимости от используемого устройства вычислительных операций.

```
//--- разветвление алгоритма по вычислительному устройству
   if(!m_cOpenCL)
     {
```

В ветке реализации алгоритма стандартными средствами *MQL5* мы воспользуемся матричными операциями. По формулам, приведенным выше, определяем градиент ошибки для коэффициента масштабирования и сдвига. Полученные значения прибавляем к ранее накопленным градиентам ошибки соответствующих элементов и обновляем значения буфера накопления градиентов ошибки.

```
      VECTOR grad = m_cGradients.Row(0);
      VECTOR delta = m_cBatchOptions.Col(2) * grad + m_cDeltaWeights.Col(0);
      if(!m_cDeltaWeights.Col(delta, 0))
         return false;
      if(!m_cDeltaWeights.Col(grad + m_cDeltaWeights.Col(1), 1))
         return false;
```

После завершения всех операций мы получим полностью обновленный буфер градиентов ошибки на уровне обучаемых параметров слоя пакетной нормализации. То есть задача метода решена, и мы закрываем блок разделения алгоритма по вычислительному устройству, а вместе с ним и весь

метод. Но предварительно установим заглушку в блоке алгоритма многопоточных вычислений средствами *OpenCL*.

```
      }
    else  // Блок OpenCL
      {
       return false;
      }
//---
   return true;
  }
```

Выше мы переопределили два метода из алгоритма обратного прохода. Метод обновления весовых коэффициентов, а в данном случае обучаемых параметров, унаследовали от родительского класса. Таким образом, работу над методами обратного прохода в части организации процесса стандартными средствами *MQL5* можно считать завершенной. Переходим к методам работы с файлами.

6.1.2.3 Методы работы с файлами

Мы уверенно приближаемся к завершению работы над методами класса пакетной нормализации *CNeuronBatchNorm*. Ранее мы уже построили методы инициализации класса, а также выстроили алгоритм работы прямого и обратного проходов стандартным средствами *MQL5*. Переходим к работе над методами работы с файлами. Мы уже не один раз обсуждали важность наличия и корректного функционирования данных методов, ведь именно от их работы зависит, как быстро мы сможем запустить в опытно-промышленную эксплуатацию однажды обученную модель

Мы уже не один раз выполняли подобную работу для других классов нашей библиотеки. Сейчас мы будем действовать по уже отработанному алгоритму. Вначале мы оцениваем необходимость записи в файл данных каждого элемента класса. В структуре нашего класса мы создавали лишь один новый буфер данных и одну переменную. Оба этих элемента важны для организации корректной работы объектов нашего класса. Поэтому мы будем сохранять оба элемента в файл данных.

```
class CNeuronBatchNorm    :  public CNeuronBase
  {
protected:
   CBufferType       m_cBatchOptions;
   uint              m_iBatchSize;       // размер пакета

public:
                     CNeuronBatchNorm(void);
                    ~CNeuronBatchNorm(void);
   //---
   virtual bool      Init(const CLayerDescription* description) override;
   virtual bool      SetOpenCL(CMyOpenCL *opencl) override;
   virtual bool      FeedForward(CNeuronBase* prevLayer) override;
   virtual bool      CalcHiddenGradient(CNeuronBase* prevLayer) override;
   virtual bool      CalcDeltaWeights(CNeuronBase* prevLayer,bool read)override;
   //--- методы работы с файлами
   virtual bool      Save(const int file_handle) override;
   virtual bool      Load(const int file_handle) override;
   //--- метод идентификации объекта
   virtual int       Type(void)  override   const {return defNeuronBatchNorm;}
  };
```

Определившись с масштабом работы мы приступаем непосредственно к созданию методов работы с файлами нашего класса. Как всегда, первым мы создаем метод записи данных в файл *CNeuronBatchNorm::Save*. Как и все ранее рассмотренные методы, он также создан виртуальным в базовом классе нейронного слоя и переопределяется в каждом новом классе нейронного слоя для полного сохранения всей необходимой информации с целью последующего восстановления корректной работы сохраняемых объектов. В параметрах метод получает хендл файла для записи данных.

```
bool CNeuronBatchNorm::Save(const int file_handle)
  {
//--- вызываем метод родительского класса
   if(!CNeuronBase::Save(file_handle))
      return false;
```

Полученный хендл файла для записи данных мы не проверяем, так как данный контроль уже реализован в одноименном методе родительского класса, который мы вызываем в теле метода. Таким образом, мы проверяем результат выполнения операций метода родительского класса.

```
   if(!CNeuronBase::Save(file_handle))
      return false;
```

Использовать метод родительского класса очень удобно. Такое использование выполняет двойную функцию. Первая функция — контрольная, так как в родительском классе уже реализованы ряд контролей, которые не нужно повторять в новом методе. Достаточно одного вызова метода родительского класса и проверки результата его работы. Вторая — функциональная. В методе родительского класса уже реализовано сохранение всех унаследованных объектов и переменных. Здесь та же ситуация: мы один раз вызываем метод родительского класса и тем самым сразу сохраняем все унаследованные объекты и переменные. Удобно, не правда ли? Более того, нам не надо вызывать метод для каждого отдельного функционала. Одним вызовом мы убиваем двух зайцев: контроль и сохранение унаследованных объектов. Проверка результата выполнения функции подтверждает корректное выполнение обоих функций метода.

После успешного выполнения метода родительского класса мы понимаем, что полученный в параметрах хендл файла действительный. Смело выполняем дальнейшие операции с файлом без риска получения критической ошибки. Сначала сохраняем размер пакета нормализации, имеющийся в переменной *m_iBatchSize*. Не забываем проверить результат выполнения операции.

```
//--- сохраняем размер пакета нормализации
   if(FileWriteInteger(file_handle, m_iBatchSize) <= 0)
      return false;
```

В заключение метода сохраняем буфер параметров нормализации *m_cBatchOptions*. Для этого достаточно вызвать соответствующий метод указанного объекта и проверить результат его работы.

```
//--- сохранение параметров нормализации
   if(!m_cBatchOptions.Save(file_handle))
      return false;
//---
   return true;
  }
```

Как видите, благодаря использованию методов родительского класса и внутренних объектов мы без проблем и довольно кратко описали метод сохранения всей необходимой информации. Основные контроли и операции сохранения данных скрыты в указанных методах.

Аналогичным образом создадим метод загрузки данных из файла *CNeuronBatchNorm::Load*. Надо сказать, что данный метод отвечает не только за чтение данных из файла, но и за полное восстановление функциональности объекта до состояния на момент сохранения данных. Поэтому в методе должны быть предусмотрены операции создания экземпляров объектов, необходимых для корректного функционирования объекта нашего класса пакетной

нормализации. Кроме того, мы должны инициализировать начальными значениями все несохраненные объекты и переменные.

В параметрах метод *CNeuronBatchNorm::Load*, как и предыдущий метод сохранения данных, получает хендл файла с сохраненными данными. Нам предстоит организовать чтение данных из файла в строгом соответствии с последовательностью их записи в файл. На этот раз в теле метода мы сразу вызываем метод родительского класса. Расчет здесь тот же: одним вызовом метода родительского класса мы сразу выполняем весь функционал с унаследованными объектами и переменными. При этом нам достаточно один раз проверить результат выполнения метода родительского класса, чтобы убедиться в корректности выполнения всех его операций.

```
bool CNeuronBatchNorm::Load(const int file_handle)
  {
//--- вызываем метод родительского класса
   if(!CNeuronBase::Load(file_handle))
      return false;
```

После успешного выполнения метода родительского класса перейдем к загрузке данных объектов класса пакетной нормализации. В соответствии с последовательностью записи данных в файл мы сначала считываем размер пакета нормализации.

```
   m_iBatchSize = FileReadInteger(file_handle);
```

В завершение осталось загрузить данные буфера параметров нормализации *m_cBatchOptions*.

```
//--- инициализируем динамический массив параметров оптимизации
   if(!m_cBatchOptions.Load(file_handle))
      return false;
//---
   return true;
  }
```

После успешной загрузки всех данных завершим работу метода с положительным результатом.

Мы закончили работу над созданием слоя пакетной нормализации данных стандартными средствами *MQL5*. Для завершения работы над классом *CNeuronBatchNorm* осталось дополнить его функционал возможностью выполнения многопоточных математических операций с помощью технологии *OpenCL*. Этим мы займемся в следующем разделе. Но уже сейчас у нас есть возможность провести первое тестирование его работы.

6.1.3 Организация многопоточных вычислений в классе пакетной нормализации

Мы продолжаем работу над нашим классом пакетной нормализации *CNeuronBatchNorm*. В предыдущих разделах мы уже полностью реализовали функционал работы класса стандартными средствами *MQL5*. Для завершения работы над классом, согласно нашей концепции, остается дополнить его функциональность возможностью выполнения многопоточных математических операций с использованием технологии *OpenCL*. Напомню, реализацию данного функционала можно условно разделить на два подпроцесса:

- создание программы *OpenCL*;
- внесение изменений в методы основной программы для организации обмена данными с контекстом и вызова программы *OpenCL*.

Начинаем работу с создания программы *OpenCL*. Сначала реализуем кернел прямого прохода *BatchNormFeedForward*. В параметрах кернелу будем передавать указатели на четыре буфера и две константы:

- *inputs* — буфер исходных данных (результатов предыдущего слоя);
- *options* — буфер параметров нормализации;
- *weights* — буфер матрицы обучаемых параметров (назван по аналогии с буфером класса);
- *output* — буфер результатов;
- *batch* — размер пакета нормализации;
- *total* — размер буфера результатов.

```
__kernel void BatchNormFeedForward(__global TYPE *inputs,
                                   __global TYPE *options,
                                   __global TYPE *weights,
                                   __global TYPE *output,
                                   int batch,
                                   int total)
  {
```

Последний параметр необходим, потому что для оптимизации процесса вычислений мы используем векторные переменные типа *TYPE4*. Подобный подход позволяет распараллелить вычисления не на программном уровне, а на уровне микропроцессора. Использование вектора из четырех элементов типа *double* позволяет полностью заполнить 256-битный регистр микропроцессора и за один такт произвести вычисления над всем вектором. Таким образом, за один такт микропроцессора мы осуществляем операции над четырьмя элементами нашего массива данных. OpenCL поддерживает векторные переменные из 2, 3, 4, 8 и 16 элементов. Перед выбором размерности вектора ознакомьтесь с техническими характеристиками вашего оборудования.

В теле кернела мы сразу определяем идентификатор текущего потока. Он нам понадобится для определения смещения в буферах тензоров до анализируемых переменных.

И тут же проверяем размер пакета нормализации. Если он не больше единицы, просто копируем соответствующие элементы из буфера градиентов текущего слоя в буфер градиентов предыдущего слоя и прекращаем дальнейшее выполнение кернела.

```
    int n = get_global_id(0);
    if(batch <= 1)
      {
       D4ToArray(output, ToVect4(inputs, n * 4, 1, total, 0), n * 4, 1, total, 0);
       return;
      }
```

Обратите внимание, что при вызове функции обращения значений тензора в векторное представление и обратно для параметра смещения в тензоре мы увеличиваем идентификатор потока в четыре раза. Это связано с тем, что при использовании векторных операций с *TYPE4* каждый поток одновременно обрабатывает четыре элемента тензора. Следовательно, запущенных потоков будет в четыре раза меньше размера обрабатываемого тензора.

Если же размер пакета нормализации больше единицы, и при этом мы продолжаем выполнение программы, то необходимо определить смещение в буферах тензоров параметров нормализации с учетом идентификатора текущего потока и размера вектора осуществления операций (*TYPE4*)

```
int shift = n * 4;
int shift_options = n * 3 * 4;
int shift_weights = n * 2 * 4;
```

Переходим непосредственно к выполнению нашего алгоритма. Сначала создадим вектор с анализируемыми исходными данными и посчитаем экспоненциальное среднее. Будем ориентироваться на предыдущее среднее значение и дисперсию для определения первой итерации. Разделим предварительно полученное значение для усреднения на размер пакета выборки только на второй и последующих итерациях. Это объясняется тем, что среднее значение из первого элемента и есть сам элемент.

После определения среднего значения найдем отклонение текущего значения от среднего и вычислим дисперсию выборки.

```
TYPE4 inp = ToVect4(inputs, shift, 1, total, 0);
TYPE4 mean = ToVect4(options, shift, 3, total * 3, 0) * ((TYPE)batch - 1) + inp ;
if(options[shift_options ] != 0 && options[shift_options + 1] > 0)
   mean /= (TYPE4)batch;
TYPE4 delt = inp - mean;
TYPE4 variance = ToVect4(options, shift, 3, total * 3, 1) * ((TYPE)batch - 1) + pc
if(options[shift_options + 1] > 0)
   variance /= (TYPE4)batch;
```

Когда есть среднее значение и дисперсия выборки, можно легко посчитать нормализованную величину параметра.

```
TYPE4 nx = delt / sqrt(variance + 1e-37f);
```

Далее, согласно алгоритму пакетной нормализации, необходимо осуществить сдвиг и масштабирование нормализованного значения. Но прежде хочу напомнить, что на начальном этапе мы инициализировали буфер матрицы обучаемых параметров нулевыми значениями. В таком виде мы получим 0 для всех значений независимо от полученного ранее нормализованного значения.

$$y_i = 0\hat{x}_i + 0 = 0$$

Поэтому проверяем наличие нулевого значения для коэффициента масштабирования и при необходимости заменяем его на единицу.

```
if(weights[shift_weights] == 0)
   D4ToArray(weights, (TYPE4)1, shift, 2, total * 2, 0);
```

Обратите внимание, что мы проверяем на равенство нулю только первый элемент анализируемого вектора значений. При этом заменяем на единицу весь вектор. Такой подход считаю допустимым, так как ожидаю получить нулевые значения только при первом проходе. В этот момент у нас будут все элементы буфера равны нулю, и их нужно заменить. Дальше коэффициенты уже будут определены, в ходе обучения модели они будут оптимизироваться и, следовательно, будут отличны от нуля.

После такой нехитрой операции мы можем смело осуществить масштабирование и сдвиг.

```
    TYPE4 res = ToVect4(weights, shift, 2, total * 2, 0) * nx +
                ToVect4(weights, shift, 2, total * 2, 1);
```

Теперь остается лишь сохранить полученные данные в соответствующие элементы буферов. При этом мы сохраняем не только последний результат, но и необходимые нам промежуточные значения.

```
    D4ToArray(options, mean, shift, 3, total * 3, 0);
    D4ToArray(options, variance, shift, 3, total * 3, 1);
    D4ToArray(options, nx, shift, 3, total * 3, 2);
    D4ToArray(output, res, shift, 1, total, 0);
  }
```

На этом мы завершаем работу с кернелом прямого прохода *BatchNormFeedForward* и переходим к работе по созданию кернелов обратного прохода.

Для реализации алгоритма обратного прохода создадим два кернела, один для распределения градиента ошибки до уровня предыдущего слоя, а второй для распределения градиента ошибки до уровня матрицы обучаемых параметров.

Начнем мы с создания кернела распределения градиента ошибки через скрытый слой нейронной сети *BatchNormCalcHiddenGradient*. В параметрах данного метода будем передавать уже пять буферов данных и две константы:

- *inputs* — буфер исходных данных (результатов предыдущего слоя);
- *options* — буфер параметров нормализации;
- *weights* — буфер матрицы обучаемых параметров (назван по аналогии с буфером класса);
- *gradient* — буфер градиентов ошибки на уровне результатов текущего слоя;
- *gradient_inputs* — буфер градиентов ошибки на уровне результатов предыдущего слоя (в данном случае результат работы кернела);
- *batch* — размер пакета нормализации;
- *total* — размер буфера результатов.

```
__kernel void BatchNormCalcHiddenGradient(__global TYPE *options,
                                          __global TYPE *gradient,
                                          __global TYPE *inputs,
                                          __global TYPE *gradient_inputs,
                                          __global TYPE *weights,
                                          int batch,
                                          int total)
  {
```

В начале кернела, как и в кернеле прямого прохода, определяем идентификатор текущего потока и проверяем размер пакета нормализации. Если размер пакета нормализации не больше единицы, то просто копируем градиенты ошибки из буфера текущего слоя в буфер предыдущего слоя и прекращаем выполнение кернела.

```
int n = get_global_id(0);
int shift = n * 4;
if(batch <= 1)
  {
   D4ToArray(gradient_inputs, ToVect4(gradient, shift, 1, total, 0),
                                                 shift, 1, total, 0);
   return;
  }
```

Если же размер пакета нормализации больше единицы, и при этом мы продолжаем выполнение операций кернела, то предстоит распределить градиент ошибки по всей цепочке от уровня результатов текущего слоя до уровня результатов предыдущего слоя. Напомню математические формулы, которые нам предстоит реализовать.

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \hat{x}_i} = \gamma G_i$$

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \sigma^2} = \sum_{i=1}^{m} \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \hat{x}_i} * (x_i - \mu) * \frac{-1}{2}(\sigma^2 + \varepsilon)^{-\frac{3}{2}}$$

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \mu} = \left(\sum_{i=1}^{m} \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \hat{x}_i} * \frac{-1}{\sqrt{\sigma^2 + \varepsilon}}\right) + \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \sigma^2} * \frac{\sum_{i=1}^{m} -2(x_i - \mu)}{m}$$

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial x_i} = \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \hat{x}_i} * \frac{1}{\sqrt{\sigma^2 + \varepsilon}} + \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \sigma^2} * \frac{2(x_i - \mu)}{m} + \frac{\partial BN_{\gamma\beta}(x_i)}{\partial \mu} * \frac{1}{m}$$

```
TYPE4 inp = ToVect4(inputs, shift, 1, total, 0);
TYPE4 gnx = ToVect4(gradient, shift, 1, total, 0) *
            ToVect4(weights, shift, 2, total * 2, 0);
TYPE4 temp = 1 / sqrt(ToVect4(options, shift, 3, total * 3, 1) + 1e-37f);
TYPE4 delt = inp - ToVect4(options, shift, 3, total * 3, 0);
TYPE4 gvar = delt / (-2 * pow(ToVect4(options, shift, 3, total * 3, 1) +
                                                 1.0e-37f, 3.0f / 2.0f)) * gnx;
TYPE4 gmu = (-temp) * gnx - gvar * 2 * delt / (TYPE4)batch;
TYPE4 gx = temp * gnx + gmu/(TYPE4)batch + gvar * 2 * delt/(TYPE4)batch;
```

После расчетов сохраняем результат операций и завершаем выполнение кернела.

```
  D4ToArray(gradient_inputs, gx, shift, 1, total, 0);
}
```

На этом мы завершаем работу над первым кернелом реализации алгоритма обратного прохода нашего класса пакетной нормализации данных и переходим к заключительной фазе работы над программой *OpenCL* — созданию второго кернела обратного прохода распределения градиента ошибки до уровня матрицы обучаемых параметров *BatchNormCalcDeltaWeights*.

В параметрах данному кернелу будем передавать три буфера данных:

- *options* — буфер параметров нормализации;

- *delta_weights* — буфер градиентов ошибки на уровне матрицы обучаемых параметров (в данном случае результат работы кернела);
- *gradient* — буфер градиентов ошибки на уровне результатов текущего слоя.

```
__kernel void BatchNormCalcDeltaWeights(__global TYPE *options,
                                        __global TYPE *delta_weights,
                                        __global TYPE *gradients)
  {
```

В данном кернеле нужно реализовать только две математические формулы:

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \gamma} = \sum_{i=1}^{m} \partial G_i * \hat{x}_i$$

$$\frac{\partial BN_{\gamma\beta}(x_i)}{\partial \beta} = \sum_{i=1}^{m} \partial G_i$$

Как видите, операции довольно простые и не потребуют много кода для реализации алгоритма. На это раз мы даже не стали использовать векторные операции.

В начале кернела определяем идентификатор текущего потока и вместе с ним смещение в буферах тензоров параметров нормализации. Смещение в буферах градиентов ошибки будет соответствовать идентификатору потока.

```
  const int n = get_global_id(0);
  int shift_options = n * 3;
  int shift_weights = n * 2;
```

Чтобы сократить обращение к глобальной памяти, сначала сохраним значение градиента ошибки в локальную переменную, а затем вычислим и сразу запишем в соответствующие элементы буфера накопления градиентов ошибки соответствующие значения текущего шага по формулам, указанным выше.

```
  TYPE grad = gradients[n];
  delta_weights[shift_weights] += grad * options[shift_options + 2];
  delta_weights[shift_weights + 1] += grad;
 }
```

Как видите, градиенты ошибок записаны в соответствующие элементы буфера. А значит, задача, поставленная перед данным кернелом, выполнена, и мы можем завершить его работу.

Таким образом, мы с вами реализовали все три кернела для организации прямого и обратного проходов в нашем классе пакетной нормализации данных. Теперь можем перейти к внесению изменений в основную программу для организации обмена данными с контекстом *OpenCL* и вызова соответствующего кернела программы.

Начнем эту работу, как обычно, с создания констант работы с кернелами программы *OpenCL*. Переходим в файл *defines.mqh* и в начало добавляем константы идентификации кернелов программы.

```
#define def_k_BatchNormFeedForward        37
#define def_k_BatchNormCalcHiddenGradient 38
#define def_k_BatchNormCalcDeltaWeights   39
```

А затем добавляем идентификаторы параметров кернелов.

```
//--- прямой проход пакетной нормализации
#define def_bnff_inputs                0
#define def_bnff_options               1
#define def_bnff_weights               2
#define def_bnff_outputs               3
#define def_bnff_batch                 4
#define def_bnff_total                 5

//--- распределение градиента через слой пакетной нормализации
#define def_bnhgr_options              0
#define def_bnhgr_gradient             1
#define def_bnhgr_inputs               2
#define def_bnhgr_gradient_inputs      3
#define def_bnhgr_weights              4
#define def_bnhgr_batch                5
#define def_bnhgr_total                6

//--- распределение градиента до оптимизируемых параметров пакетной нормализации
#define def_bndelt_options             0
#define def_bndelt_delta_weights       1
#define def_bndelt_gradient            2
```

На следующем этапе надо инициализировать новые кернелы в программе. Для этого переходим в метод инициализации программы *OpenCL* основного класса-диспетчера нашей модели *CNet::InitOpenCL*. Сначала изменяем общее количество используемых кернелов.

```
   if(!m_cOpenCL.SetKernelsCount(40))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
   if(!m_cOpenCL.KernelCreate(def_k_BatchNormFeedForward,
                              "BatchNormFeedForward"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
   if(!m_cOpenCL.KernelCreate(def_k_BatchNormCalcHiddenGradient,
                              "BatchNormCalcHiddenGradient"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
   if(!m_cOpenCL.KernelCreate(def_k_BatchNormCalcDeltaWeights,
                              "BatchNormCalcDeltaWeights"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
```

Теперь, когда кернелы созданы, а мы можем к ним обращаться, переходим к работе с методами нашего класса пакетной нормализации.

По уже сложившейся традиции начнем работу с метода прямого прохода. Мы вносим изменения только в части реализации алгоритма многопоточных операций с использованием технологии *OpenCL*. Весь остальной код метода остается без изменений.

В соответствии с алгоритмом подготовки к запуску кернела, нужно сначала передать все необходимые данные в память контекста *OpenCL*. Поэтому мы проверяем наличие созданных буферов в памяти контекста.

```
bool CNeuronBatchNorm::FeedForward(CNeuronBase *prevLayer)
  {
   ......
//--- разветвление алгоритма по вычислительному устройству
   if(!m_cOpenCL)
     {
//--- Реализация средствами MQL5
   ......
     }
   else  // Блок OpenCL
     {
      //--- проверка буферов данных
      CBufferType *inputs = prevLayer.GetOutputs();
      if(inputs.GetIndex() < 0)
         return false;
      if(m_cBatchOptions.GetIndex() < 0)
         return false;
      if(m_cWeights.GetIndex() < 0)
         return false;
      if(m_cOutputs.GetIndex() < 0)
         return false;
```

На следующем шаге передаем в параметры кернела указатели на буферы данных и значения необходимых констант.

```
      //--- передача параметров кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormFeedForward,
                                      def_bnff_inputs, inputs.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormFeedForward,
                                   def_bnff_weights, m_cWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormFeedForward,
                              def_bnff_options, m_cBatchOptions.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormFeedForward,
                                   def_bnff_outputs, m_cOutputs.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_BatchNormFeedForward,
                                   def_bnff_total, (int)m_cOutputs.Total()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_BatchNormFeedForward,
                                             def_bnff_batch, m_iBatchSize))
         return false;
```

После выполнения подготовительной работы приступаем к постановке кернела в очередь выполнения. Но прежде нужно заполнить два динамических массива. В одном заполняем размерность пространства задач, а во втором — смещение в каждом измерении пространства задач. Запускать кернел будем в одномерном пространстве задач с нулевым смещением. Количество запускаемых потоков будет в 4четыре раза меньше размерности тензора результатов текущего слоя. Но так как размерность тензора не всегда будет кратной четырем, а нам необходимо произвести вычисления для всех элементов тензора результатов, то мы

предусмотрим дополнительный поток, который выполнит вычисления для «хвостовой» части тензора, которая не кратна четырем.

После расчета количества потоков и заполнения буферов вызовем метод постановки кернела в очередь выполнения.

```
      //--- постановка в очередь выполнения
       uint off_set[] = {0};
       uint NDRange[] = { (int)(m_cOutputs.Total() + 3) / 4 };
       if(!m_cOpenCL.Execute(def_k_BatchNormFeedForward, 1, off_set, NDRange))
          return false;
      }
//---
   if(!m_cActivation.Activation(m_cOutputs))
      return false;
//---
   return true;
  }
```

На этом мы завершаем работу с методом прямого прохода, в котором мы уже реализовали полный функционал, включая возможность организации параллельных вычислений на *GPU* с использованием технологии *OpenCL*.

Переходим к методам обратного прохода, в которых нам предстоит провести аналогичную работу. При реализации метода обратного прохода средствами *MQL5* мы переопределили два метода. Следовательно, оба метода мы нужно дополнить функционалом многопоточных вычислений. Сначала добавим функционал в метод распределения градиента ошибки до предыдущего нейронного слоя *CNeuronBatchNorm::CalcHiddenGradient*. Как и в методе прямого прохода, сначала мы создаем необходимые буферы данных в контексте *OpenCL*.

```
bool CNeuronBatchNorm::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
   ......
//--- разветвление алгоритма по вычислительному устройству
   if(!m_cOpenCL)
     {
//--- Реализация средствами MQL5
   ......
     }
   else  // блок OpenCL
     {
      //--- проверка буферов данных
      CBufferType* inputs = prevLayer.GetOutputs();
      CBufferType* inputs_grad = prevLayer.GetGradients();
      if(inputs.GetIndex() < 0)
         return false;
      if(m_cBatchOptions.GetIndex() < 0)
         return false;
      if(m_cWeights.GetIndex() < 0)
         return false;
      if(m_cOutputs.GetIndex() < 0)
         return false;
      if(m_cGradients.GetIndex() < 0)
         return false;
      if(inputs_grad.GetIndex() < 0)
         return false;
```

Затем, согласно нашему алгоритму реализации многопоточных вычислений с использованием технологии *OpenCL*, мы передаем параметры вызываемого кернела.

```
      //--- передача параметров кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormCalcHiddenGradient,
                                           def_bnhgr_inputs, inputs.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormCalcHiddenGradient,
                                        def_bnhgr_weights, m_cWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormCalcHiddenGradient,
                                    def_bnhgr_options, m_cBatchOptions.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormCalcHiddenGradient,
                                      def_bnhgr_gradient, m_cGradients.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormCalcHiddenGradient,
                                def_bnhgr_gradient_inputs, inputs_grad.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_BatchNormCalcHiddenGradient,
                                        def_bnhgr_total, (int)m_cOutputs.Total()))
         return false;
      if(!m_cOpenCL.SetArgument(def_k_BatchNormCalcHiddenGradient,
                                                  def_bnhgr_batch, m_iBatchSize))
         return false;
```

После передачи всех параметров мы готовим кернел к постановке в очередь выполнения. Напомню, что при создании кернела мы определили использование векторных операций с типом *TYPE4*. Соответственно, мы сокращаем в четыре раза количество выполняемых потоков. Вызываем метод постановки кернела в очередь.

```
      //--- постановка в очередь выполнения
      int off_set[] = {0};
      int NDRange[] = { (int)(m_cOutputs.Total() + 3) / 4 };
      if(!m_cOpenCL.Execute(def_k_BatchNormCalcHiddenGradient, 1, off_set, NDRange))
         return false;
     }
//---
   return true;
  }
```

На этом мы завершаем работу с методом распределения градиента ошибки через скрытый слой *CNeuronBatchNorm::CalcHiddenGradient*. Нам остается повторить операции для второго метода обратного прохода *CNeuronBatchNorm::CalcDeltaWeights*.

И снова мы повторяем алгоритм постановки кернела в очередь. На этот раз кернел использует три буфера данных.

```
bool CNeuronBatchNorm::CalcDeltaWeights(CNeuronBase *prevLayer, bool read)
  {
   ......
//--- разветвление алгоритма по вычислительному устройству
   if(!m_cOpenCL)
     {
//--- Реализация средствами MQL5
   ......
     }
   else
     {
      //--- проверка буферов данных
      if(m_cBatchOptions.GetIndex() < 0)
         return false;
      if(m_cGradients.GetIndex() < 0)
         return false;
      if(m_cDeltaWeights.GetIndex() < 0)
         return false;
```

Затем передаем указатели на созданные буферы в параметры запускаемого кернела.

```
      //--- передача параметров кернелу
      if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormCalcDeltaWeights,
                           def_bndelt_delta_weights, m_cDeltaWeights.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormCalcDeltaWeights,
                                def_bndelt_options, m_cBatchOptions.GetIndex()))
         return false;
      if(!m_cOpenCL.SetArgumentBuffer(def_k_BatchNormCalcDeltaWeights,
                                  def_bndelt_gradient, m_cGradients.GetIndex()))
         return false;
```

Отправляем кернел в очередь выполнения. На этот раз количество потоков будет равно количеству элементов в тензоре результатов нашего слоя пакетной нормализации.

```
      //--- постановка в очередь выполнения
      int off_set[] = {0};
      int NDRange[] = {(int)m_cOutputs.Total()};
      if(!m_cOpenCL.Execute(def_k_BatchNormCalcDeltaWeights, 1, off_set, NDRange))
         return false;
      if(read && !m_cDeltaWeights.BufferRead())
         return false;
     }
//---
   return true;
  }
```

На этом мы заканчиваем работу с нашим классом пакетной нормализации данных *CNeuronBatchNorm*. Он готов к использованию — сейчас в нем полностью реализован алгоритм нормализации данных. При этом мы повторили алгоритм в двух вариантах: стандартными средствами *MQL5* и с использованию технологии многопоточных вычислений *OpenCL*. Тем самым мы предоставляем пользователю право выбора используемой технологии в соответствии с его требованиями.

Сейчас я предлагаю посмотреть на реализацию метода пакетной нормализации на языке *Python*.

6.1.4 Реализация пакетной нормализации на Python

Мы уже познакомились с алгоритмом пакетной нормализации данных и даже реализовали слой пакетной нормализации для нашей библиотеки на языке *MQL5*. В том числе мы добавили возможность использования технологии многопоточных вычислений с помощью *OpenCL*. Теперь давайте посмотрим, какой вариант реализации данного метода предлагает знакомая нам библиотека *Keras* для *TensorFlow.*

Данная библиотека предлагает к использованию слой *tf.keras.layers.BatchNormalization*.

```
tf.keras.layers.BatchNormalization(
    axis=-1, momentum=0.99, epsilon=0.001, center=True, scale=True,
    beta_initializer='zeros', gamma_initializer='ones',
    moving_mean_initializer='zeros',
    moving_variance_initializer='ones', beta_regularizer=None,
    gamma_regularizer=None, beta_constraint=None, gamma_constraint=None, **kwargs
)
```

Пакетная нормализация применяет преобразование для поддержания среднего значения результатов около нуля, а стандартное отклонение — около единицы.

Важно отметить, что слой пакетной нормализации работает по-разному во время обучения и во время логического вывода.

В период обучения слой нормализует свои выходные данные, используя среднее значение и стандартное отклонение текущего пакета исходных данных. Для каждого нормализуемого канала слой возвращает:

$$BN_{Out}^{Train} = \frac{\gamma * (X - \bar{x})}{\sqrt{\sigma_X^2 + \varepsilon}} + \beta$$

где:

- ε — небольшая константа (настраивается как часть аргументов конструктора) для исключения ошибки деления на ноль;
- γ — это обучаемый коэффициент масштабирования (инициализированный как 1), который можно отключить установкой параметра *scale=False* в конструкторе объекта;
- β — это обучаемый коэффициент смещения (инициализированный как 0), который можно отключить установкой параметра *center=False* в конструкторе объекта;
- X — тензор пакета исходных данных;
- $\bar{x}$ — среднее значение по пакету данных;
- σ_X^2 — дисперсия пакета данных.

В процессе эксплуатации слой нормализует свои выходные данные, используя скользящее среднее значение и стандартное отклонение пакетов, которые он видел во время обучения.

$$BN_{Out} = \frac{\gamma * (x - \bar{x}_{Train})}{\sqrt{\sigma_{Train}^2 + \varepsilon}} + \beta$$

Таким образом, слой будет нормализовать исходные данные во время вывода только после обучения на данных, которые имеют аналогичные статистические показатели.

В конструктор слоя можно передавать следующие аргументы:

- *axis* — целое число, ось, которая должна быть нормализована (обычно это ось признаков);
- *momentum* — импульс для скользящей средней;
- *epsilon* — небольшая константа для исключения ошибки деления на ноль;
- *center* — при True добавляет смещение *beta* к нормализованному тензору, при False *beta* игнорируется;
- *scale* — при True умножается на *gamma*, при False *gamma* не используется. Когда следующий слой является линейным, его можно отключить, поскольку масштабирование будет выполняться следующим слоем;
- *beta_initializer* — тип инициализатора бета-веса;
- *gamma_initializer* — тип инициализатора гамма-веса;
- *moving_mean_initializer* — тип инициализатора для скользящей средней;
- *moving_variance_initializer* — тип инициализатора для скользящей средней дисперсии;
- *beta_regularizer* — дополнительный регуляризатор для бета-веса;
- *gamma_regularizer* — дополнительный регуляризатор для гамма-веса;
- *beta_constraint* — необязательное ограничение для бета-веса;
- *gamma_constraint* — необязательное ограничение для гамма-веса.

При обращении к слою возможно использование следующих параметров:

- *inputs* — тензор исходных данных, допускается использование тензора любого ранка;
- *training* — логический флаг, указывающий на режим работы слоя: обучение или эксплуатация (различие режимов работы указано выше);
- *input_shape* — используется для описания размерности исходных данных в случае указания слоя первым в модели.

На выходе слой выдает тензор результатов с сохранением размерности исходных данных.

Кроме того, конструкция слоя позволяет использовать настройку *layer.trainable*, которая блокирует изменение параметров в процессе обучения. Это не обязательно, и обычно это означает, что слой работает в режиме вывода. Для включения этого режима чаще всего используется параметр training, который может быть передан при вызове слоя. Но нужно понимать, что «Замораживание параметров» и «Режим вывода» — это два разных понятия.

Однако в случае слоя *BatchNormalization* установка *trainable = False* означает, что слой будет впоследствии запускаться в режиме логического вывода. Это значит, что он будет использовать скользящее среднее значение и скользящую дисперсию для нормализации текущей партии, вместо среднего значения и дисперсии текущей выборки.

Такое поведение было добавлено в *TensorFlow 2.0*, чтобы обеспечить при *layer.trainable = False* наиболее часто ожидаемое поведение в случае тонкой настройки.

Обратите внимание, что установка *trainable* для модели, содержащей другие слои, рекурсивно установит *trainable* значение всех внутренних слоев.

Если значение *trainable* атрибута изменяется после компиляции модели, новое значение не вступает в силу для этой модели до тех пор, пока модель не будет заново перекомпилирована.

6.1.4.1 Подготовка скрипта для тестирования пакетной нормализации

Чтобы проанализировать влияние пакетной нормализации на результат, возьмем самые простые модели с полносвязным перцептроном. Напомню, что одним из самых первых наших тестов была проверка влияния предварительной нормализации исходных данных на результат работы модели. Тогда мы сделали вывод о важности нормализации исходных данных и во всех последующих моделях использовали нормализованные исходные данные. Но, как вы понимаете, предварительная нормализация исходных данных всегда требует затрат и не совсем удобна для работы на финансовых рынках, когда исходные данные идут сплошным потоком. В таком случае нормализацию исходных данных надо прописывать в коде программы. При изменении выборки, будь то влияние времени или изменение анализируемого инструмента, потребуется изменение программного кода или внешних параметров, которые придется определять вне модели. А это дополнительные затраты. И конечно после этого нужно будет дообучать модель. Поэтому было бы логично найти возможность включить процесс нормализации данных в модель и обновлять ее параметры в процессе обучения модели. Не кажется ли вам, что рассматриваемая нами модель пакетной нормализации подойдет для решения этой задачи? Это и будет наш первый тест.

Для проведения такого эксперимента мы возьмем скрипт тестирования моделей перцептрона *perceptron.py* и создадим его копию с именем *batch_norm.py*. Внесем в него небольшие изменения.

В начале скрипта мы как обычно импортируем необходимые библиотеки.

```
# Импорт библиотек
import os
import pandas as pd
import numpy as np
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
import MetaTrader5 as mt5
```

Перед обучением нам необходимо загрузить обучающие выборки, которые лежат в песочнице терминала *MetaTrader 5*. Чтобы определить путь в песочницу, мы подключаемся к терминалу и получаем путь к папке данных терминала. К полученному пути добавляем *MQL5\Files*. В результате мы получим путь в песочницу терминала. Если вы сохранили данные обучающей выборки в подкаталог, его также надо добавить к нашему пути в песочницу. Теперь можно отключиться от терминала. Создаем две локальные переменные с полным путем к фалам обучающих выборок, один с нормализованными данными, второй с ненормализованными.

```
# Загрузка обучающей выборки
if not mt5.initialize():
    print("initialize() failed, error code =",mt5.last_error())
    quit()

path=os.path.join(mt5.terminal_info().data_path,r'MQL5\Files')
mt5.shutdown()
filename = os.path.join(path,'study_data.csv')
filename_not_norm = os.path.join(path,'study_data_not_norm.csv')
```

Сначала мы загрузим данные нормированной выборки.

```
data = np.asarray( pd.read_table(filename,
                   sep=',',
                   header=None,
                   skipinitialspace=True,
                   encoding='utf-8',
                   float_precision='high',
                   dtype=np.float64,
                   low_memory=False))
```

И разделим загруженные данные на паттерны и цели. Напомню, что при создании обучающей выборки мы записывали в файл всю информацию о паттерне в одну строку. При этом в каждая строка содержит информацию только об одном паттерне. Последние два элемента в строке содержат целевые значения паттерна. Воспользуемся этим свойством и определим количество элементов во втором измерении нашего массива данных. Отняв от полученного значения количество элементов на целевые значения мы получаем количество элементов описания одного паттерна. Используя эту информацию разделим данные на два массива.

```
# Разделение обучающей выборки на исходные данные и цели
targets=2
inputs=data.shape[1]-targets
train_data=data[:,0:inputs]
train_target=data[:,inputs:]
```

После этого мы аналогично загрузим и разделим данные не нормированной обучающей выборки.

```
#загрузка ненормированной обучающей выборки
data = np.asarray( pd.read_table(filename_not_norm,
                   sep=',',
                   header=None,
                   skipinitialspace=True,
                   encoding='utf-8',
                   float_precision='high',
                   dtype=np.float64,
                   low_memory=False))

# Разделение ненормированной обучающей выборки на исходные данные и цели
train_nn_data=data[:,0:inputs]
train_nn_target=data[:,inputs:]

del data
```

После разделения данных обучающей выборки на два тензора удалим объект исходных данных, чтобы более эффективно использовать наши ресурсы.

Следующим этапом после загрузки данных мы приступаем к созданию моделей нейронных сетей для проведения тестирования.

Первой мы создадим небольшой полносвязный перцептрон с одним скрытым слоем на 40 элементов и слоем результатов из 2 элементов.

```
# Создание первой модели с одним скрытым слоем
model1 = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
                           keras.layers.Dense(40, activation=tf.nn.swish),
                           keras.layers.Dense(targets, activation=tf.nn.tanh)
```

```
                                ])
```

После этого мы создаем объект обратного вызова для раннего прекращения обучения если ошибка модели на обучающей выборке не будет снижаться более пяти эпох обучения. Компилируем модель. При компиляции мы указываем метод оптимизации параметров *Adam* и среднеквадратичное отклонение в качестве функции ошибки обучения модели. Дополнительно к функции ошибки для отслеживания качества обучения мы добавим метрику *Accuracy*, которая показывает долю правильных ответов модели.

```
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=5)
model1.compile(optimizer='Adam',
               loss='mean_squared_error',
               metrics=['accuracy'])
model1.summary()
```

Следом мы создаем вторую модель, в которой просто добавим слой пакетной нормализации между слоем исходных данных и скрытым слоем модели.

```
# Добавление пакетной нормализации для исходных данных
# в модель с одним скрытым слоем
model1bn = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
                             keras.layers.BatchNormalization(),
                             keras.layers.Dense(40, activation=tf.nn.swish),
                             keras.layers.Dense(targets, activation=tf.nn.tanh)
                            ])
```

И скомпилируем модель с теми же параметрами.

```
model1bn.compile(optimizer='Adam',
               loss='mean_squared_error',
               metrics=['accuracy'])
model1bn.summary()
```

Модели для нашего первого эксперимента готовы.

Во втором эксперименте я бы хотел оценить влияние использования пакетной нормализации внутри сети между скрытыми слоями модели. Для проведения этого эксперимента мы так же создадим полносвязные перцептроны, но уже с тремя аналогичными скрытыми слоями. В первой мы создадим модель без использования пакетной нормализации. Просто возьмем первую модель из этого скрипта и добавим в нее два скрытых слоя, аналогичных первому скрытому слою. Слои исходных данных и результатов остаются без изменений.

```
# Создание модели с тремя скрытыми слоями
model2 = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
                           keras.layers.Dense(40, activation=tf.nn.swish),
                           keras.layers.Dense(40, activation=tf.nn.swish),
                           keras.layers.Dense(40, activation=tf.nn.swish),
                           keras.layers.Dense(targets, activation=tf.nn.tanh)
                          ])
```

Для чистоты эксперимента компилировать модель будем с теми же параметрами.

```
model2.compile(optimizer='Adam',
               loss='mean_squared_error',
               metrics=['accuracy'])
model2.summary()
```

Теперь добавим по слою пакетной нормализации перед каждым скрытым слоем. Обратите внимание, что мы ***не добавляем*** слой пакетной нормализации перед слоем результатов, так как авторы метода не рекомендуют использовать пакетную нормализацию перед слоем результатов. На проведенных ими экспериментах это только ухудшило результаты моделей.

```
# Добавление пакетной нормализации для исходных данных и скрытых слоев второй модели
model2bn = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
                             keras.layers.BatchNormalization(),
                             keras.layers.Dense(40, activation=tf.nn.swish),
                             keras.layers.BatchNormalization(),
                             keras.layers.Dense(40, activation=tf.nn.swish),
                             keras.layers.BatchNormalization(),
                             keras.layers.Dense(40, activation=tf.nn.swish),
                             keras.layers.Dense(targets, activation=tf.nn.tanh)
                            ])
```

Как и ранее, компиляцию модели осуществляем без изменения параметров.

```
model2bn.compile(optimizer='Adam',
               loss='mean_squared_error',
               metrics=['accuracy'])
model2bn.summary()
```

Теперь, когда все модели собраны, можно приступать к их обучению. Все модели будем обучать с одинаковыми параметрами. Для обучения модели мы будем использовать пакеты из 1000 паттернов между обновлениями весовых матриц, обучение будет длиться 500 эпох, если не наступит событие раннего выхода из обучения. Для валидации данных будет использоваться последние 10% обучающей выборки. При этом в процессе обучения паттерны будут перемешиваться.

Сначала проведем обучение модели с одним скрытым слоем на нормализованных данных.

```
# Обучение первой модели на ненормализованных данных
history1 = model1.fit(train_data, train_target,
                      epochs=500, batch_size=1000,
                      callbacks=[callback],
                      verbose=2,
                      validation_split=0.1,
                      shuffle=True)
model1.save(os.path.join(path,'perceptron1.h5'))
```

Следом обучим ту же модель на ненормализованных данных.

```
# Обучение первой модели на ненормализованных данных
history1nn = model1.fit(train_nn_data, train_nn_target,
                      epochs=500, batch_size=1000,
                      callbacks=[callback],
                      verbose=2,
                      validation_split=0.1,
```

```
                        shuffle=True)
```

В третий раз мы проведем обучение аналогичной модели с использованием слоя пакетной нормализации между исходными данными и скрытым слоем. Обучение будем проводить на обучающей выборке без нормализации.

```
history1bn = model1bn.fit(train_nn_data, train_nn_target,
                        epochs=500, batch_size=1000,
                        callbacks=[callback],
                        verbose=2,
                        validation_split=0.1,
                        shuffle=True)
model1bn.save(os.path.join(path,'perceptron1bn.h5'))
```

Результаты первых двух обучений послужат нам ориентирами для оценки работы модели со слоем пакетной нормализации.

На данном этапе мы получим достаточно информации, чтобы сделать вывод по первому эксперименту: нормализацию данных на этапе подготовки данных можно заменить слоем пакетной нормализации данных между исходными данными и обучаемой моделью.

Перейдем к работе над вторым экспериментом и выясним влияние использования слоя пакетной нормализации перед скрытым слоем модели на процесс обучения и результат работы обученной модели в целом. Для этого нам потребуется провести обучение еще двух моделей.

Сначала мы проведем обучение модели с тремя скрытыми слоями на предварительно нормализованных данных. Для обучения модели мы используем те же самые параметры обучения.

```
history2 = model2.fit(train_data, train_target,
                        epochs=500, batch_size=1000,
                        callbacks=[callback],
                        verbose=2,
                        validation_split=0.1,
                        shuffle=True)
model2.save(os.path.join(path,'perceptron2.h5'))
```

Следом проведем обучение модели на ненормализованной обучающей выборке, но уже с использованием слоя пакетной нормализации перед каждым скрытым слоем. В том числе, слой пакетной нормализации используется и перед первым скрытым слоем после слоя исходных данных.

```
history2bn = model2bn.fit(train_nn_data, train_nn_target,
                        epochs=500, batch_size=1000,
                        callbacks=[callback],
                        verbose=2,
                        validation_split=0.1,
                        shuffle=True)
model2bn.save(os.path.join(path,'perceptron2bn.h5'))
```

После обучения двух моделей у нас будет достаточно информации для выводов по результатам второго эксперимента. Для наглядности мы построим графики изменения ошибки на этапе обучения и валидации параметров обучения в зависимости от количества эпох обучения.

Сначала построим график изменения среднеквадратичного отклонения данных работы наших моделей от целевых данных для первого эксперимента.

```
# Отрисовка результатов обучения моделей с одним скрытым слоем
plt.plot(history1.history['loss'], label='Normalized inputs train')
plt.plot(history1.history['val_loss'], label='Normalized inputs validation')
plt.plot(history1nn.history['loss'], label='Unnormalized inputs train')
plt.plot(history1nn.history['val_loss'], label='Unnormalized inputs vvalidation')
plt.plot(history1bn.history['loss'],
                        label='Unnormalized inputs\nvs BatchNormalization train')
plt.plot(history1bn.history['val_loss'],
                    label='Unnormalized inputs\nvs BatchNormalization validation')
plt.ylabel('$MSE$ $loss$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics\n1 hidden layer')
plt.legend(loc='upper right', ncol=2)
```

В дополнение к первому графику построим график динамики изменения метрики *Accuracy*.

```
plt.figure()
plt.plot(history1.history['accuracy'], label='Normalized inputs train')
plt.plot(history1.history['val_accuracy'], label='Normalized inputs validation')
plt.plot(history1nn.history['accuracy'], label='Unnormalized inputs train')
plt.plot(history1nn.history['val_accuracy'], label='Unnormalized inputs validation')
plt.plot(history1bn.history['accuracy'],
                            label='Unnormalized inputs\nvs BatchNormalization train')
plt.plot(history1bn.history['val_accuracy'],
                        label='Unnormalized inputs\nvs BatchNormalization validation')
plt.ylabel('$Accuracy$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics\n1 hidden layer')
plt.legend(loc='lower right', ncol=2)
```

Построим аналогичные графики для вывода результатов второго эксперимента.

```
# Отрисовка результатов обучения моделей с тремя скрытыми слоями
plt.figure()
plt.plot(history2.history['loss'], label='Normalized inputs train')
plt.plot(history2.history['val_loss'], label='Normalized inputs validation')
plt.plot(history2bn.history['loss'],
                  label='Unnormalized inputs\nvs BatchNormalization train')
plt.plot(history2bn.history['val_loss'],
              label='Unnormalized inputs\nvs BatchNormalization validation')
plt.ylabel('$MSE$ $loss$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics\n3 hidden layers')
plt.legend(loc='upper right', ncol=2)

plt.figure()
plt.plot(history2.history['accuracy'], label='Normalized inputs train')
plt.plot(history2.history['val_accuracy'], label='Normalized inputs validation')
plt.plot(history2bn.history['accuracy'],
                      label='Unnormalized inputs\nvs BatchNormalization train')
plt.plot(history2bn.history['val_accuracy'],
                  label='Unnormalized inputs\nvs BatchNormalization validation')
plt.ylabel('$Accuracy$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics\n3 hidden layers')
plt.legend(loc='lower right', ncol=2)
```

Итак, на данном этапе мы обучили все модели на обучающей выборке. Обучающая выборка для нас представляет собой исторические данные. То, что модель может аппроксимировать исторические данные, конечно, хорошо. Но нам бы хотелось, чтобы модель хорошо работала в реальном времени. Чтобы проверить, как модель поведет себя на неизвестных данных, проверим работу моделей на тестовой выборке.

Загрузим тестовые выборки точно так же, как мы загружали обучающие выборки. Сначала загрузим данные нормализованной тестовой выборки.

```
# Загрузка тестовой выборки
test_filename = os.path.join(path,'test_data.csv')
test = np.asarray( pd.read_table(test_filename,
                   sep=',',
                   header=None,
                   skipinitialspace=True,
                   encoding='utf-8',
                   float_precision='high',
                   dtype=np.float64,
                   low_memory=False))
```

Разделим загруженные данные на паттерны и целевые значения.

```
# Разделение тестовой выборки на исходные данные и цели
test_data=test[:,0:inputs]
test_target=test[:,inputs:]
```

Потом повторим алгоритм для загрузки ненормализованной тестовой выборки.

```
test_filename = os.path.join(path,'test_data_not_norm.csv')
test = np.asarray( pd.read_table(test_filename,
                   sep=',',
                   header=None,
                   skipinitialspace=True,
                   encoding='utf-8',
                   float_precision='high',
                   dtype=np.float64,
                   low_memory=False))

# Разделение тестовой выборки на исходные данные и цели
test_nn_data=test[:,0:inputs]
test_nn_target=test[:,inputs:]

del test
```

После копирования данных удаляем массив исходных данных, что позволит нам более эффективно распоряжаться нашими ресурсами.

Далее мы проверим работу всех моделей на тестовых выборках. Работу моделей без слоев пакетной нормализации мы проверяем на нормализованных данных. Модели с использованием слоев пакетной нормализации мы протестируем на ненормализованных данных тестовой выборки.

```
# Проверка результатов моделей на тестовой выборке
test_loss1, test_acc1 = model1.evaluate(test_data, test_target, verbose=2)
test_loss1bn, test_acc1bn = model1bn.evaluate(test_nn_data, test_nn_target,
                                              verbose=2)
test_loss2, test_acc2 = model2.evaluate(test_data, test_target, verbose=2)
test_loss2bn, test_acc2bn = model2bn.evaluate(test_nn_data, test_nn_target,
                                              verbose=2)
```

Результаты тестирования мы выведем в журнал.

```
# Вывод результатов тестирования в журнал
print('Model 1 hidden layer')
print('Test accuracy:', test_acc1)
print('Test loss:', test_loss1)

print('Model 1 hidden layer with BatchNormalization')
print('Test accuracy:', test_acc1bn)
print('Test loss:', test_loss1bn)

print('Model 3 hidden layers')
print('Test accuracy:', test_acc2)
print('Test loss:', test_loss2)

print('Model 3 hidden layer with BatchNormalization')
print('Test accuracy:', test_acc2bn)
print('Test loss:', test_loss2bn)
```

Для наглядности сделаем графическое представление результатов отдельно по среднеквадратичному отклонению и по метрике *Accuracy*.

```
plt.figure()
plt.bar(['1 hidden layer','1 hidden layer\nvs BatchNormalization',
         '3 hidden layers','3 hidden layers\nvs BatchNormalization'],
        [test_loss1,test_loss1bn,test_loss2,test_loss2bn])
plt.ylabel('$MSE$ $Loss$')
plt.title('Test results')

plt.figure()
plt.bar(['1 hidden layer','1 hidden layer\nvs BatchNormalization',
         '3 hidden layers','3 hidden layers\nvs BatchNormalization'],
        [test_acc1,test_acc1bn,test_acc2,test_acc2bn])
plt.ylabel('$Accuracy$')
plt.title('Test results')

plt.show()
```

После построения графиков вызовем команду их отрисовки на экране пользователя.

На этом мы заканчиваем работу над скриптом, который позволяет протестировать, как использование слоя пакетной нормализации влияет на результат обучения и работу модели. С полученными результатами мы познакомимся в следующем разделе, посвященном тестированию моделей.

6.1.5 Сравнительное тестирование моделей с использованием пакетной нормализации

Мы с вами проделали большую работу, в результате которой создали новый класс реализации метода пакетной нормализации. Напомню его основное назначение — решение проблемы внутреннего ковариационного сдвига. В итоге модель должна быстрее обучаться, а результаты должны стать более стабильными. Давайте проведем несколько экспериментов и посмотрим, так ли это.

Вначале мы проведем тестирование модели с использованием нашего класса, написанного на языке *MQL5*. Для экспериментов мы будем использовать самые простые модели, содержащих только полносвязные слои.

В качестве первого эксперимента мы попробуем использовать слой пакетной нормализации вместо предварительной нормализации на стадии подготовки данных. Такой подход позволит снизить затраты на подготовку данных как для обучения модели, так и в процессе промышленной эксплуатации. Кроме того, включение нормализации в модель позволяет использовать ее при потоке данных в реальном времени. Именно так идут биржевые котировки, и их обработка в реальном времени дает преимущество.

Для тестирования подхода создадим скрипт, в котором будет использоваться один полносвязный скрытый слой и полносвязный слой в качестве нейронного слоя результатов. Между скрытым слоем и слоем исходных данных мы установим слой пакетной нормализации.

Задача ясна, и мы переходим к практической реализации. Для создания скрипта возьмем за базу скрипт первого тестирования полносвязного перцептрона *perceptron_test.mq5*. Создадим его копию с именем *perceptron_test_norm.mq5*.

В начале скрипта идут внешние параметры. Их перенесем в новый скрипт без изменений.

```
//+------------------------------------------------------------------+
//| Внешние параметры для работы скрипта                             |
//+------------------------------------------------------------------+
// Имя файла с обучающей выборкой
input string   StudyFileName = "study_data_not_norm.csv";
// Имя файла для записи динамики ошибки
input string   OutputFileName = "loss_study_vs_norm.csv";
// Количество исторических баров в одном паттерне
input int      BarsToLine     = 40;
// Количество нейронов входного слоя на 1 бар
input int      NeuronsToBar   = 4;
// Использовать OpenCL
input bool     UseOpenCL      = false;
// Размер пакета для обновления матрицы весов
input int      BatchSize      = 10000;
// Коэффициент обучения
input double   LearningRate   = 3e-5;
// Количество скрытых слоев
input int      HiddenLayers   =  1;
// Количество нейронов в одном скрытом слое
input int      HiddenLayer    =  40;
// Количество итераций обновления матрицы весов
input int      Epochs         =  1000;
```

В скрипте мы внесем изменения только в функцию задания архитектуры модели *CreateLayersDesc*. В параметрах данная функция получает указатель на объект динамического массива, который нам предстоит заполнить описаниями создаваемых нейронных слоев. Чтобы исключить возможные недоразумения, сразу очистим полученный динамический массив.

```
bool CreateLayersDesc(CArrayObj &layers)
  {
   layers.Clear();
   CLayerDescription *descr;
//--- создаем слой исходных данных
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
```

Первым мы создаем слой для приема исходных данных. Прежде чем передать описание создаваемого слоя, надо создать экземпляр объекта описания нейронного слоя *CLayerDescription*. Мы создаем экземпляр объекта и сразу проверяем результат выполнения операции. Обратите внимание, что в случае возникновения ошибки мы выводим сообщение пользователю, затем обязательно удаляем созданный ранее объект динамического массива и только потом завершаем выполнение программы.

При успешном создании объекта мы начинаем наполнять его нужным содержимым. Для нейронного слоя исходных данных мы указываем базовый тип полносвязного нейронного слоя с нулевым окном исходных данных без функции активации и оптимизации параметров. Количество нейронов указываем достаточным для приема всей последовательности описания паттерна. В данном случае количество равно произведению числа нейронов в описании паттерна на число элементов описания одной свечи.

```
   descr.type         = defNeuronBase;
   int prev =
      descr.count     = NeuronsToBar * BarsToLine;
   descr.window       = 0;
   descr.activation   = AF_NONE;
   descr.optimization = None;
```

Когда все параметры создаваемого нейронного слоя указаны, мы добавляем наш объект описания нейронного слоя в динамический массив описания архитектуры модели и сразу проверяем результат выполнения операции.

```
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
```

После успешного добавления описания нейронного слоя в динамический массив переходим к описанию следующего нейронного слоя. И вновь мы создаем новый экземпляр объекта описания нейронного слоя и проверяем результат выполнения операции.

```
//--- слой пакетной нормализации
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
```

Вторым слоем модели будет слой пакетной нормализации. Об этом мы скажем модели, указав соответствующую константу в поле *type*. Количество нейронов и размер окна исходных данных укажем в соответствии с размером предыдущего нейронного слоя. Для слоя пакетной нормализации добавляется новый параметр — размер пакета *batch*. Для данного параметра мы укажем значение равное размеру пакета между обновлениями параметров пакета. Функцию активации мы не используем, но указываем метод оптимизации параметров *Adam*.

```
   descr.type = defNeuronBatchNorm;
   descr.count = prev;
   descr.window = descr.count;
   descr.batch = BatchSize;
   descr.activation = AF_NONE;
   descr.optimization = Adam;
```

После указания всех необходимых параметров нового нейронного слоя мы добавляем его в динамический массив описания архитектуры модели. Как всегда, проверяем результат выполнения операции.

```
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
```

Далее в нашей модели идет блок скрытых слоев. Количество скрытых слоев указывает во внешних параметрах скрипта пользователь. Все скрытые слои представляют собой базовые полносвязные слои с одинаковым количеством нейронов, число которых указывается во внешних параметрах скрипта. Таким образом, для создания всех скрытых нейронных слоев достаточно одного объекта описания нейронного слоя, которое можно добавить несколько раз в динамический массив описания архитектуры модели.

Следовательно, следующим шагом мы создаем новый экземпляр объекта описания нейронного слоя и проверяем результат выполнения операции.

```
//--- блок скрытых слоев
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
```

После создания объекта заполним его необходимыми значениями. Укажем базовый тип нейронного слоя *defNeuronBase*. Количество элементов в нейронном слое перенесем из внешнего параметра скрипта *HiddenLayer*. В качестве функции активации будем использовать *Swish*, а метода оптимизации параметров — *Adam*.

```
   descr.type         = defNeuronBase;
   descr.count        = HiddenLayer;
   descr.activation   = AF_SWISH;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
```

Когда внесена достаточная информация для создания нейронного слоя, организуем цикл с числом итераций равным количеству скрытых слоев. В цикле будем добавлять созданное описание скрытого нейронного слоя в динамический массив описания архитектуры модели. При этом не забываем на каждой итерации контролировать процесс добавления объекта описания в массив.

```
   for(int i = 0; i < HiddenLayers; i++)
      if(!layers.Add(descr))
        {
         PrintFormat("Error adding layer: %d", GetLastError());
         delete descr;
         return false;
        }
```

В заключение описания модели остается добавить описания нейронного слоя результатов. Для этого создаем еще один экземпляр объекта описания нейронного слоя и сразу же проверяем результат выполнения операции.

```
//---  слой результатов
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
```

После создания объекта мы заполняем его описанием создаваемого нейронного слоя. Для слоя результатов используем базовый тип полносвязного нейронного слоя из двух элементов (по числу целевых значений для паттерна). Используем линейную функцию активации и метод оптимизации *Adam*, как и для других слоев модели.

```
   descr.type         = defNeuronBase;
   descr.count        = 2;
   descr.activation   = AF_LINEAR;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
```

Добавляем готовое описание нейронного слоя в динамический массив описания архитектуры модели.

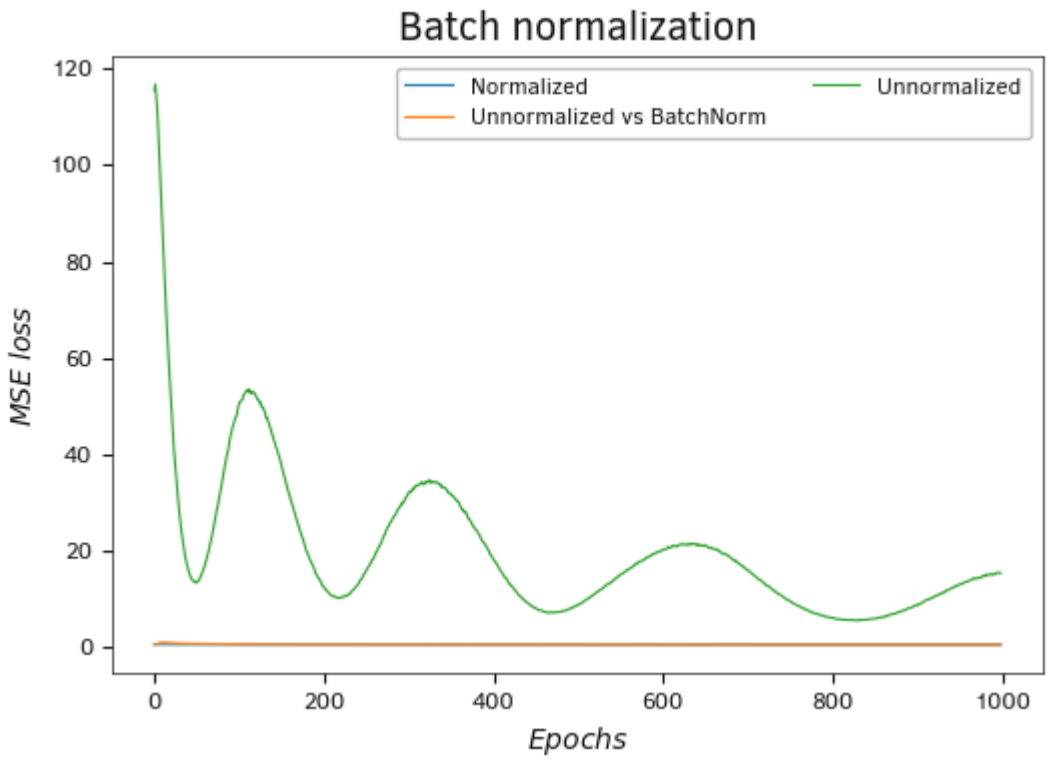
```
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
   return true;
  }
```

И конечно, не забываем проверить результат операции.

На этом мы заканчиваем работу по построению скрипта для проведения первого теста, так как остальной код скрипта перенесен без изменений. Запускать скрипт будем на ранее подготовленных данных обучающей выборки. Напомню, что для чистоты экспериментов все модели в рамках данной книги обучаются на одной обучающей выборке. Это касается как моделей созданных в среде *MQL5*, так и тех, что написаны на языке Python.

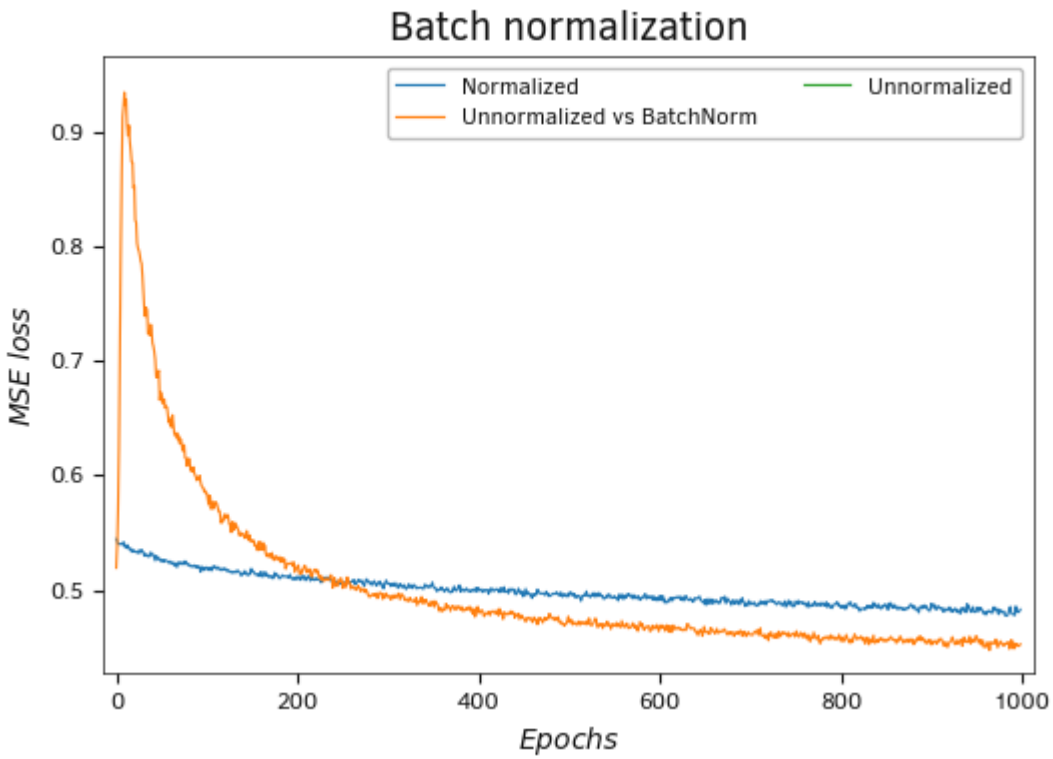
Из результатов тестирования, представленных на рисунках ниже, можно с уверенностью сказать, что использование слоя пакетной нормализации вполне может заменить процедуру предварительной нормализации на стадии подготовки исходных данных.

Непосредственно влияние слоя пакетной нормализации можно оценить, сравнив график динамики ошибки модели без слоя пакетной нормализации при обучении на ненормализованной обучающей выборке. Разрыв между графиками колоссальный.



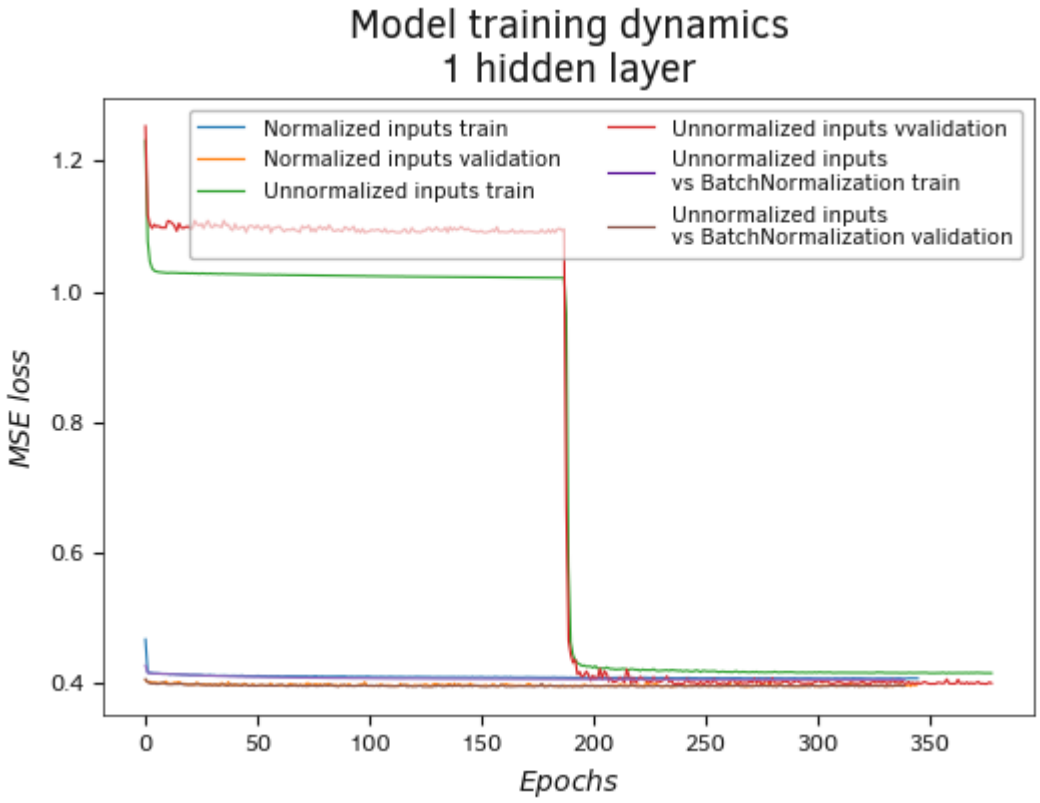
Пакетная нормализация исходных данных

В то же время различия в графиках изменения ошибки модели в процессе обучения на нормализованных данных и на ненормализованных, но с использованием слоя пакетной нормализации, можно увидеть лишь при увеличении масштаба графика. Лишь в начале обучения есть разрыв между показателями моделей. С ростом итераций обучения, разрыв в точности работы моделей резко сокращается. После 200 итераций модель с использованием слоя нормализации демонстрирует даже лучшие показатели. Это лишний раз подтверждает возможность включения слоя пакетной нормализации в модель для нормализации исходных данных в режиме реального времени.

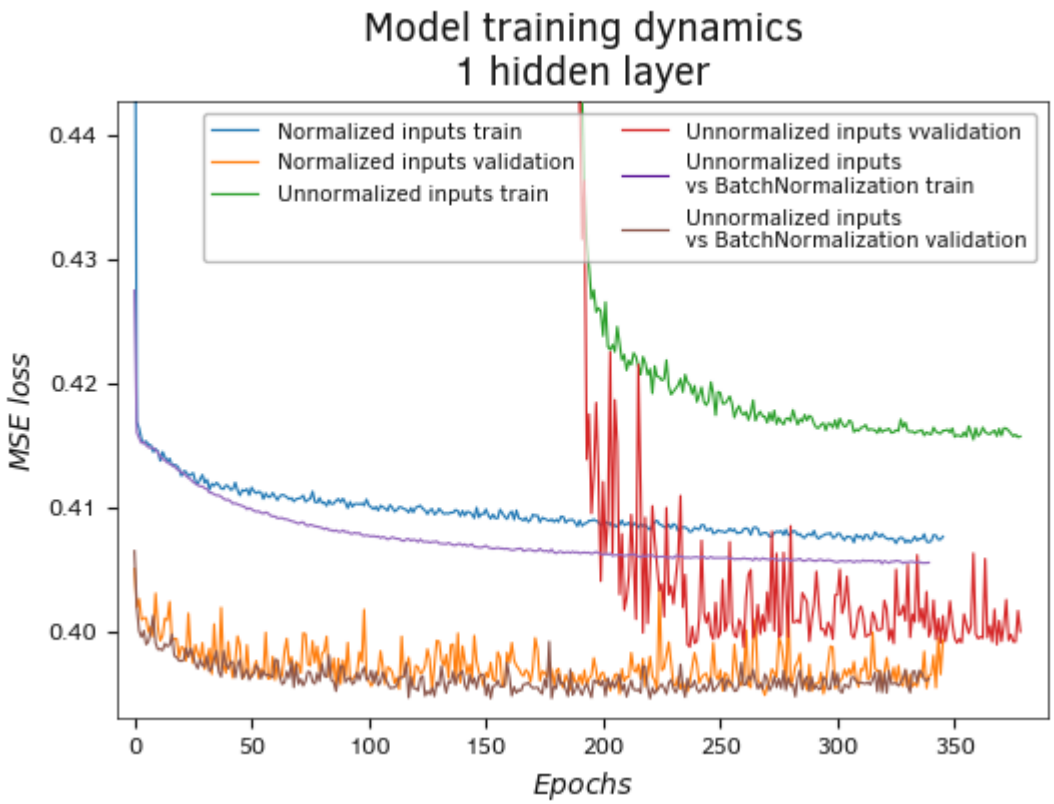


Пакетная нормализация исходных данных

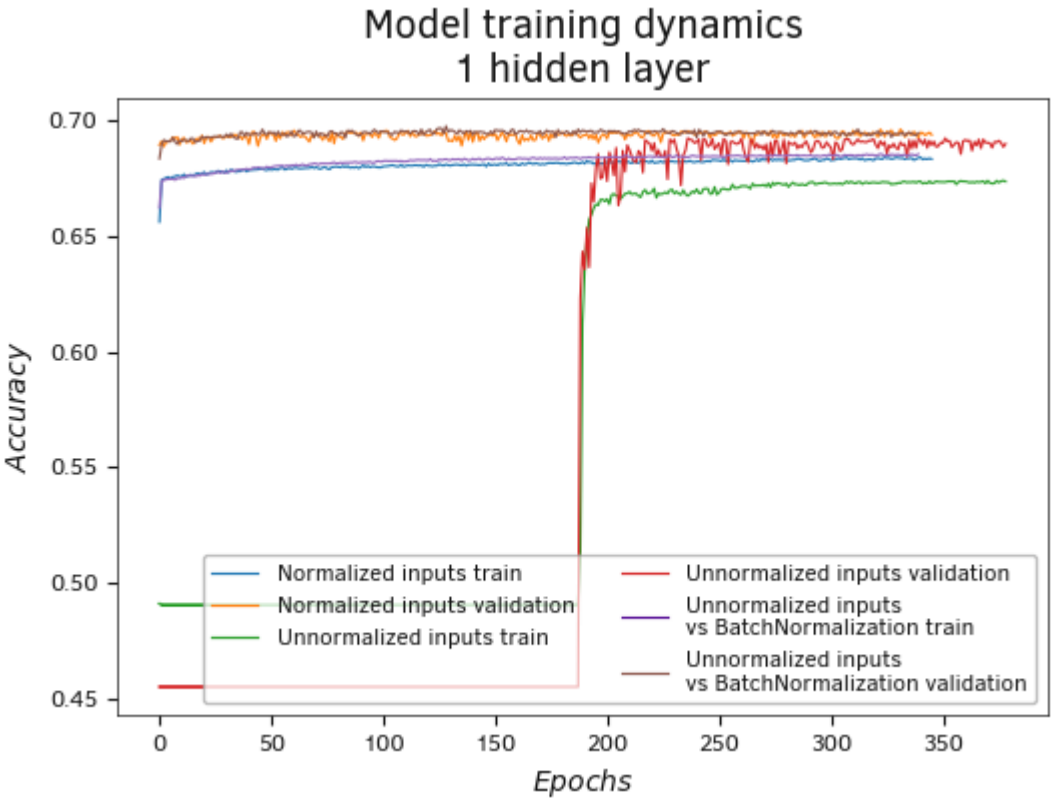
Аналогичный эксперимент мы провели и с моделями, созданными на языке *Python*. Этот эксперимент подтвердил ранее сделанные выводы.



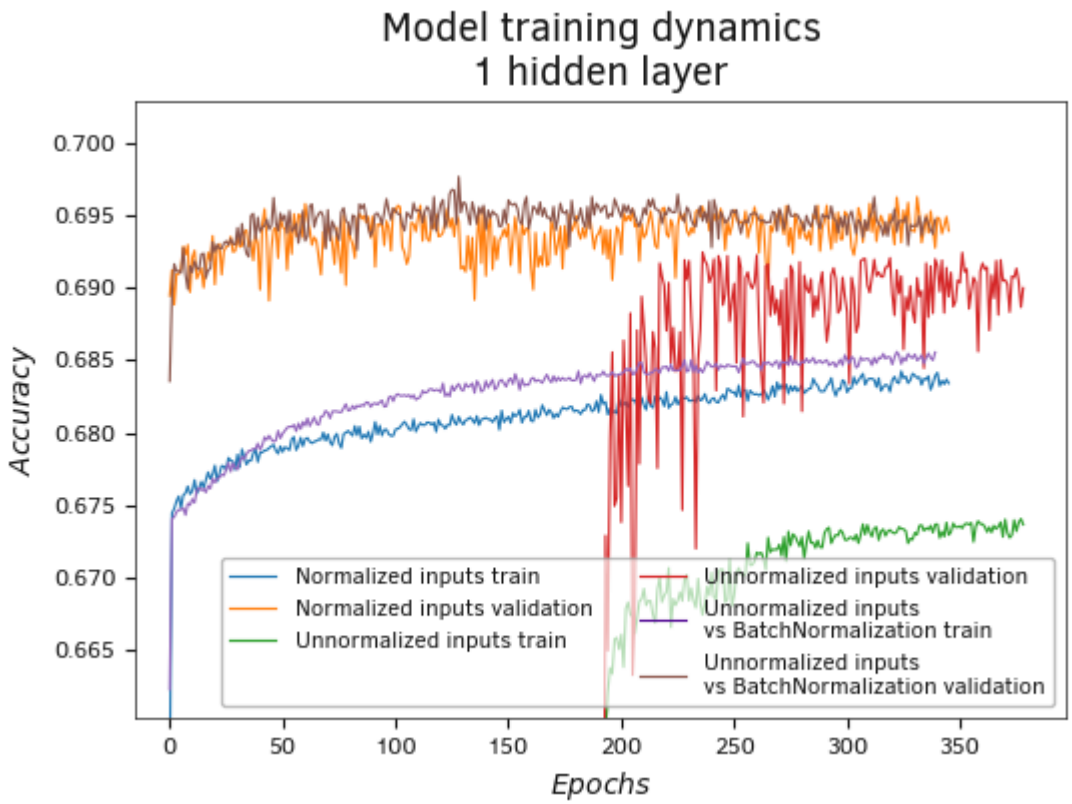
Пакетная нормализация исходных данных (MSE)



Пакетная нормализация исходных данных (MSE)



Пакетная нормализация исходных данных (Accuracy)



Пакетная нормализация исходных данных (Accuracy)

Более того, в рамках данного эксперимента модель с использованием слоя пакетной нормализации продемонстрировала немного лучше результаты как на обучающей выборке, так и на валидации.

Анализ графика метрики Accuracy позволяет сделать аналогичные выводы.

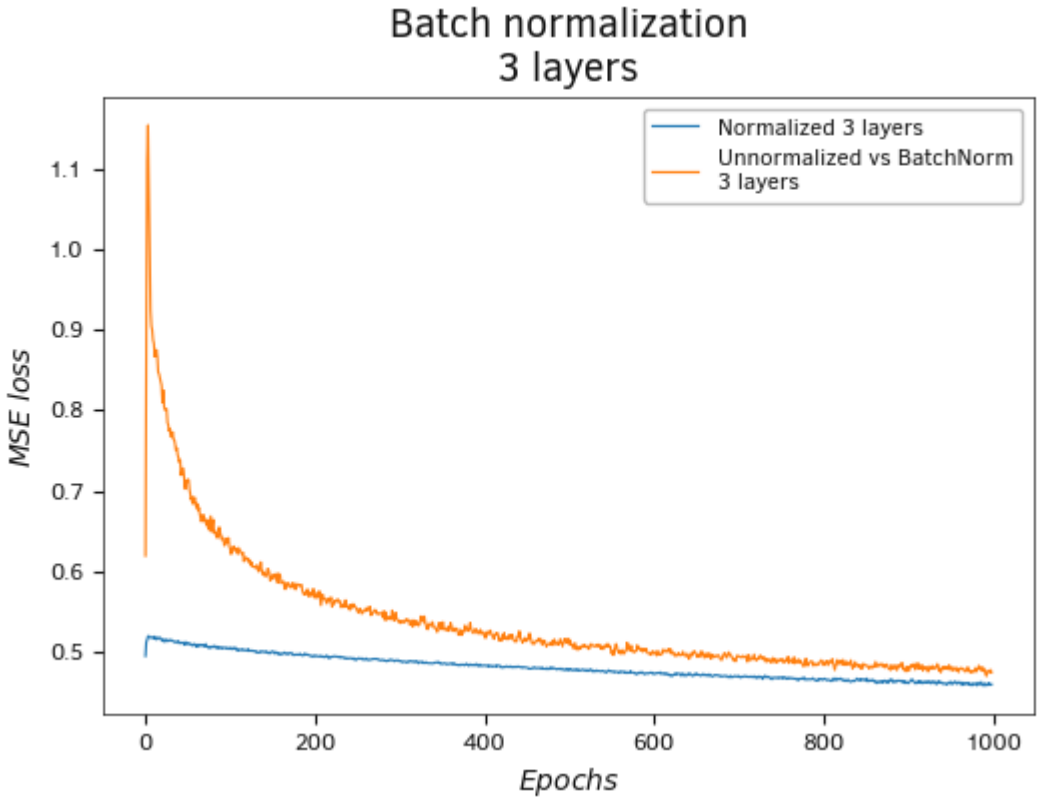
Второй и, наверное, основной вариант использования слоя пакетной нормализации — это установка слоя пакетной нормализации перед скрытыми слоями. Авторы предложили использовать этот метод именно в таком варианте для решения проблемы внутреннего ковариационного сдвига. Для проверки эффективности такого подхода создадим копию нашего скрипта с именем *perceptron_test_norm2.mq5*. Сделаем небольшие изменения в блоке создания скрытых слоев. Дело в том, что в новом скрипте нам потребуется чередовать полносвязные скрытые слои со слоями пакетной нормализации, поэтому создание слоя пакетной нормализации внесем в цикл.

```
//--- Слой пакетной нормализации
   CLayerDescription *norm = new CLayerDescription();
   if(!norm)
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
   norm.type = defNeuronBatchNorm;
   norm.count = prev;
   norm.window = descr.count;
   norm.batch = BatchSize;
   norm.activation = AF_NONE;
   norm.optimization = Adam;
//--- Скрытый слой
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      delete norm;
      return false;
     }
   descr.type         = defNeuronBase;
   descr.count        = HiddenLayer;
   descr.activation   = AF_SWISH;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
   for(int i = 0; i < HiddenLayers; i++)
     {
      if(!layers.Add(norm))
        {
         PrintFormat("Error adding layer: %d", GetLastError());
         delete descr;
         delete norm;
         return false;
        }
      CLayerDescription *temp = new CLayerDescription();
      if(!temp)
        {
         PrintFormat("Error creating CLayerDescription: %d", GetLastError());
         delete descr;
         return false;
        }
      temp.Copy(norm);
      norm = temp;
      norm.count = descr.count;
      if(!layers.Add(descr))
        {
         PrintFormat("Error adding layer: %d", GetLastError());
         delete descr;
         delete norm;
         return false;
        }
```

```
    }
delete norm;
```

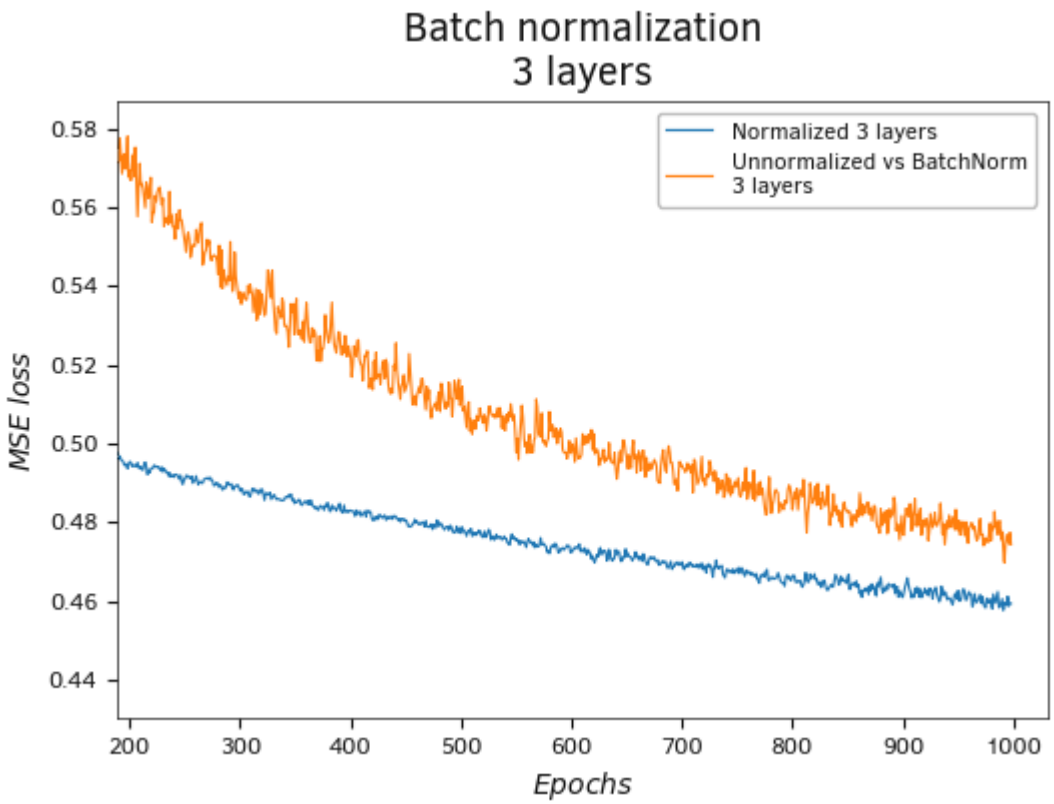
В остальном скрипт остался без изменений.

Тестирование работы скрипта полностью подтвердило ранее сделанные выводы. На начальном этапе при обучении на ненормированных данных модели с пакетной нормализацией требуется немного времени на адаптацию. Но разрыв в точности работы моделей резко сокращается.

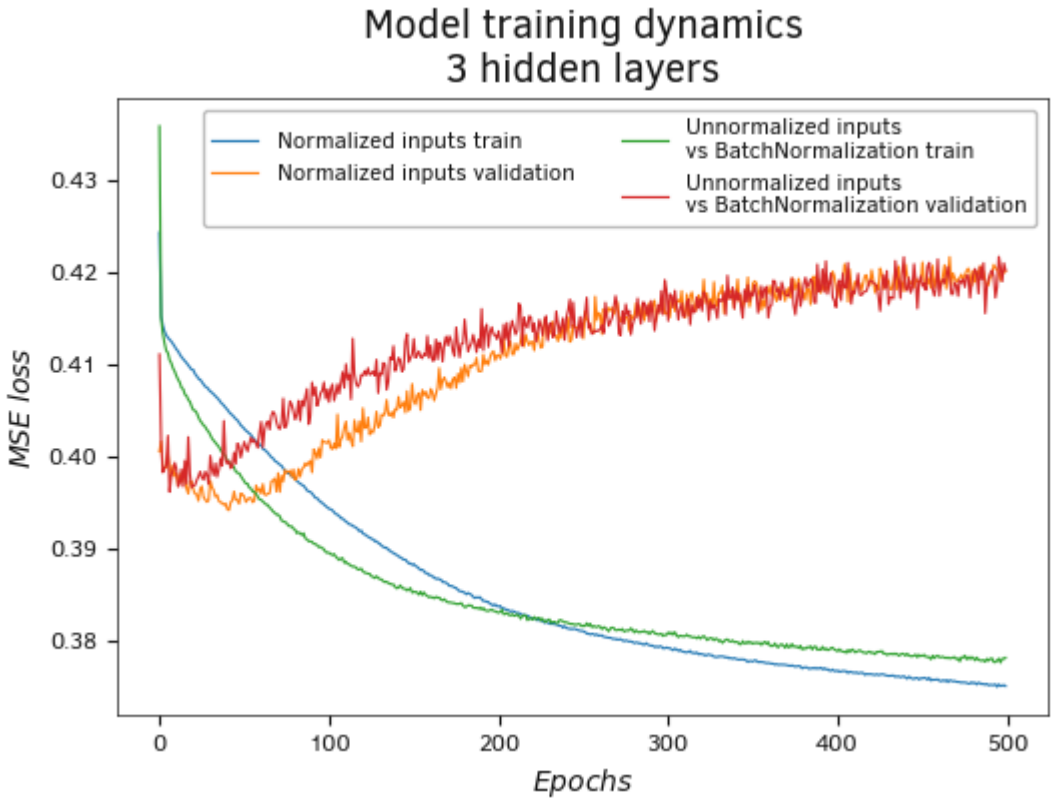


Пакетная нормализация перед скрытым слоем

При увеличении масштаба четко видно, что модель с тремя скрытыми слоями и слоем пакетной нормализации перед каждым скрытым слоем даже на ненормализованных исходных данных демонстрирует лучшие результаты. При этом ее график динамики ошибки снижается более быстрыми темпами по сравнению с остальными моделями.



Пакетная нормализация перед скрытым слоем



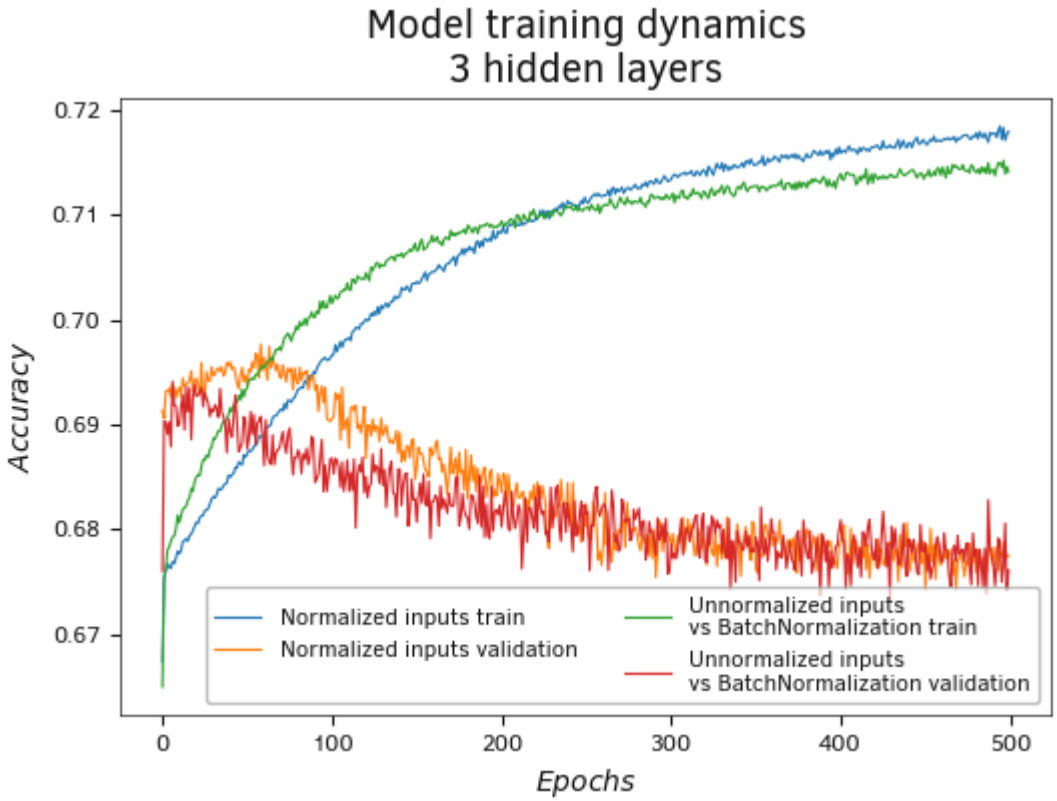
Пакетная нормализация перед скрытым слоем (MSE)

Проведение аналогично эксперимента с моделями, созданными на Python, также подтверждает, что модели с использованием слоя пакетной нормализации перед каждым скрытым слоем при прочих равных условиях обучаются быстрее и менее склонны к переобучению.

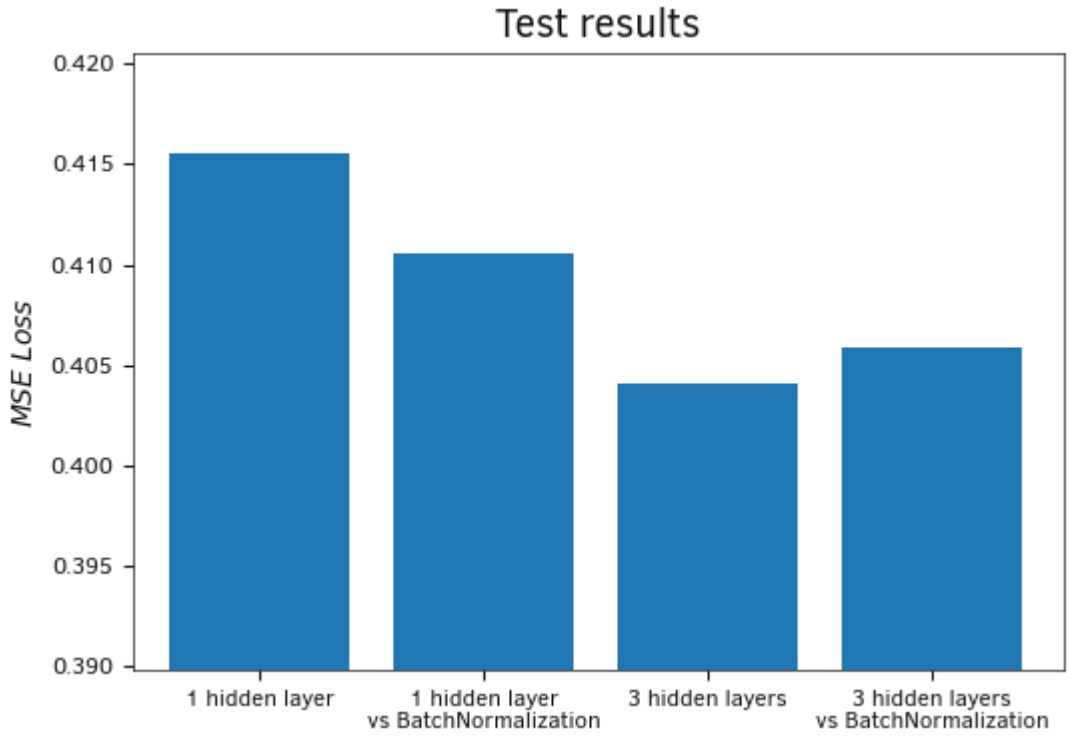
Динамика изменения значения метрики *Accuracy* также подтверждает ранее сделанные выводы.

Дополнительно мы проверили модели на тестовой выборке, чтобы оценить эффективность работы на новых данных. Полученные результаты продемонстрировали довольно ровную работу всех четырех моделей. Расхождение среднеквадратичной ошибки моделей не превысило $5*10^{-3}$. Лишь небольшое преимущество продемонстрировали модели с тремя скрытыми слоями.

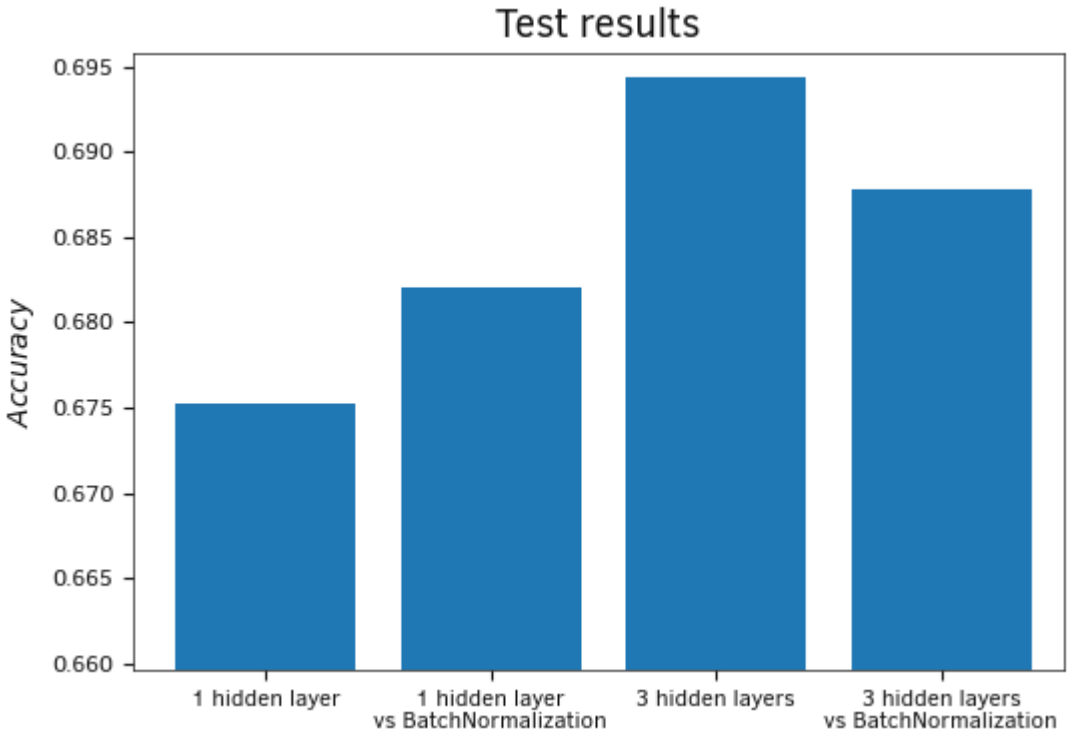
Оценка моделей по метрике *Accuracy* показало схожие результаты.



Пакетная нормализация перед скрытым слоем (Accuracy)



Проверка эффективности пакетной нормализации на новых данных

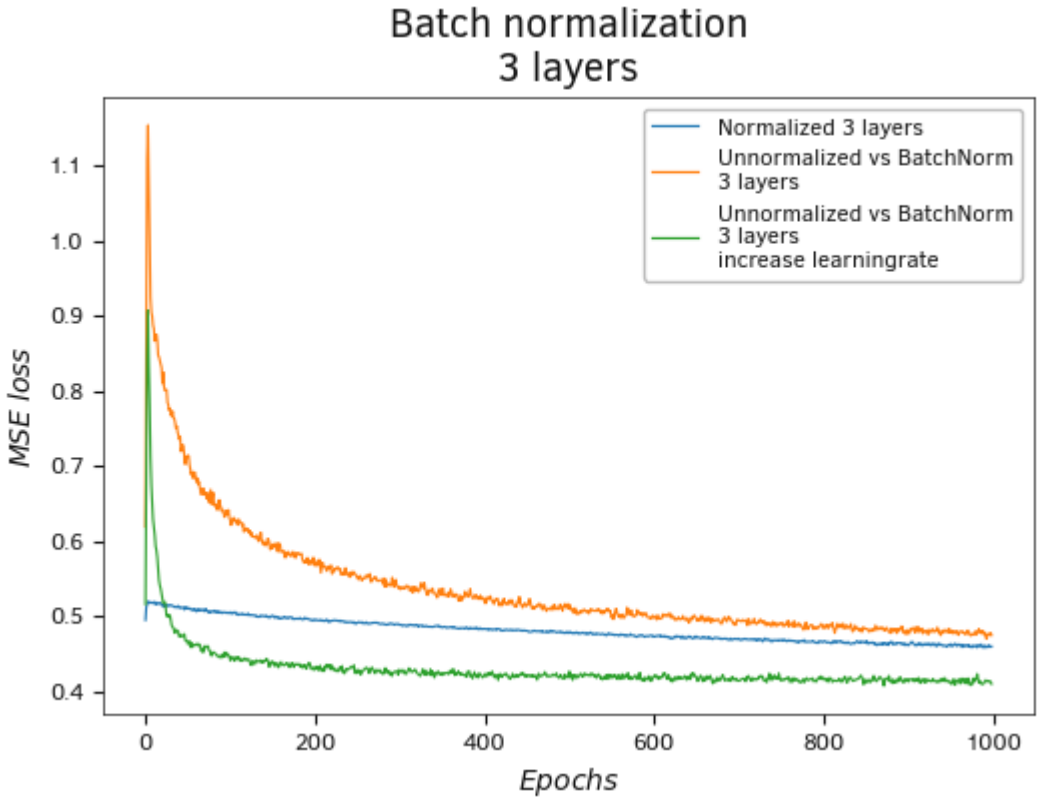


Проверка эффективности пакетной нормализации на новых данных

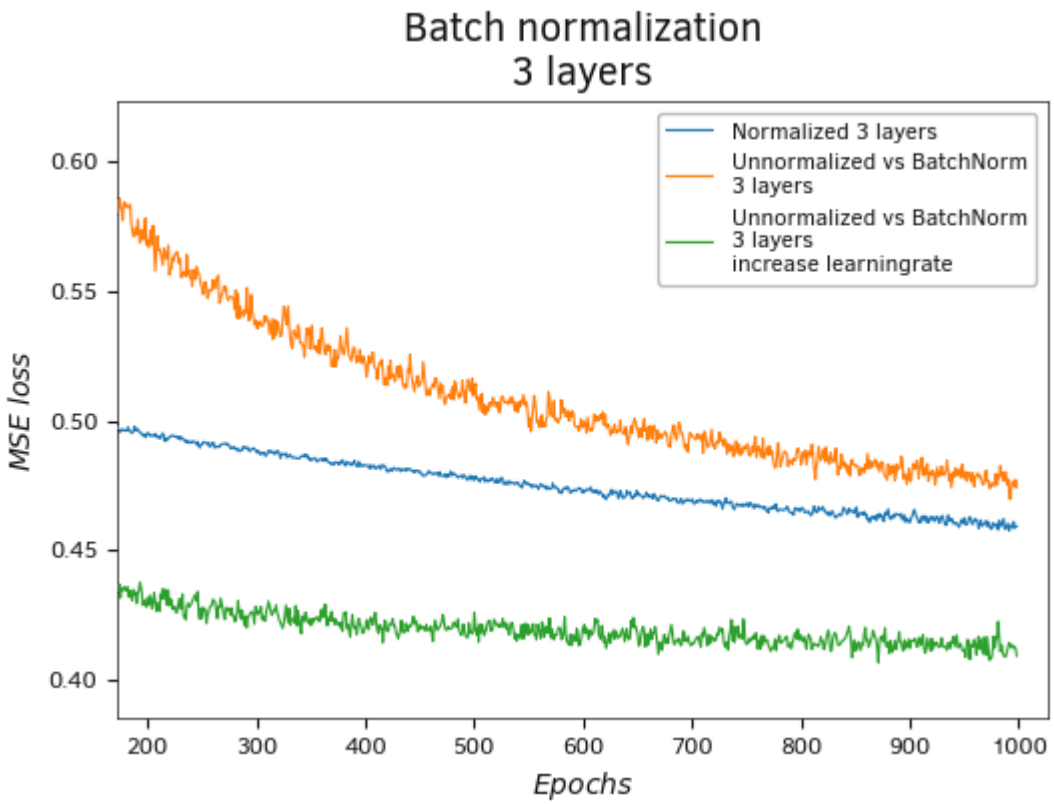
В заключение я решил провести еще один тест. Авторы метода утверждают, что использование слоя пакетной нормализации позволяет увеличить коэффициент обучения для ускорения

процесса. Проверим это утверждение. Повторно запустим скрипт *perceptron_test_norm2.mq5*, но на этот раз увеличим коэффициент обучения в 10 раз.

Тестирование показало перспективность такого подхода. Помимо более быстрого процесса обучения, мы получили результат обучения лучше предыдущих.



Пакетная нормализация перед скрытым слоем с увеличенным коэффициентом обучения



Пакетная нормализация перед скрытым слоем с увеличенным коэффициентом обучения

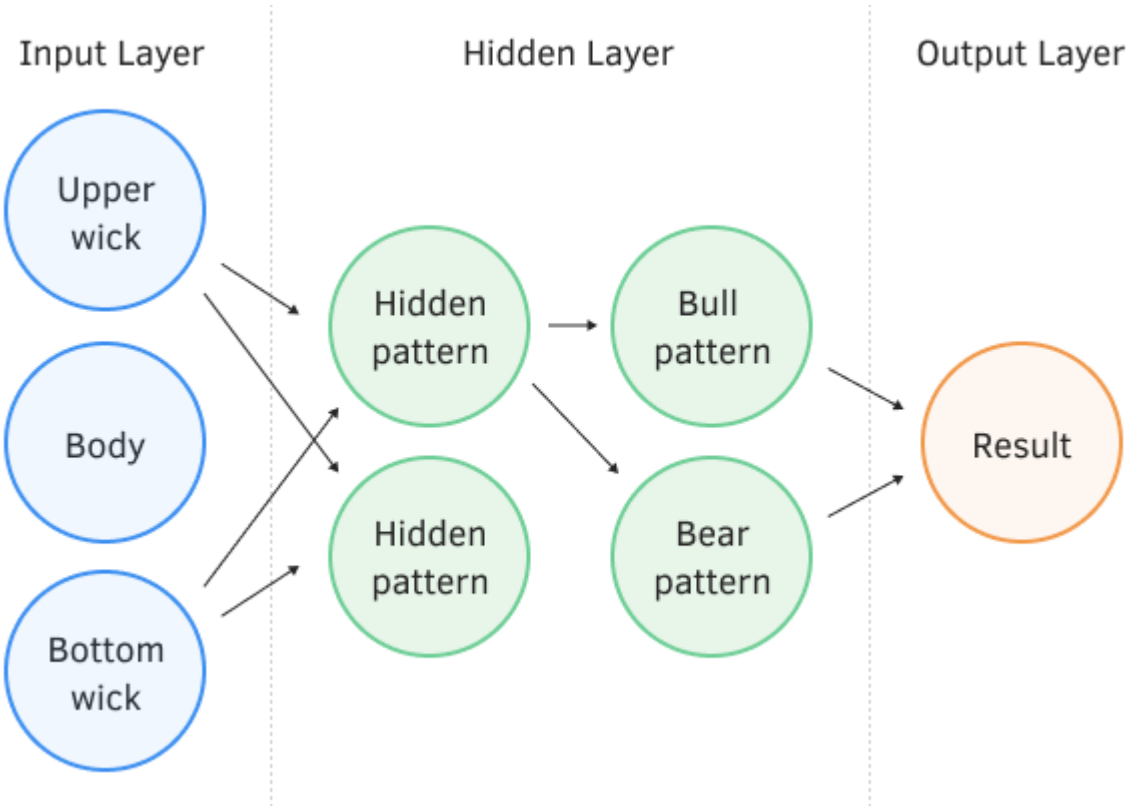
В данном разделе мы провели ряд тестовых обучений различных моделей с использованием слоя пакетной нормализации и без. Полученные результаты продемонстрировали, что слой пакетной нормализации после исходных данных может заменить процесс нормализации на стадии подготовки исходных данных для обучения. Такой подход позволяет встроить процесс нормализации данных в модель и настраивать его в процессе обучения модели. Таким образом, в процессе эксплуатации модели мы можем обрабатывать исходные данные в режиме реального времени без усложнения общей программы принятия решений.

Кроме того, использование слоя пакетной нормализации перед скрытыми слоями модели позволяет ускорить процесс обучения при прочих равных условиях.

6.2 Dropout

В продолжение разговора о повышении сходимости моделей предлагаю рассмотреть метод *Dropout*.

При обучении нейронной сети на вход каждого нейрона подается большое количество признаков, и сложно оценить влияние каждого из них. Как результат, ошибки одних нейронов сглаживаются правильными значениями других, а на выходе нейронной сети ошибки накапливаются. Как результат, обучение останавливается в некоем локальном минимуме с достаточно большой ошибкой. Данный эффект был назван совместной адаптацией признаков, когда влияние каждого признака как бы подстраивается под окружающую среду. Для нас было бы лучше получить обратный эффект, когда среда будет разложена по отдельным признакам, и оценивать отдельно влияние каждого.



Dropout

Для борьбы со сложной совместной адаптацией признаков в июле 2012 года группа ученных из университета Торонто в статье *Improving neural networks by preventing co-adaptation of feature detectors* предложила случайным образом исключать часть нейронов в процессе обучения. Снижение количества признаков при обучении повышает значимость каждого, а постоянное изменение количественного и качественного состава признаков снижает риск их совместной адаптации. Такой метод получил название *Dropout*. Некоторые сравнивают применение данного метода с деревьями решений, ведь согласитесь, исключая часть нейронов, мы на каждой итерации обучения получаем новую нейронную сеть со своими весовыми коэффициентами. По правилам комбинаторики вариативность таких сетей довольно высока.

В процессе эксплуатации нейронной сети оцениваются все признаки и нейроны. Тем самым мы получаем максимально точную и независимую оценку текущего состояния изучаемой среды.

Авторы метода в своей статье указывают на возможность его использования и для повышения качества предварительно обученных моделей.

Описывая предложенное решение с точки зрения математики, можно сказать, что каждый отдельный нейрон выкидывается из процесса с некой заданной вероятностью p, или нейрон будет участвовать в процессе обучения нейронной сети с вероятностью q.

$$q = 1 - p$$

Для определения списка исключаемых нейронов используется генератор псевдослучайных чисел с нормальным распределением. Такой подход позволяет достичь максимально возможного равномерного исключения нейронов. На практике мы сгенерируем вектор размером равным входной последовательности. Для используемых признаков в векторе пропишем 1, а для исключаемых элементов поставим 0.

Однако исключение анализируемых признаков несомненно ведет к снижению суммы на входе функции активация нейрона. Для компенсации этого эффекта умножим значение каждого признака на коэффициент *1/q*. Легко заметить, что данный коэффициент будет увеличивать значения, так как вероятность *q* всегда будет в диапазоне от 0 до 1.

$$d_i = \frac{1}{q} x_i n_i$$

где:

- d_i — элементы вектора результатов *Dropout*;
- q — вероятность использования нейрона в процессе обучения;
- x_i — элементы вектора маскирования;
- n_i — элементы входной последовательности.

При обратном проходе в процессе обучения градиент ошибки умножается на производную вышеприведенной функции. В случае *Dropout* обратный проход будет аналогичен прямому с использованием вектора маскирования из прямого прохода.

$$\left(\frac{1}{q} x_i n_i\right)' = \frac{1}{q} x_i$$

В процессе эксплуатации нейронной сети вектор маскирования заполняется единицами, что позволяет беспрепятственно передавать значения в обоих направлениях.

На практике коэффициент *1/q* постоянен на протяжении всего обучения, поэтому мы легко можем посчитать данный коэффициент один раз и записывать его вместо единицы в тензор маскирования. Тем самым мы объединим операции пересчета коэффициента и умножения маски на 1 в каждой итерации обучения.

6.2.1 Реализация Dropout средствами MQL5

После рассмотрения теоретических аспектов предлагаю перейти к рассмотрению варианта реализации данного метода в нашей библиотеке.

Для реализации алгоритма *Dropout* создадим новый класс *CNeuronDropout*, который мы будем включать в нашу модель отдельным слоем. Новый класс будет наследоваться напрямую от базового класса нейронного слоя *CNeuronBase*.

```
class CNeuronDropout    :  public CNeuronBase
  {
protected:
   TYPE              m_dOutProbability;
   int               m_iOutNumber;
   TYPE              m_dInitValue;

   CBufferType       m_cDropOutMultiplier;

public:
                     CNeuronDropout(void);
                    ~CNeuronDropout(void);
   //---
   virtual bool      Init(const CLayerDescription *desc) override;
   virtual bool      FeedForward(CNeuronBase *prevLayer) override;
   virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer, bool read)
                                                        override { return true; }
   virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                       VECTOR &Beta, VECTOR &Lambda) override { return true; }
   //--- методы работы с файлами
   virtual bool      Save(const int file_handle) override;
   virtual bool      Load(const int file_handle) override;
   //--- метод идентификации объекта
   virtual int       Type(void) override    const { return defNeuronDropout; }
  };
```

И первое, с чем мы сталкиваемся, — это реализация двух различных алгоритмов: одного для процесса обучения, а второго для тестирования и промышленной эксплуатации. Соответственно, нам нужно явно указать нейронному слою, по какому алгоритму нужно работать в каждом отдельно взятом случае. Для этого введем флаг *m_bTrain*, которому будем присваивать значение *true* в процессе обучения и *false* в процессе тестирования.

Для управления значениями флага создадим вспомогательный перегруженный метод *TrainMode*. В одном варианте, при указании параметра, он будет устанавливать флаг, а во втором, при вызове без параметров, он вернет текущее значение флага *m_bTrain*.

```
   virtual void      TrainMode(bool flag)       {  m_bTrain = flag; }
   virtual bool      TrainMode(void)      const {  return m_bTrain; }
```

В процессе всей работы с библиотекой мы выстаивали механизм переопределения методов всех классов. Тем самым мы создавали универсальную архитектуру классов, позволяющую диспетчерскому классу нашей модели работать одинаково с любым нейронным слоем, не тратя время на проверку типа нейронного слоя и разветвления алгоритмов в зависимости от используемого нейронного слоя. Для поддержания этой концепции мы вводим переменную флага и методы работы с ним на уровне базового нейронного слоя *CNeuronBase*.

В блоке *protected* нашего класса мы объявим следующие переменные:

- *m_dOutProbability* — заданная вероятность исключения нейронов;
- *m_iOutNumber* — количество исключаемых нейронов;
- *m_dInitValue* — значение для инициализации вектора маскирования, в теоретической части этой статьи мы обозначали данный коэффициент как *1/q*.

Также объявим указатель на объект буфера данных для вектора маскирования *m_cDropOutMultiplier*.

Список методов класса довольно узнаваем, и все они переопределяют методы родительского класса.

Надо сказать, что наш новый слой не имеет матриц весовых коэффициентов. Переопределение методов *CalcDeltaWeights* и *UpdateWeights*, отвечающих за распределение градиента ошибки до матрицы весовых коэффициентов и обновление параметров модели, создано для поддержания общей архитектуры работы нейронных слоев и модели в целом. Использовать методы родительского класса мы не можем, так как отсутствие соответствующих объектов приведет к возникновению критической ошибки. А создание дополнительных неиспользуемых объектов — это нерациональная трата ресурсов. Поэтому мы переопределяем методы. Но при этом создаем их пустыми — они просто всегда будут возвращать положительное значение.

```
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer, bool read)
                                                        override { return true; }
   virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                       VECTOR &Beta, VECTOR &Lambda) override { return true; }
```

После короткого рассмотрения структуры класса предлагаю перейти к работе над его методами. Начнем, как всегда, с конструктора класса. В данном методе укажем значение переменных по умолчанию. Использование статического объекта буфера вектора маскирования позволяет пропустить операцию его создания в конструкторе и удаления в деструкторе.

```
CNeuronDropout::CNeuronDropout(void)    :  m_dInitValue(1.0),
                                           m_dOutProbability(0),
                                           m_iOutNumber(0)
  {
   m_bTrain = false;
  }
```

Обратите внимание, что значения флага режима работы класса *m_bTrain*, в отличие от других переменных, указывается в теле метода. Это связано с объявлением переменной в родительском классе.

Деструктор метода остается пустым.

Далее идет метод инициализации класса *CNeuronDropout::Init*. В параметрах метод получает указатель на объект класса описания создаваемого нейронного слоя. В теле метода мы сразу проверяем действительность полученного указателя, а также соответствие размеров создаваемого нейронного слоя и предыдущего. Напомню, что единственная роль слоя *Dropout* это маскирования нейронов, при этом размеры тензора никак не изменяются.

```
bool CNeuronDropout::Init(const CLayerDescription *description)
  {
//--- блок контролей
   if(!description || description.count != description.window)
      return false;
```

После успешного прохождения блока контролей мы обнуляем размер окна исходных данных и вызываем метод инициализации родительского класса. Обнуление размера окна исходных данных укажет методу родительского класса не создавать матрицу весовых коэффициентов и

другие объекты, связанные с обучением параметров нейронного слоя. Как всегда, не забываем проверить результат выполнения операций.

```
//--- вызов метода родительского класса
   CLayerDescription *temp = new CLayerDescription();
   if(!temp || !temp.Copy(description))
      return false;
   temp.window = 0;
   if(!CNeuronBase::Init(temp))
      return false;
   delete temp;
```

После успешного выполнения метода родительского класса мы сохраняем основные параметры работы нейронного слоя, включая вероятность исключения нейронов, количество исключаемых нейронов и значение инициализации матрицы маскирования. Первый параметр мы получаем от пользователя, а два других находим расчетным путем.

```
//--- расчет коэффициентов
   m_dOutProbability = (TYPE)MathMin(description.probability, 0.9);
   if(m_dOutProbability < 0)
      return false;
   m_iOutNumber = (int)(m_cOutputs.Total() * m_dOutProbability);
   m_dInitValue = (TYPE)(1.0 / (1.0 - m_dOutProbability));
```

После этого инициализируем начальными значениями буфер маскирования и устанавливаем флаг обучения нейронного слоя в положение *true*.

```
//--- инициируем буфер маскирования
   if(!m_cDropOutMultiplier.BufferInit(m_cOutputs.Rows(), m_cOutputs.Cols(),
                                                                m_dInitValue))
      return false;
   m_bTrain = true;
//---
   return true;
  }
```

На этом мы завершаем работу с методами инициализации класса и переходим к непосредственному созданию алгоритма метода *Dropout*.

Но прежде давайте вспомним, что у нас нет доступа к нейронному слою напрямую из основной программы. Сейчас мы ввели флаг режима работы нейронного слоя. Следовательно, нужно вернуться к диспетчерскому классу модели и добавить в него метод изменения состояния флага.

```
void CNet::TrainMode(bool mode)
  {
   m_bTrainMode = mode;
   int total = m_cLayers.Total();
   for(int i = 0; i < total; i++)
     {
      if(!m_cLayers.At(i))
         continue;
      CNeuronBase *temp = m_cLayers.At(i);
      temp.TrainMode(mode);
     }
  }
```

В данном методе мы сохраним значение флага в локальную переменную и в цикле переберем все нейронные слои модели с вызовом аналогичного метода для каждого нейронного слоя модели.

6.2.1.1 Метод прямого прохода Dropout

Прямой проход традиционно организован в методе *FeedForward*. Напомню, что данный метод объявлен виртуальным в базовом классе нейронного слоя. Он переопределяется в каждом новом классе для выстраивания конкретного алгоритма работы класса. Данный класс не будет исключением — в нем также переопределим данный метод.

В параметрах метод *CNeuronDropout::FeedForward* получает указатель на объект предыдущего слоя нашей модели. В теле метода мы сразу организуем блок контролей для проверки указателей на объекты, используемые в данном методе. Как обычно, здесь мы проверяем не только указатели на внешние объекты, полученные в параметрах, но и на внутренние объекты класса. В данном случае мы проверим указатели на объект предыдущего слоя и его буфер результатов. Также проверим действительность указателя на буфер результатов текущего слоя.

```
bool CNeuronDropout::FeedForward(CNeuronBase *prevLayer)
  {
//--- блок контролей
   if(!prevLayer || !prevLayer.GetOutputs() || !m_cOutputs)
      return false;
```

После успешного прохождения блока контролей мы переходим к выполнению алгоритма метода *Dropout*.

Для выполнения алгоритма в режиме обучения, подготовим буфер маскирования. Вначале заполняем весь буфер повышающими коэффициентами *1/q*, который мы сохранили в переменную *m_dInitValue* на стадии инициализации класса.

После этого организуем цикл с числом итераций равным количеству исключаемых элементов. В теле цикла будем генерировать случайные значения из диапазона от 0 до количества элементов последовательности. Для случайно выбранных элементов мы будем заменять множитель в буфере маскирования на 0.

Хотя снаряд не падает дважды в одну воронку, мы все же предусмотрим алгоритм действий на случай выпадения одного элемента дважды. Перед записью 0 в буфер маскирования сначала проверим текущий коэффициент для выпавшего элемента. Если он равен нулю, то мы уменьшаем значение счетчика итераций цикла и переходим к выбору следующего элемента. Такой подход позволит исключить именно заданное количество элементов.

```
//--- генерируем тензор маскирования данных
   ulong total = m_cOutputs.Total();
   if(!m_cDropOutMultiplier.m_mMatrix.Fill(m_dInitValue))
      return false;
   for(int i = 0; i < m_iOutNumber; i++)
     {
      int pos = (int)(MathRand() * MathRand() / MathPow(32767.0, 2) * total);
      if(m_cDropOutMultiplier.m_mMatrix.Flat(pos) == 0)
        {
         i--;
         continue;
        }
      if(!m_cDropOutMultiplier.m_mMatrix.Flat(pos, 0))
         return false;
     }
```

После генерации вектора маскирования нам остается его применить к исходным данным. Для этого достаточно поэлементно умножить два буфера: исходных данных и маскирования.

Здесь надо вспомнить, что согласно нашей концепции построения библиотеки, в каждом методе классе по возможности мы создаем две ветки выполнения алгоритма: стандартными средствами *MQL5* и с использованием технологии многопоточных вычислений средствами *OpenCL*. Поэтому далее мы создаем разветвление алгоритма в зависимости от выбранного устройства вычислительных операций.

Как всегда, сейчас мы рассмотрим реализацию алгоритма средствами *MQL5*. К реализации алгоритма в режиме многопоточных операций с использованием технологии *OpenCL* мы вернемся немного позже. В блоке реализации алгоритма средствами *MQL5* воспользуемся матричными операциями.

```
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
```

Как вы помните, метод имеет два режима работы: в процессе обучения и эксплуатации. Поэтому перед выполнением алгоритма мы проверяем текущий режим работы. Если класс работает в режиме эксплуатации, мы просто копируем содержимое буфера результатов предыдущего слоя в буфер результатов текущего слоя. В случае процесса обучения умножаем тензор исходных данных на тензор маскирования.

```
//--- проверка флага режима работы
      if(!m_bTrain)
         m_cOutputs.m_mMatrix = prevLayer.GetOutputs().m_mMatrix;
      else
         m_cOutputs.m_mMatrix = prevLayer.GetOutputs().m_mMatrix *
                                m_cDropOutMultiplier.m_mMatrix;
     }
   else  // Блок OpenCL
     {
      return false;
     }
//---
   return true;
  }
```

И так, в результате операций указанных выше в буфере результатов нашего слоя содержатся замаскированные данные предыдущего слоя. Задача, поставленная перед методом прямого прохода, выполнена, и мы можем завершить работу метода. Предварительно поставим временную заглушку вместо алгоритма многопоточных вычислений.

Далее переходим к организации процесса обратного прохода.

6.2.1.2 Методы обратного прохода Dropout

Традиционно после реализации алгоритма прямого прохода мы переходим к организации процесса обратного прохода. Как вы знаете, в базовом классе нейронного слоя алгоритм обратного прохода реализован четырьмя виртуальными методами:

- метод расчета градиента ошибки на выходе нейронной сети ***CalcOutputGradient***;
- метод распространения градиента через скрытый слой ***CalcHiddenGradient***;
- метод необходимого расчета корректирующих значений для весовых коэффициентов ***CalcDeltaWeights***;
- метод обновления матрицы весовых коэффициентов ***UpdateWeights***.

Все вышеуказанные методы переопределяются в новых классах по мере необходимости. Как уже было сказано ранее, наш слой *Dropout* не содержит обучаемых параметров. Как следствие, он не содержит матрицы весовых коэффициентов. Таким образом, последние два метода не актуальны для нашего класса. В то же время нам придется их переопределить для поддержания целостности архитектуры нашей модели, ведь в процессе обучения она будет вызывать указанные методы для всех используемых нейронных слоев. И если мы их не переопределим, то при вызове указанных методов будут осуществляться операции унаследованного родительского метода. При этом отсутствие буфера матрицы весовых коэффициентов и сопутствующих объектов может привести к критическим ошибкам. В лучшем случае, в результате работы наших контролей мы завершим работу метода с ложным результатом, а это приведет к остановке процесса обучения. Поэтому мы переопределим данные методы и заменим их пустыми методами, которые всегда будут возвращать положительный результат.

```
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer, bool read)
                                                        override { return true; }
   virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                          VECTOR &Beta, VECTOR &Lambda) override { return true; }
```

Первый метод ***CalcOutputGradient*** используется только для слоя результатов. Принцип работы *Dropout* не предполагает его использование в качестве слоя результатов. Следовательно, его мы не будем переопределять.

Таким образом нам осталось переопределить только один метод — метод распространения градиента через скрытый слой ***CalcHiddenGradient***. Данный метод, как и большинство предыдущих, объявлен виртуальным в базовом классе нейронной сети переопределяется во всех новых классах для выстраивания конкретного алгоритма работы нейронного слоя. В параметрах метод получает указатель на объект предыдущего слоя. Сразу в теле метода организуем блок контролей для проверки действительности указателей на используемые методом объекты. Как и в методе прямого прохода, мы проверяем указатели на все используемые объекты, и внешние, и внутренние.

```
bool CNeuronDropout::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
//--- блок контролей
   if(!prevLayer || !prevLayer.GetGradients() || !m_cGradients)
      return false;
```

После успешного прохождения блока контролей мы должны создать разветвление алгоритма в зависимости от вычислительного устройства. И, как всегда, в этом разделе мы рассмотрим реализацию алгоритма средствами *MQL5*, а к многопоточной реализации алгоритма вернемся в следующем разделе.

```
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   ulong total = m_cOutputs.Total();
   if(!m_cOpenCL)
     {
```

В блоке реализации средствами *MQL5* мы проверяем режим работы класса. В случае работы в режиме промышленной эксплуатации мы просто копируем данные из буфера градиентов ошибки текущего слоя в аналогичный буфер предыдущего слоя.

```
      //--- проверка флага режима работы
      if(!m_bTrain)
         prevLayer.GetGradients().m_mMatrix = m_cGradients.m_mMatrix;
      else
         prevLayer.GetGradients().m_mMatrix = m_cGradients.m_mMatrix *
                                              m_cDropOutMultiplier.m_mMatrix;
     }
   else  // блок OpenCL
     {
      return false;
     }
//---
   return true;
  }
```

В случае работы метода в режиме обучения модели, согласно алгоритму работы *Dropout,* нам необходимо поэлементно умножить буфер градиентов ошибки текущего слоя на буфер вектора маскирования. Матричная операция умножения нам позволяет сделать это буквально в одну строку кода.

Как видите, на данном этапе мы передали градиент ошибки в буфер предыдущего слоя. Следовательно, задача, поставленная перед данным методом, решена, и мы можем завершить работу метода. Предварительно установим заглушку в блоке организации многопоточных операций. К ней мы вернемся в одном из последующих разделов.

Таким образом, мы полностью реализовали алгоритм работы *Dropout* стандартными средствами *MQL5*. Уже сейчас можно создать модель и получить первые результаты работы подхода. Но мы уже не раз говорили, что для полноценной работы любого нейронного слоя в рамках модели не менее важно иметь возможность восстановить работоспособность обученной ранее модели в любое удобное для пользователя время. Поэтому в следующем разделе мы рассмотрим методы сохранения данных нейронного слоя и восстановления функционирования слоя из ранее сохраненных данных.

6.2.1.3 Методы работы с файлами

В предыдущих разделах мы уже познакомились с алгоритмом работы подхода *Dropout* и даже успели создать класс *CNeuronDropout* для его реализации в рамках нашей библиотеки. В рамках указанного класса мы реализовали алгоритм работы прямого и обратного проходов *Dropout.* Теперь, для полноценной реализации данного класса, нам необходимо добавить методы работы с файлами, которые позволят сохранить и восстановить работу обученной ранее модели в любое удобное время. Это дает возможность восстановить работоспособность модели в кратчайшие сроки.

И как всегда, начиная подобную работу, мы критически оцениваем переменные и объекты нашего класса чтобы решить, сохранить их в файл полностью или частично, или же восстановить их по каким-то параметрам.

```
class CNeuronDropout    :  public CNeuronBase
  {
protected:
   TYPE              m_dOutProbability;
   int               m_iOutNumber;
   TYPE              m_dInitValue;
   CBufferType       m_cDropOutMultiplier;

public:
                     CNeuronDropout(void);
                    ~CNeuronDropout(void);
   //---
   virtual bool      Init(const CLayerDescription *desc) override;
   virtual bool      FeedForward(CNeuronBase *prevLayer) override;
   virtual bool      CalcHiddenGradient(CNeuronBase *prevLayer) override;
   virtual bool      CalcDeltaWeights(CNeuronBase *prevLayer, bool read)
                                                         override { return true; }
   virtual bool      UpdateWeights(int batch_size, TYPE learningRate,
                         VECTOR &Beta, VECTOR &Lambda) override { return true; }
   //--- методы работы с файлами
   virtual bool      Save(const int file_handle) override;
   virtual bool      Load(const int file_handle) override;
   //--- метод идентификации объекта
   virtual int       Type(void) override     const { return(defNeuronDropout); }
  };
```

Кроме унаследованных от родительского класса объектов мы создаем только один буфер данных и три переменные. Все три переменные имеют между собой математическую зависимость. Буфер вектора маскирования переопределяется на каждом прямом проходе. Таким образом, для восстановления работоспособности слоя *Dropout* нам достаточно сохранить объекты родительского класса и одну переменную.

Следовательно, метод сохранения данных будет довольно простым и коротким. В параметрах метод получает указатель на хендл файла для сохранения. В теле метода вызываем аналогичный метод родительского класса, в котором уже реализованы все контроли и сохранение объектов родительского класса. После успешного выполнения метода родительского класса мы лишь запишем в файл вероятность «выкидывания» нейронов из обработки. Выбор на данную

переменную пал очень просто — именно этот параметр указывает пользователь, а остальные вторичны и рассчитываются при инициализации класса.

```
bool CNeuronDropout::Save(const int file_handle)
  {
//--- вызов метода родительского класса
   if(!CNeuronBase::Save(file_handle))
      return false;
//--- сохраняем константу вероятности "выкидывания" элементов
   if(FileWriteDouble(file_handle, m_dOutProbability) <= 0)
      return false;
//---
   return true;
  }
```

Метод восстановления работоспособности слоя *CNeuronDropout::Load* выглядит немного сложнее метода сохранения. Как и метод сохранения данных, в параметрах метод загрузки данных получает хендл файла с данными для загрузки. Мы помним об основном правиле загрузке данных — считывание данных из файла осуществляется в строгом соответствии последовательности их записи. Следовательно, в теле метода мы первым вызываем аналогичный метод родительского класса, в котором уже реализованы все контроли и загрузка данных, унаследованных от родительского класса объектов.

```
bool CNeuronDropout::Load(const int file_handle)
  {
//--- вызов метода родительского класса
   if(!CNeuronBase::Load(file_handle))
      return false;
```

Обязательно проверяем результат выполнения метода родительского класса, так как в нем подтверждается не только загрузка данных, но и прохождение всех реализованных контролей.

После успешного выполнения метода родительского класса мы считываем из файла вероятность «выкидывания» нейронов. На основании полученного значения рассчитываем количество нейронов, подлежащих маскированию на каждой итерации прямого прохода, и начальное значение элементов буфера маскирования.

```
//--- считывание и восстановление констант
   m_dOutProbability = (TYPE)FileReadDouble(file_handle);
   m_iOutNumber = (int)(m_cOutputs.Total() * m_dOutProbability);
   m_dInitValue = (TYPE)(1.0 / (1.0 - m_dOutProbability));
```

В завершение метода восстановления работоспособности нашего слоя инициализируем буфер для записи вектора маскирования.

```
//--- инициализация буфера маскирования данных
   if(!m_cDropOutMultiplier.BufferInit(m_cOutputs.Rows(), m_cOutputs.Cols(),
                                                                m_dInitValue))
      return false;
//---
   return true;
  }
```

После успешной загрузки данных и инициализации объектов нашего слоя выходим из метода с положительным результатом.

На данном этапе мы завершаем работу над классом слоя *Dropout* стандартными средствами *MQL5*. В следующем разделе мы рассмотрим реализацию многопоточного алгоритма с использованием технологии *OpenCL*.

6.2.2 Организация многопоточных операций в Dropout

Мы продолжаем реализацию технологи *Dropout.* В предыдущих разделах мы уже полностью реализовали алгоритм работы данной технологии стандартными средствами *MQL5*. Теперь переходим к реализации алгоритма с использованием возможности многопоточных операций на *GPU* средствами *OpenCL.* В рамках этой книги мы уже не раз выполняли подобную операцию. Но все же напомню, что для ее реализации нам предстоит выполнить работу в двух направлениях. Вначале мы создаем программу *OpenCL*, а затем нам необходимо выполнить работу на стороне основной программы — организовать обмен данными между основной программой и контекстом *OpenCL*, в котором будет выполняться программа, а также осуществить вызов самой программы *OpenCL*.

Как всегда, данную работу начинаем с создания программы *OpenCL*. В данном случае нам не придется писать много кода на стороне *OpenCL*-программы. Более того, для реализации и прямого, и обратного прохода мы будем использовать один и тот же кернел. Как такое стало возможным? Давайте вспомним, какие операции нам надо реализовать.

При прямом проходе мы осуществляем маскирование данных. Непосредственно вектор-маску мы создаем средствами *MQL5* на стороне основной программы. Здесь же нам предстоит замаскировать исходные данные. Для этого будем поэлементно умножать тензор исходных данных на вектор-маску.

$$d_i = \frac{1}{q} x_i n_i$$

Следовательно, для прямого прохода нужно создать кернел для поэлементного умножения двух тензоров одинакового размера.

В процессе обратного прохода необходимо провести градиент ошибки через операцию маскирования. Давайте внимательно посмотри на формулу операции маскирования. *1/q* представляет собой константу, которая определяется на стадии инициализации класса и не меняется в течение всего процесса обучения и эксплуатации модели. x_i — элемент вектора маскирования, который может принимать только два значения: 1 или 0. Следовательно, весь процесс маскирования можно представить как умножение некоего исходного значения на константу. Как вы знаете, производная от такой операции является константа, на которую осуществлялось умножением.

$$\left(\frac{1}{q}x_i n_i\right)' = \frac{1}{q}x_i$$

В нашем случае для корректировки градиента ошибки нам необходимо поэлементно умножить градиент ошибки текущего слоя на вектор маскирования.

Таким образом, при прямом и обратном проходах мы поэлементно умножаем различные тензоры на вектор маскирования. Следовательно, для реализации обоих проходов на стороне программы *OpenCL* достаточно создать один кернел поэлементного умножения двух векторов. На самом деле это довольно простая задача. Использование векторных переменных для оптимизации процесса не делает ее сложнее.

Для этого создадим кернел маскирования *MaskMult*. В параметрах данный кернел получает указатели на три буфера данных, два из которых содержат исходные данные, а третий используется для записи результатов. Кроме того, так как предполагается использование векторных операций, то и общее количество потоков будет меньше числа операций. Значит, у нас не будет возможности определить размерность тензоров исходных данных по количеству запущенных потоков. Следовательно, для определения размерности тензоров мы передадим необходимую информацию о размерах в параметрах кернела.

В теле кернела мы определяем идентификатор текущего потока и переносим необходимые данные из буферов в векторные локальные переменные. Сразу умножим две векторные переменные. Полученный результат вернем из локальной векторной переменной в скалярный буфер данных.

```
__kernel void MaskMult(__global TYPE *inputs,
                       __global TYPE *mask,
                       __global TYPE *outputs,
                       int outputs_total)
  {
   const int n = get_global_id(0) * 4;
//---
  TYPE4 out = ToVect4(inputs, n, 1, outputs_total, 0) *
              ToVect4(mask, n, 1, outputs_total, 0);
   D4ToArray(outputs, out, n, 1, outputs_total, 0);
  }
```

Как видите, весь код кернела уместился в три строчки. Конечно, такое стало возможным благодаря использованию созданных ранее функций перевода данных скалярного буфера в локальную векторную переменную и обратно.

После создания кернела программы *OpenCL* переходим к реализации функционала на стороне основной программы. Вначале нам необходимо создать константы для обращения к элементам программы *OpenCL*. Для этого перейдем в файл *defines.mqh* и укажем константы для кернела и его параметров.

```
#define def_k_MaskMult                40

//--- маскирование данных
#define def_mask_inputs               0
#define def_mask_mask                 1
#define def_mask_outputs              2
#define def_mask_total                3
```

Затем мы перейдем к диспетчерскому классу модели. В методе инициализации контекста *OpenCL* изменим общее количество кернелов, а затем создадим кернел в контексте.

```
bool CNet::InitOpenCL(void)
  {
   ......
   if(!m_cOpenCL.SetKernelsCount(41))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
   ......

   if(!m_cOpenCL.KernelCreate(def_k_MaskMult, "MaskMult"))
     {
      m_cOpenCL.Shutdown();
      delete m_cOpenCL;
      return false;
     }
//---
   return true;
  }
```

После завершения подготовительной работы переходим к работе непосредственно с методами нашего класса *CNeuronDropout*. Как всегда, начнем с метода прямого прохода *CNeuronDropout::FeedForward*, в котором нужно реализовать следующие процессы:

- передача информации в контекст *OpenCL*;
- передача параметров в кернел *OpenCL*;
- постановка кернела в очередь выполнения;
- загрузка результатов работы кернела;
- очистка памяти контекста.

Переходим к методу прямого прохода. Изменения коснутся только блока многопоточных операций, а остальной код метода останется без изменений.

Класс *Dropout* может работать в двух режимах: обучение и промышленная эксплуатация. Выше мы создали кернел для работы в режиме обучения, но не подготовили кернел для режима промышленной эксплуатации. К примеру, операция копирования данных из буфера в буфер несложная, и мы можем выполнить ее средствами *MQL5*. Но здесь надо вспомнить, что мы минимизировали обмен данными между контекстом *OpenCL* и основной программой. Значит, на стороне основной программы содержание буферов будет неактуальным. Чтобы совершить операцию копирования данных, нужно сначала загрузить данные из контекста *OpenCL* в память основной программы, а затем скопировать данные из одного буфера в другой. После этого нужно

вернуть данные в контекст *OpenCL* уже в другом буфере для выполнения последующих операций. Это совсем не соответствует нашей политике минимизации операций обмена данными между контекстом *OpenCL* и основной программой.

Рассмотрим второй вариант — использование одного кернела в двух режимах работы. В режиме промышленной эксплуатации буфер маскирования заполняем единицами. Это тоже рабочий метод. При этом буфер маскирования мы готовим на стороне основной программы. *OpenCL* не предоставляет генератора псевдослучайных чисел. Значит, перед выполнением кернела нужно передать содержимое буфера маскирования из основной программы в контекст *OpenCL.* Но в режиме обучения это вынужденная мера. Зачем же тратить время на эту излишнюю операции в режиме промышленной эксплуатации? Может сделать шаг назад и подготовить еще один кернел?

Я нашел другое решение. Мы уже имеем кернел для выполнения линейной функции активации. Напомню ее математическое представление.

$$f(x) = ax + b$$

Если рассмотреть частный случай при *a=1* и *b=0*, то получим простое копирование данных.

$$f(x) = 1x + 0 = x$$

Для запуска кернела не нужно загружать дополнительных буферов в память контекста *OpenCL.* Вместо этого мы лишь передадим в параметры два целочисленных значения.

Алгоритм работы с кернелом остается прежним: проверяем наличие буферов в памяти контекста, передаем параметры кернела и осуществляем постановку кернела в очередь.

```
bool CNeuronDropout::FeedForward(CNeuronBase *prevLayer)
  {
   ......
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   if(!m_cOpenCL)
     {
   ......
     }
   else  // Блок OpenCL
     {
      //--- проверка флага режима работы
      if(!m_bTrain)
        {
         //--- проверка буферов данных
         CBufferType *inputs = prevLayer.GetOutputs();
         if(inputs.GetIndex() < 0)
            return false;
         if(m_cOutputs.GetIndex() < 0)
            return false;
         //--- передача параметров кернелу
         if(!m_cOpenCL.SetArgumentBuffer(def_k_LineActivation,
                                         def_activ_inputs, inputs.GetIndex()))
            return false;
         if(!m_cOpenCL.SetArgumentBuffer(def_k_LineActivation,
                                       def_activ_outputs, m_cOutputs.GetIndex()))
            return false;
         if(!m_cOpenCL.SetArgument(def_k_LineActivation,
                                                 def_activ_param_a, (TYPE)1))
            return false;
         if(!m_cOpenCL.SetArgument(def_k_LineActivation,
                                                 def_activ_param_b, (TYPE)0))
            return false;
         uint offset[] = {0};
         uint NDRange[] = {(uint)m_cOutputs.Total()};
         if(!m_cOpenCL.Execute(def_k_LineActivation, 1, offset, NDRange))
            return false;
        }
```

Для организации работы в процессе обучения повторим выше указанный алгоритм с постановкой в очередь нового кернела.

```
      else
        {
         //--- проверка буферов данных
         CBufferType *inputs = prevLayer.GetOutputs();
         if(inputs.GetIndex() < 0)
            return false;
         if(!m_cDropOutMultiplier.BufferCreate(m_cOpenCL))
            return false;
         if(m_cOutputs.GetIndex() < 0)
            return false;
         //--- передача параметров кернелу
         if(!m_cOpenCL.SetArgumentBuffer(def_k_MaskMult,
                                         def_mask_inputs, inputs.GetIndex()))
            return false;
         if(!m_cOpenCL.SetArgumentBuffer(def_k_MaskMult,
                                 def_mask_mask, m_cDropOutMultiplier.GetIndex()))
            return false;
         if(!m_cOpenCL.SetArgumentBuffer(def_k_MaskMult,
                                       def_mask_outputs, m_cOutputs.GetIndex()))
            return false;
         if(!m_cOpenCL.SetArgument(def_k_MaskMult, def_mask_total, total))
            return false;
         //--- постановка в очередь выполнения
         int off_set[] = {0};
         int NDRange[] = { (int)(total + 3) / 4};
         if(!m_cOpenCL.Execute(def_k_MaskMult, 1, off_set, NDRange))
            return false;
        }
     }
//---
   return true;
  }
```

На этом мы заканчиваем работу с кернелом прямого прохода и переходим к выполнению аналогичных операций для метода обратного прохода *CNeuronDropout::CalcHiddenGradient*. Напомню, что для обратного прохода в данном случае мы будем использовать те же кернелы. Алгоритм вызова не изменяется. Изменения коснутся лишь указания используемых буферов.

```
bool CNeuronDropout::CalcHiddenGradient(CNeuronBase *prevLayer)
  {
   ......
//--- разветвление алгоритма в зависимости от устройства выполнения операций
   ulong total = m_cOutputs.Total();
   if(!m_cOpenCL)
     {
   ......
     }

   else  // блок OpenCL
     {
      //--- проверка флага режима работы
      if(!m_bTrain)
        {
         //--- проверка буферов данных
         CBufferType *grad = prevLayer.GetGradients();
         if(grad.GetIndex() < 0)
            return false;
         if(m_cGradients.GetIndex() < 0)
            return false;
         //--- передача параметров кернелу
         if(!m_cOpenCL.SetArgumentBuffer(def_k_LineActivation,
                                 def_activ_inputs, m_cGradients.GetIndex()))
            return false;
         if(!m_cOpenCL.SetArgumentBuffer(def_k_LineActivation,
                                       def_activ_outputs, grad.GetIndex()))
            return false;
         if(!m_cOpenCL.SetArgument(def_k_LineActivation,
                                             def_activ_param_a, (TYPE)1))
            return false;
         if(!m_cOpenCL.SetArgument(def_k_LineActivation,
                                             def_activ_param_b, (TYPE)0))
            return false;
         uint offset[] = {0};
         uint NDRange[] = {(uint)m_cOutputs.Total()};
         if(!m_cOpenCL.Execute(def_k_LineActivation, 1, offset, NDRange))
            return false;
        }
```

И режим работы в процессе обучения.

```
      else
        {
         //--- проверка буферов данных
         CBufferType* prev = prevLayer.GetGradients();
         if(prev.GetIndex() < 0)
            return false;
         if(m_cDropOutMultiplier.GetIndex() < 0)
            return false;
         if(m_cGradients.GetIndex() < 0)
            return false;
         //--- передача параметров кернелу
         if(!m_cOpenCL.SetArgumentBuffer(def_k_MaskMult,
                                         def_mask_inputs, m_cGradients.GetIndex()))
            return false;
         if(!m_cOpenCL.SetArgumentBuffer(def_k_MaskMult,
                                    def_mask_mask, m_cDropOutMultiplier.GetIndex()))
            return false;
         if(!m_cOpenCL.SetArgumentBuffer(def_k_MaskMult,
                                              def_mask_outputs, prev.GetIndex()))
            return false;
         if(!m_cOpenCL.SetArgument(def_k_MaskMult, def_mask_total, total))
            return false;
         //--- постановка в очередь выполнения
         int off_set[] = {0};
         int NDRange[] = { (int)(total + 3) / 4 };
         if(!m_cOpenCL.Execute(def_k_MaskMult, 1, off_set, NDRange))
            return false;
        }
     }
//---
   return true;
  }
```

Обратите внимание, что при обратном проходе мы уже не загружаем данные маскирования в контекст *OpenCL*. Мы ожидаем, что он остался в памяти контекста с метода прямого прохода.

Поздравляю, мы завершили работу над методами класса реализации алгоритма *Dropout*. Мы проделали довольно большую работу и реализовали алгоритм *Dropout* и средствами *MQL5*, и в режиме многопоточных операций с использованием технологии *OpenCL*. Теперь можем протестировать модели. Но прежде я предлагаю посмотреть на реализацию такого подхода на языке *Python* в библиотеке *TensorFlow*.

6.2.3 Реализация Dropout в Python

Для построения моделей на языке *Python* ранее мы использовали библиотеку *Keras* для *TensorFlow*. В этой библиотеке уже есть готовая реализация слоя *Dropout*.

```
tf.keras.layers.Dropout(
    rate, noise_shape=None, seed=None, **kwargs
)
```

Слой *Dropout* случайным образом устанавливает входные единицы на 0 с частотой *rate* на каждой итерации в процессе обучения. Это помогает предотвратить переобучение модели. Исходные данные, не установленные на 0, масштабируются на 1/(1 - *rate*). Поэтому сумма передаваемых всех исходных данным остается неизменной.

Обратите внимание, что слой Dropout применяется только в том случае, если для него в поле *training* задано значение *True.* В противном случае никакие значения не маскируются. При обучении модели флаг *training* будет автоматически установлен в значение *True*. В других случаях пользователь может явно установить для *training* значение *True* при вызове слоя.

Это отличается от настройки *trainable* = *False* для слоя Dropout. В данном случае значение флага *trainable* не влияет на поведение слоя, так как Dropout не имеет никаких весовых коэффициентов, которые можно было бы заморозить во время обучения.

Конструктор слоя *Dropout* имеет следующие аргументы:

- *rate* — число с плавающей запятой в диапазоне от 0 до 1, которое представляет собой долю элементов исходных данных, маскируемых в процессе обучения;
- *noise_shape* — одномерный целочисленный тензор, представляющий форму двоичной маски исключения в виде (*batch_size, timesteps, features*). Форма будет умножена на тензор исходных данных. Например, если исходные данные имеют форму и вы хотите, чтобы маска исключения была одинаковой для всех временных шагов, вы можете использовать *noise_shape=(batch_size, 1, features);*
- *seed* — целое число для использования в качестве случайного начального числа.

При вызове слоя допускается использование двух аргументов:

- *inputs* — тензор исходных данных, допускается использование тензора любого ранга;
- *training* — логический флаг, указывающий режим работы слоя.

Для проверки эффективности использования технологии *Dropout* мы создадим скрипт и обучим несколько моделей с использованием данного слоя. Мы не будем создавать слишком сложных моделей. Вместо этого возьмем скрипт *batch_norm.py,* который использовали при тестировании пакетной нормализации. Создадим копию данного скрипта в файле *dropout.py.* В каждую модель добавим слои *Dropout.*

Сначала добавим два слоя *Dropout* в модель с одним скрытым слоем без использования пакетной нормализации. Вставлять новые слои будем перед каждым полносвязным слоем.

```
# Добавление Dropout в модель с одним скрытым слоем
model1do = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
                             keras.layers.Dropout(0.3),
                             keras.layers.Dense(40, activation=tf.nn.swish),
                             keras.layers.Dropout(0.3),
                             keras.layers.Dense(targets, activation=tf.nn.tanh)
                            ])
model1do.compile(optimizer='Adam',
                 loss='mean_squared_error',
                 metrics=['accuracy'])
model1do.summary()
```

Обратите внимание, что во всех слоях *Dropout* мы будем маскировать 30% нейронов предыдущего слоя.

Затем мы аналогичным образом добавим два слоя *Dropout* в модель с одним скрытым слоем и пакетной нормализацией исходных данных. Надо сказать, что здесь мы немного лукавим. В настоящее время не рекомендуется одновременно использовать в рамках оной модели пакетную нормализацию и *Dropout*, так как это только снизит общий результат модели. Давайте проверим это утверждение на практических примерах.

```
# Добавление Dropout в модель с пакетной нормализацией исходных данных
# и одним скрытым слоем
model1bndo = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
                               keras.layers.BatchNormalization(),
                               keras.layers.Dropout(0.3),
                               keras.layers.Dense(40, activation=tf.nn.swish),
                               keras.layers.Dropout(0.3),
                               keras.layers.Dense(targets, activation=tf.nn.tanh)
                              ])
model1bndo.compile(optimizer='Adam',
                   loss='mean_squared_error',
                   metrics=['accuracy'])
model1bndo.summary()
```

Аналогичным образом мы добавим пакеты *Dropout* в модели с тремя скрытыми слоями.

```
# Добавление Dropout в модель с тремя скрытыми слоями
model2do = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
                             keras.layers.Dropout(0.3),
                             keras.layers.Dense(40, activation=tf.nn.swish),
                             keras.layers.Dropout(0.3),
                             keras.layers.Dense(40, activation=tf.nn.swish),
                             keras.layers.Dropout(0.3),
                             keras.layers.Dense(40, activation=tf.nn.swish),
                             keras.layers.Dropout(0.3),
                             keras.layers.Dense(targets, activation=tf.nn.tanh)
                            ])
model2do.compile(optimizer='Adam',
                 loss='mean_squared_error',
                 metrics=['accuracy'])
model2do.summary()

# Добавление Dropout в модель с пакетной нормализацией исходных данных
# и трех скрытых слоев
model2bndo = keras.Sequential([keras.layers.InputLayer(input_shape=inputs),
                               keras.layers.BatchNormalization(),
                               keras.layers.Dropout(0.3),
                               keras.layers.Dense(40, activation=tf.nn.swish),
                               keras.layers.BatchNormalization(),
                               keras.layers.Dropout(0.3),
                               keras.layers.Dense(40, activation=tf.nn.swish),
                               keras.layers.BatchNormalization(),
                               keras.layers.Dropout(0.3),
                               keras.layers.Dense(40, activation=tf.nn.swish),
                               keras.layers.Dropout(0.3),
                               keras.layers.Dense(targets, activation=tf.nn.tanh)
                              ])
model2bndo.compile(optimizer='Adam',
                 loss='mean_squared_error',
                 metrics=['accuracy'])
model2bndo.summary()
```

После создания моделей добавим код для запуска процесса обучения новых моделей.

```
history1do = model1do.fit(train_data, train_target,
                      epochs=500, batch_size=1000,
                      callbacks=[callback],
                      verbose=2,
                      validation_split=0.1,
                      shuffle=True)
model1do.save(os.path.join(path,'perceptron1do.h5'))

history1bndo = model1bndo.fit(train_nn_data, train_nn_target,
                      epochs=500, batch_size=1000,
                      callbacks=[callback],
                      verbose=2,
                      validation_split=0.1,
                      shuffle=True)
model1bndo.save(os.path.join(path,'perceptron1bndo.h5'))

history2do = model2do.fit(train_data, train_target,
                      epochs=500, batch_size=1000,
                      callbacks=[callback],
                      verbose=2,
                      validation_split=0.1,
                      shuffle=True)
model2do.save(os.path.join(path,'perceptron2do.h5'))

history2bndo = model2bndo.fit(train_nn_data, train_nn_target,
                      epochs=500, batch_size=1000,
                      callbacks=[callback],
                      verbose=2,
                      validation_split=0.1,
                      shuffle=True)
model2bndo.save(os.path.join(path,'perceptron2bndo.h5'))
```

Обязательно добавим запуск моделей на тестовой выборке.

```
test_loss1do, test_acc1do = model1do.evaluate(test_data, test_target,
                                              verbose=2)
test_loss1bndo, test_acc1bndo = model1bndo.evaluate(test_nn_data,
                                                    test_nn_target,
                                                    verbose=2)
test_loss2do, test_acc2do = model2do.evaluate(test_data, test_target,
                                              verbose=2)
test_loss2bndo, test_acc2bndo = model2bndo.evaluate(test_nn_data,
                                                    test_nn_target,
                                                    verbose=2)
```

Кроме изменений в части обучения и тестирования моделей, дополним и блок отрисовки результатов работы моделей. Сначала изменим код, который создает графики динамики среднеквадратичной ошибки и *Accuracy* в процессе обучения. Изменения здесь не глобальные — мы лишь добавляем новые показатели на график.

```
# Отрисовка результатов обучения моделей с одним скрытым слоем
plt.figure()
plt.plot(history1.history['loss'], label='Normalized inputs train')
plt.plot(history1.history['val_loss'], label='Normalized inputs validation')
plt.plot(history1do.history['loss'], label='Normalized inputs\nvs Dropout train')
plt.plot(history1do.history['val_loss'],
                                label='Normalized inputs\nvs Dropout validation')
plt.plot(history1bn.history['loss'],
                        label='Unnormalized inputs\nvs BatchNormalization train')
plt.plot(history1bn.history['val_loss'],
                   label='Unnormalized inputs\nvs BatchNormalization validation')
plt.plot(history1bndo.history['loss'],
            label='Unnormalized inputs\nvs BatchNormalization and Dropout train')
plt.plot(history1bndo.history['val_loss'],
       label='Unnormalized inputs\nvs BatchNormalization and Dropout validation')
plt.ylabel('$MSE$ $loss$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics\n1 hidden layer')
plt.legend(loc='upper right',ncol=2)

plt.figure()
plt.plot(history1.history['accuracy'], label='Normalized inputs trin')
plt.plot(history1.history['val_accuracy'], label='Normalized inputs validation')
plt.plot(history1do.history['accuracy'],
                                    label='Normalized inputs\nvs Dropout train')
plt.plot(history1do.history['val_accuracy'],
                                label='Normalized inputs\nvs Dropout validation')
plt.plot(history1bn.history['accuracy'],
                        label='Unnormalized inputs\nvs BatchNormalization train')
plt.plot(history1bn.history['val_accuracy'],
                   label='Unnormalized inputs\nvs BatchNormalization validation')
plt.plot(history1bndo.history['accuracy'],
            label='Unnormalized inputs\nvs BatchNormalization and Dropout train')
plt.plot(history1bndo.history['val_accuracy'],
      label='Unnormalized inputs\nvs BatchNormalization and Dropout validation')
plt.ylabel('$Accuracy$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics\n1 hidden layer')
plt.legend(loc='lower right',ncol=2)
```

```
# Отрисовка результатов обучения моделей с тремя скрытыми слоями
plt.figure()
plt.plot(history2.history['loss'], label='Normalized inputs train')
plt.plot(history2.history['val_loss'], label='Normalized inputs validation')
plt.plot(history2do.history['loss'], label='Normalized inputs\nvs Dropout train')
plt.plot(history2do.history['val_loss'],
                                  label='Normalizedinputs\nvs Dropout validation')
plt.plot(history2bn.history['loss'],
                         label='Unnormalized inputs\nvs BatchNormalization train')
plt.plot(history2bn.history['val_loss'],
                   label='Unnormalized inputs\nvs BatchNormalization validation')
plt.plot(history2bndo.history['loss'],
            label='Unnormalized inputs\nvs BatchNormalization and Dropout train')
plt.plot(history2bndo.history['val_loss'],
       label='Unnormalized inputs\nvs BatchNormalization and Dropout validation')
plt.ylabel('$MSE$ $loss$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics\n3 hidden layers')
plt.legend(loc='upper right',ncol=2)

plt.figure()
plt.plot(history2.history['accuracy'], label='Normalized inputs train')
plt.plot(history2.history['val_accuracy'], label='Normalized inputs validation')
plt.plot(history2do.history['accuracy'], label='Normalized inputs\nvs Dropout train')
plt.plot(history2do.history['val_accuracy'],
                                     label='Normalized inputs\nvs Dropout validation')
plt.plot(history2bn.history['accuracy'],
                              label='Unnormalized inputs\nvs BatchNormalization train')
plt.plot(history2bn.history['val_accuracy'],
                         label='Unnormalized inputs\nvs BatchNormalization validation')
plt.plot(history2bndo.history['accuracy'],
                 label='Unnormalized inputs\nvs BatchNormalization and Dropout train')
plt.plot(history2bndo.history['val_accuracy'],
            label='Unnormalized inputs\nvs BatchNormalization and Dropout validation')
plt.ylabel('$Accuracy$')
plt.xlabel('$Epochs$')
plt.title('Model training dynamics\n3 hidden layers')
plt.legend(loc='lower right',ncol=2)
```

Последние изменения в скрипте будут касаться отображения результатов проверки моделей на тестовой выборке. Здесь помимо добавления новых данных мы разделим графики: отдельно покажем результаты моделей с одним скрытым слоем, а результаты моделей с тремя скрытыми слоями вынесем на новую диаграмму.

```
plt.figure()
plt.bar(['Normalized inputs','\n\nNormalized inputs\nvs Dropout',
         'Unnormalized inputs\nvs BatchNornalization',
         '\n\nUnnormalized inputs\nvs BatchNornalization and Dropout'],
        [test_loss1,test_loss1do,
         test_loss1bn,test_loss1bndo])
plt.ylabel('$MSE$ $loss$')
plt.title('Test results\n1 hidden layer')

plt.figure()
plt.bar(['Normalized inputs','\n\nNormalized inputs\nvs Dropout',
         'Unnormalized inputs\nvs BatchNornalization',
         '\n\nUnnormalized inputs\nvs BatchNornalization and Dropout'],
        [test_loss2,test_loss2do,
         test_loss2bn,test_loss2bndo])
plt.ylabel('$MSE$ $loss$')
plt.title('Test results\n3 hidden layers')

plt.figure()
plt.bar(['Normalized inputs','\n\nNormalized inputs\nvs Dropout',
         'Unnormalized inputs\nvs BatchNornalization',
         '\n\nUnnormalized inputs\nvs BatchNornalization and Dropout'],
        [test_acc1,test_acc1do,
         test_acc1bn,test_acc1bndo])
plt.ylabel('$Accuracy$')
plt.title('Test results\n1 hidden layer')

plt.figure()
plt.bar(['Normalized inputs','\n\nNormalized inputs\nvs Dropout',
         'Unnormalized inputs\nvs BatchNornalization',
         '\n\nUnnormalized inputs\nvs BatchNornalization and Dropout'],
        [test_acc2,test_acc2do,
         test_acc2bn,test_acc2bndo])
plt.ylabel('$Accuracy$')
plt.title('Test results\n3 hidden layers')

plt.show()
```

В остальном код скрипта остался без изменений.

С результатами тестирования моделей мы познакомимся в следующем разделе.

6.2.4 Сравнительное тестирование моделей с Dropout

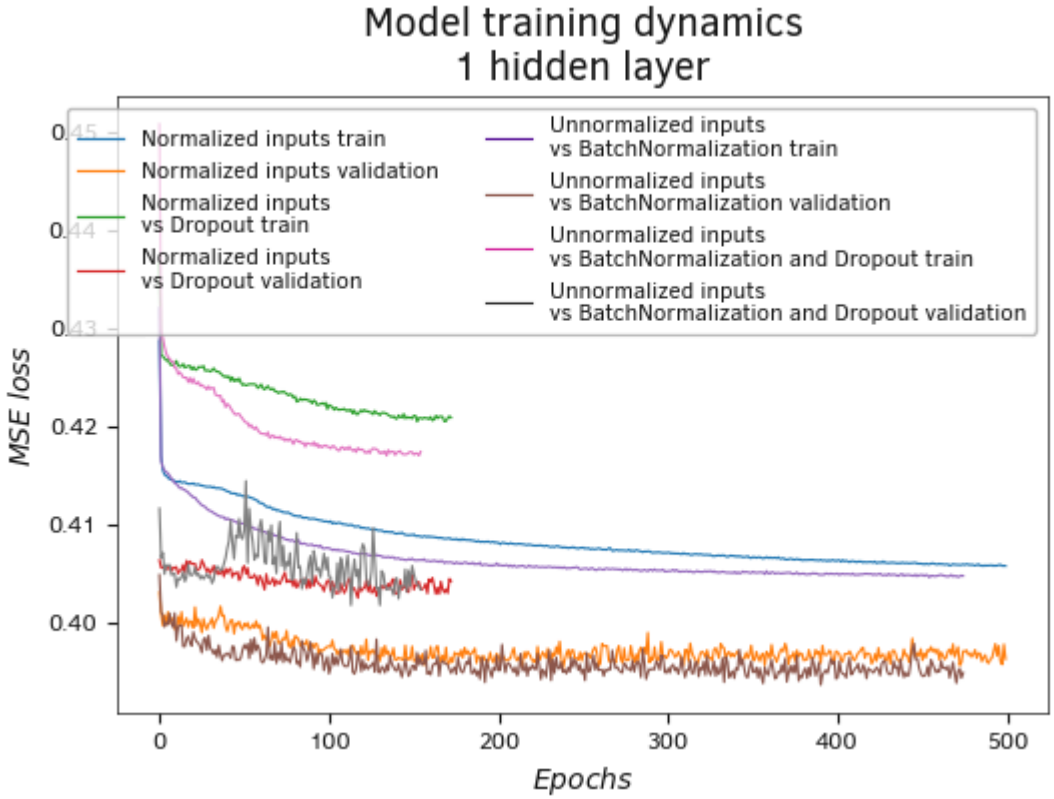
Еще один этап работы с нашей библиотекой завершен. Мы изучили метод *Dropout*, который борется с явлением совместной адаптации признаков, и построили класс для реализации данного алгоритма в наших моделях. В предыдущем разделе мы собрали скрипт на языке *Python* для сравнительного тестирования моделей с использованием данного метода и без. Предлагаю посмотреть на результаты такого тестирования.

Сначала посмотрим на график тестового обучения моделей с одним скрытым слоем. Динамика среднеквадратичной ошибки моделей с использование *Dropout* оказалась хуже, чем у моделей без его использования. Это касается как модели обучаемой на нормализованных данных, так и

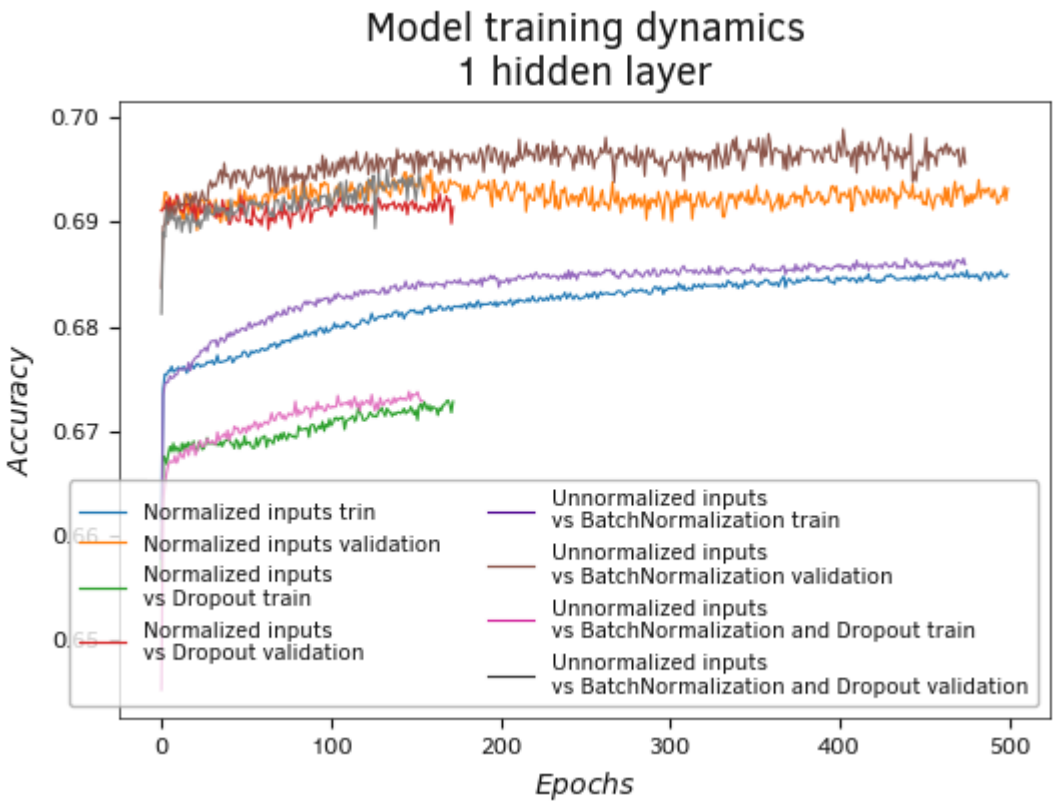
модели с использованием слоя пакетной нормализации для предварительной обработки исходных данных. Можно заметить, что обе модели с использованием слоя *Dropout* отработали синхронно. Их линии на графике практически наложены друг на друга как в процессе обучения, так и на этапе валидации.

Аналогичные выводы можно сделать и при анализе динамики метрики *Accuracy*. Но в отличие от *MSE*, в процессе валидации значения показателя близки к значениям других моделей.

Проверка моделей на тестовой выборке также показала ухудшение показателей моделей при использовании слоя *Dropout*, как для среднеквадратичной ошибки, так и по метрике *Accuracy.* О причинах такого явления можно только догадываться. Вполне возможно, что одной из причин может быть использование слишком простых моделей. Обучаемые модели и так содержали не слишком много нейронов, а маскирование части из них понижает способности модели, которые и так ограничены небольшим количеством используемых нейронов.

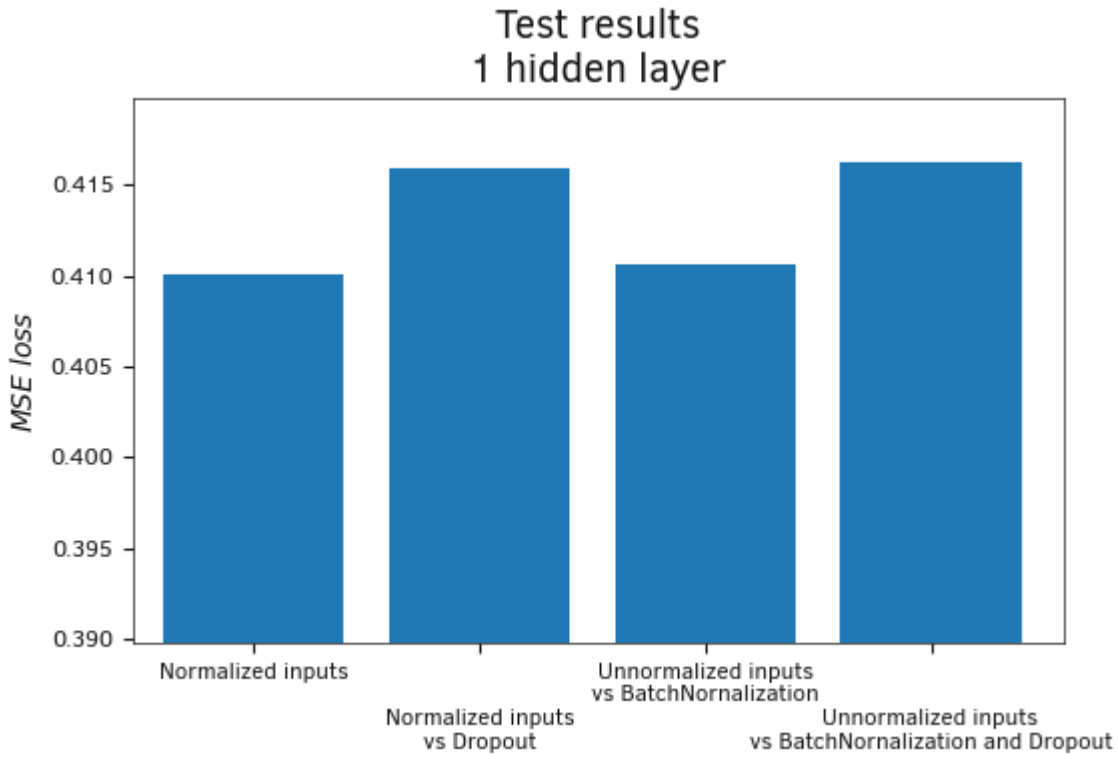


Сравнительное тестирование моделей с Dropout

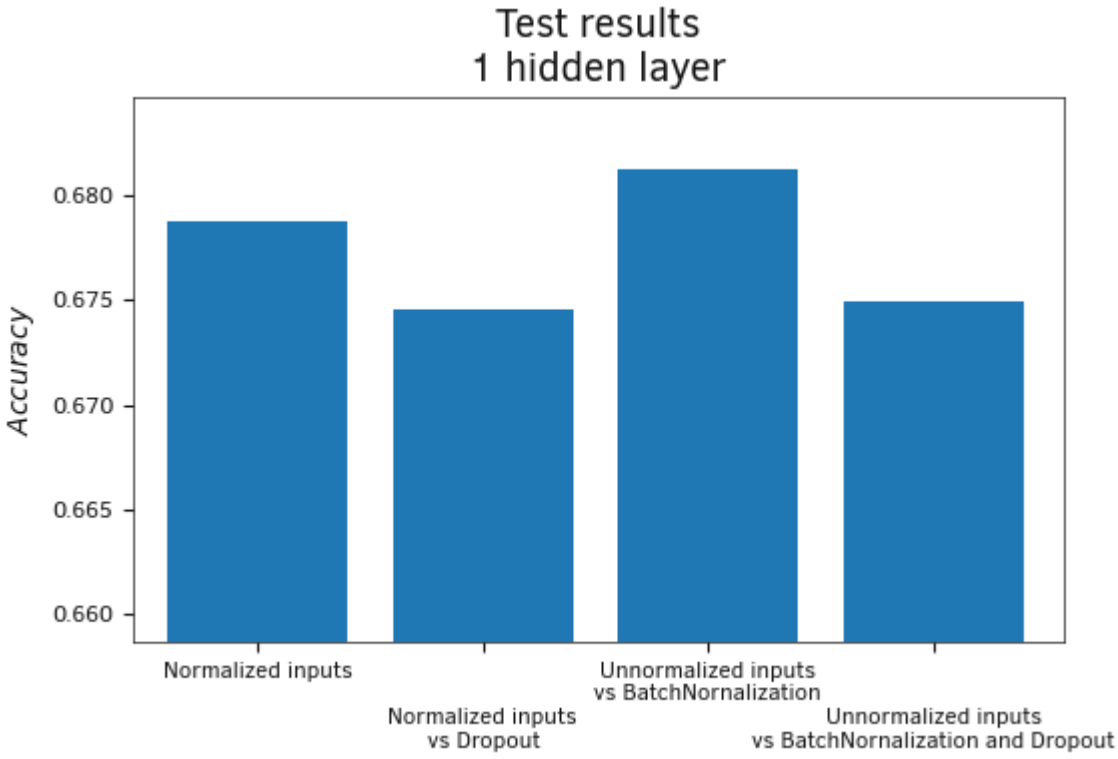


Сравнительное тестирование моделей с Dropout

С другой стороны, на стадии отбора данных мы с вами выбирали некоррелируемые показатели. Вполне возможно, что в связи с малым количеством используемых показателей и отсутствием корреляции между ними в наших моделях не так уж сильно развита взаимная адаптация признаков. Как следствие, негативное влияние использования *Dropout* в виде ухудшения разрешающей способности модели превысило позитивное влияние метода.



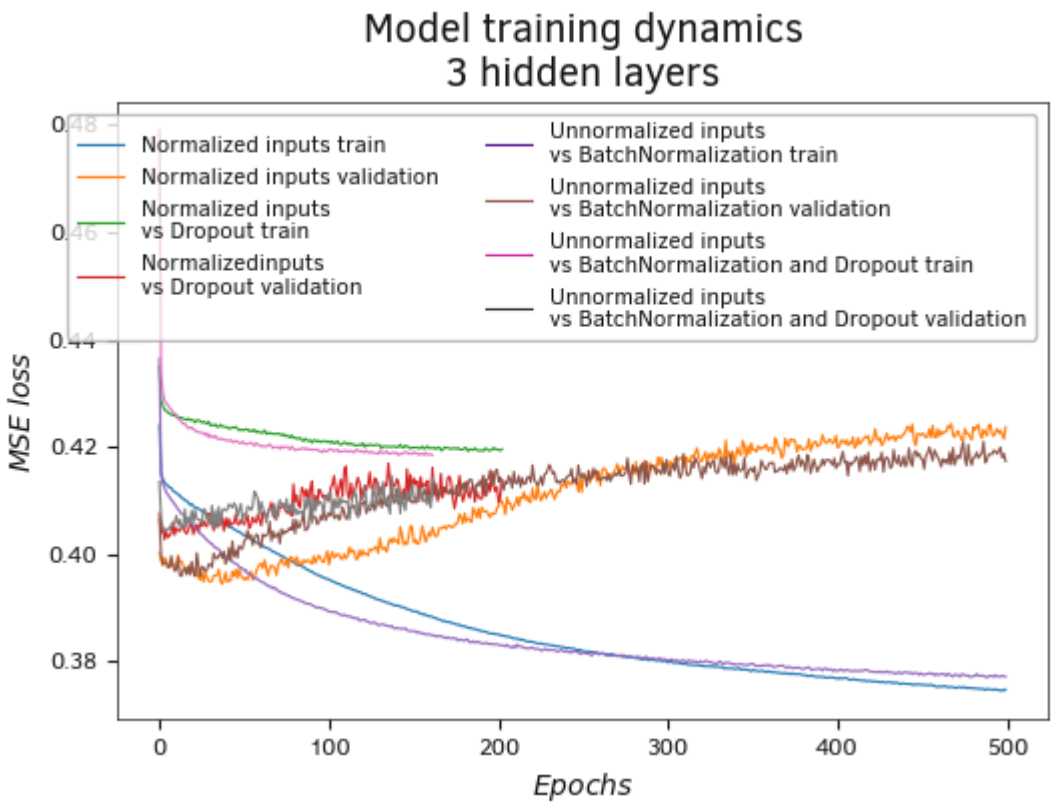
Сравнительное тестирование моделей с Dropout (тестовая выборка)



Сравнительное тестирование моделей с Dropout (тестовая выборка)

Это лишь мои предположения. Для получения конкретных выводов у меня недостаточно информации. Потребуются дополнительные тесты. Но это больше направленность научной

работы. Наша же цель — практическое использование моделей. Мы проводим эксперименты с различными архитектурными решениями и выбираем лучшее для каждой конкретной задачи.



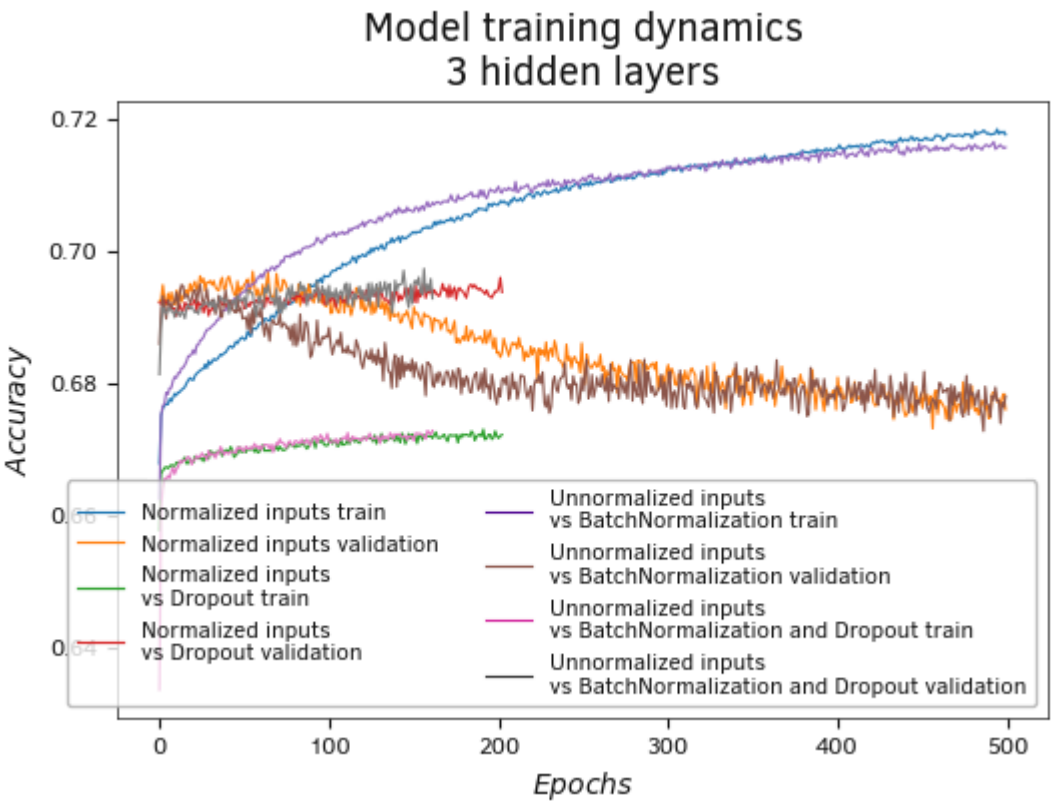
Сравнительное тестирование моделей с Dropout

Второй тест был проведен с использованием слоя *Dropout* перед каждым полносвязным слоем в моделях с тремя скрытыми слоями. И опять мы видим, что модели с использованием слоя *Dropout* проигрывают другим моделям в процессе обучения. Но зато в процессе валидации картина несколько меняется. В то время, как модели без использования слоев *Dropout* с увеличением числа эпох обучения снижают свои показатели на этапе валидации, модели с использованием данной технологии немного улучшают свои позиции или остаются на прежнем уровне.

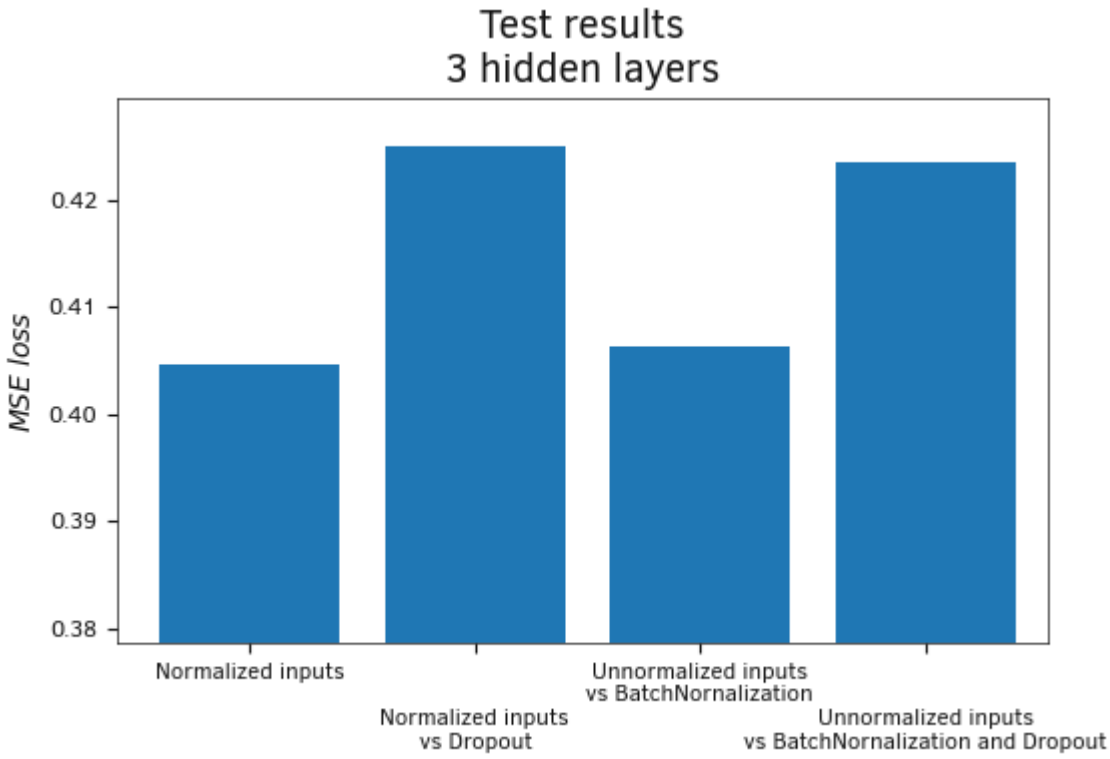
Это позволяет сделать вывод, что использование слоя *Dropout* снижает вероятность переобучения модели.

На графике динамики значений метрики *Accuracy* в процессе обучения мы видим подтверждение сделанному выше выводу. С ростом количества эпох обучения моделей без использования слоя *Dropout* наблюдается увеличение разрыва между значениями показателя в процессе обучения и на этапе валидации. Это свидетельствует о переобучении модели. В то же время у моделей с использованием технологии *Dropout* разрыв между показателями наоборот уменьшается. Это подтверждает сделанный ранее вывод о том, что использование слоя *Dropout* снижает склонность модели к переобучению.

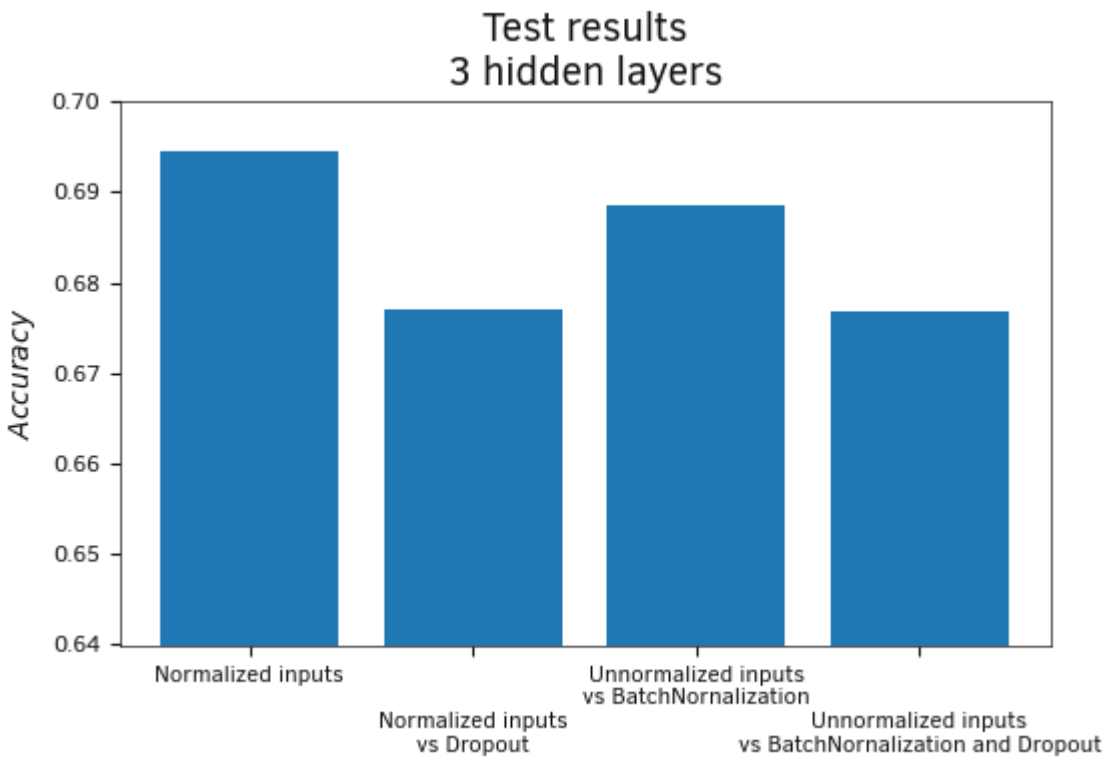
На тестовой выборке наблюдается отставание моделей с использованием слоя *Dropout* по обоим показателям.



Сравнительное тестирование моделей с Dropout



Сравнительное тестирование моделей с Dropout (тестовая выборка)



Сравнительное тестирование моделей с Dropout (тестовая выборка)

В данном разделе мы провели сравнительное тестирование моделей с использованием технологии *Dropout* и без. Проведенные тесты позволяют сделать следующие выводы:

- использование технологии *Dropout* позволяет снизить риск переобучения модели;
- эффективность технологии *Dropout* повышается с ростом модели.

7. Проверка торговых возможностей модели

Мы с вами проделали довольно большую работу по изучению различных архитектурных решений организации нейронных сетей. Мы создали целую библиотеку для создания различных нейронных слоев и теперь с ее помощью можем строить различные модели нейронных сетей для поиска наилучшего варианта решения своих задач. И это все, конечно, очень хорошо и полезно. Но давайте вспомним для чего мы это делаем. Мы не занимаемся этим ради науки или собственного просвещения, хотя и это отнюдь не плохой повод для изучения чего-либо. Но в данном случае мы занялись изучением организации нейронных сетей и их архитектурных решений с практической целью — найти решение для использования на финансовых рынках. Тут есть два видения такого решения:

- создание индикатора на базе модели нейронной сети;
- создание электронного советника, способного совершать торговые операции по сигналам модели нейронной сети.

Сейчас мы не будем обсуждать, какой из указанных выше вариантов предпочтительнее. На самом деле, это риторический вопрос, потому что он зависит от личных предпочтений пользователя. В любом случае, нам нужно организовать правильную работу модели и трактовку ее сигналов.

При этом мы бы хотели оценить ожидаемую доходность нашей модели. Для проведения подобной работы терминал *MetaTrader 5* предлагает использовать *Тестер стратегий*.

В этой главе мы переходим от теоретического изучения и создания нейронных сетей к практическому применению разработанных моделей в финансовом секторе, с акцентом на их торговые возможности. Наша цель — оценить эффективность нейронных сетей для создания индикаторов и электронных советников, способных выполнять торговые операции на финансовых рынках. Мы начнем с изучения функционала Тестера стратегий в MetaTrader 5, который является ключевым инструментом для оценки работы наших моделей.

Далее мы перейдем к созданию шаблона советника с использованием языка программирования MQL5, что позволит нам применять наши модели в реальных торговых условиях. Затем мы сконцентрируем внимание на создании модели для тестирования, обсуждая, как правильно подготовить и сконфигурировать модель для получения максимально точных и полезных результатов.

После этого мы обсудим определение параметров советника, что включает настройку различных параметров и опций, оптимизирующих работу советника в соответствии с торговыми стратегиями и целями пользователя. Наконец, мы проведем тестирование модели на новых данных, что является критически важным шагом для оценки способности модели адаптироваться к изменяющимся рыночным условиям и предсказывать будущие торговые сигналы.

Таким образом, эта глава направлена на практическое применение разработанных нами нейронных сетей в реальных торговых стратегиях, подчеркивая важность тестирования и оптимизации моделей для достижения наилучших результатов на финансовых рынках.

7.1 Знакомство с тестером стратегий MetaTrader 5

Для проверки качества торгового робота в *MetaTrader 5* встроен тестер торговых стратегий. Он позволяет еще до запуска советника на торговом счете определить его эффективность и подобрать наилучшие входные параметры.

Вся работа тестера торговых стратегий строится на истории котировок валют и акций. Тестер автоматически скачивает с торгового сервера брокера тиковую историю и учитывает спецификацию контрактов — разработчику ничего не нужно делать руками. Это позволяет легко и максимально достоверно воспроизводить все условия торгового окружения — вплоть до миллисекундных интервалов между поступлениями тиков на разных символах. Робот анализирует накопленные котировки и совершает виртуальные сделки в соответствии с заложенным в него алгоритмом. Это позволяет оценить, как бы данная стратегия торговала в прошлом.

Кроме того, тестер стратегий в *MetaTrader 5* является мультивалютным. Все тестируемые в нем роботы могут получить информацию обо всех финансовых инструментах, доступных на зарегистрированном в терминале счете, и могут совершать на них торговые операции. Таким образом, инструмент позволяет проверять даже сложные советники, которые способны анализировать сразу несколько валют или акций и корреляцию между ними.

Основным преимуществом такого тестирования является оценка торгового робота в условиях максимально приближенным к реальным без его реальной работы на рынке. Более того, это занимает намного меньше времени, так как исторические тики тестером генерируются гораздо быстрее реального рынка. Это бесспорное преимущество тестера стратегий, но далеко не все его возможности.

Тестер стратегий *MetaTrader 5* предлагает несколько режимов тестирования. Они позволяют выбрать оптимальное соотношение скорости и качества в соответствии с потребностями

пользователя. Режим «Все тики» предназначается для наиболее точной проверки, в этом случае моделируемые условия будут наиболее приближены к реальным. Режим «1 minute OHLC» позволяет протестировать стратегию быстрее с достаточным уровнем точности. Если нужна очень быстрая и грубая оценка, выбирайте режим «Только цены открытия» — тестирование будет проходить только по ценам открытия баров. Самое высокое качество предлагает режим «Все тики на основе реальных», однако при этом затраты времени будут максимальными.

Возможности тестера не ограничиваются только проверкой. Его можно использовать и для решения массовых математических задач оптимизации параметров. В режиме математических вычислений не используется торговая история и не моделируется рыночное окружение — выполняются только заложенные в советнике математические расчеты.

Стресс-тестирование — это возможность еще больше приблизить условия проверки торгового робота к реальным. Режим произвольных задержек исполнения эмулирует сетевые задержки при передаче и обработке торговых запросов, а также моделирует задержки исполнения приказов дилерами при реальной торговле.

Одной из главных особенностей тестера стратегий является представление результатов проверки торговых советников. Это не только сухие цифры — сколько заработал робот за время тестирования. Это еще и масса статистических показателей работы:

- процентное соотношение прибыли и убытка,
- количество удачных и неудачных сделок,
- фактор риска,
- ожидание выигрыша.

И это далеко не полный их список. Кроме того, результаты тестирования стратегий также предоставляются в графическом виде, что делает анализ торговой стратегии еще более удобным и наглядным.

Существующий режим визуального тестирования позволяет в режиме реального времени отслеживать торговлю робота на исторических ценовых данных. Все сделки эксперта отображаются на графике, и их легко анализировать. Процесс тестирования можно замедлить или поставить на паузу, чтобы посмотреть, как осуществляется торговля на том или ином временном промежутке.

Режим визуализации — это не только возможность самому увидеть, как торгует робот. Помимо этого, он позволяет проверить работу пользовательских технических индикаторов. Например, перед покупкой через Маркет вы можете оценить его поведение на исторических данных.

Важной функцией тестера стратегий является оптимизация торгового робота, которая позволяет подобрать лучшие входные параметры для конкретного советника. Различные режимы оптимизации позволяют найти оптимальные параметры, чтобы торговый робот стал максимально прибыльным и устойчивым, отличался минимальной рискованностью и так далее.

В процессе оптимизации происходит тестирование одного торгового робота с разными входными параметрами. По завершению тестов результаты прогонов можно сравнить между собой и выбрать настройки, которые наилучшим образом соответствуют предъявляемым к роботу требованиям.

Количество комбинаций входных параметров при оптимизации может достигать десятков или сотен тысяч. В итоге оптимизация может превратиться в очень длительный процесс, который все же можно существенно сократить при помощи генетических алгоритмов. Эта функция отключает

последовательный перебор всех комбинаций входных параметров и выбирает только те, которые наилучшим образом отвечают критериям оптимизации. На последующих этапах "оптимальные" комбинации скрещиваются до тех пор, пока результаты не перестанут улучшаться. Таким образом, количество комбинаций и общее время оптимизации сокращаются в разы.

В дополнение к этому тестер стратегий работает в могопоточном режиме и позволяет задействовать все ядра процессора. При этом на каждом ядре будет запущен советник со своим набором параметров. Кроме того, для большого пула задач тестер стратегий дает возможность подключать облачные вычисления благодаря использованию технологии *MQL5 Cloud Network*. Это сеть облачных вычислений, объединяющая в себе тысячи компьютеров по всему миру. Тестер стратегий может использовать ее практически безграничные вычислительные мощности. При помощи сети *MQL5 Cloud Network* оптимизация, которая заняла бы месяцы в обычном режиме, может быть выполнена за считанные часы.

В тестере стратегий доступны мощные инструменты визуального анализа результатов оптимизации в режимах 2D и 3D. Например, в двухмерном представлении можно сразу проанализировать зависимости итогового результата от двух показателей, а в 3D — увидеть всю картину поиска наилучшего результата при оптимизации.

Помимо встроенных возможностей, вы можете использовать собственные методы визуализации. При этом нет необходимости подготавливать данные, экспортировать и обрабатывать их в стороннем приложении. Просто выведите результаты оптимизации на экран прямо во время ее выполнения.

Встроенная функция форвард-тестирования позволяет избавиться от "переоптимизации", или подгонки параметров. С включением этой опции история котировок валют и акций делится на две части. Непосредственно оптимизация происходит на первом отрезке истории, а второй используется только для подтверждения полученных результатов. Если на обоих отрезках эффективность торгового робота одинаково высока, значит, торговая система обладает наилучшими параметрами и подгонка параметров практически исключена.

Тестер торговых стратегий — это незаменимый инструмент для разработчиков экспертов. Без него практически невозможно написать эффективного торгового робота. Он позволяет сэкономить время и сделать по-настоящему прибыльный инструмент для использования на финансовых рынках.

7.2 Создание шаблона советника средствами MQL5

Итак, для проверки работы нашей модели в тестере стратегий необходимо обернуть ее в торговый робот. Поэтому в данном разделе я решил привести небольшой шаблон электронного советника с использованием нейронной сети в качестве основного и единственного блока принятия решения. Сразу скажу, что это лишь шаблон, основная цель которого — продемонстрировать принципы и подходы реализации. Его код сильно упрощен и не предназначен для использования на реальных счетах. Впрочем, он вполне работоспособен и может быть использован в качестве базы для создания рабочего советника. Также хочу предупредить, что финансовые рынки являются высоко рисковым способом инвестирования средств. Вы совершаете все свои операции на свой страх и риск, полностью под свою ответственность, в том числе, если вы используете электронные советники на своих счетах. Конечно, если авторы таких роботов не дают вам свои гарантии, о чем могут быть ваши личные договоренности. Что касается электронных советников, то прежде, чем их устанавливать на свои реальные торговые счета и доверять им свои средства, тщательно изучите их параметры и

варианты настроек, а также протестируйте их работоспособность в различных режимах в тестере стратегий и на демо-счетах.

Надеюсь, это отступление всем понятно. Теперь давайте посмотрим на реализацию шаблона. Прежде всего, как я уже сказал, представленный шаблон сильно упрощен, в нем отсутствует ряд обязательных для советников функций, которые не относятся к режиму работы нашей модели. В частности, в советнике полностью отсутствует блок управления капиталом, так называемый мани-менеджмент. Для упрощения мы используем фиксированный лот *Lot*. Также мы используем фиксированный стоп-лосс *StopLoss* и задаем диапазон для тейк-профита в виде минимальной границы *MinTarget* и максимальной *MaxTP*. Такой подход к тейк-профиту вызван тем, что в тестируемых нами моделях второй целевой показатель как раз и показывал расстояние до ближайшего будущего экстремума.

```
sinput string          Model = "our_model.net";
sinput int             BarsToPattern = 40;
sinput bool            Common = true;
input ENUM_TIMEFRAMES  TimeFrame = PERIOD_M5;
input double           TradeLevel=0.9;
input double           Lot = 0.01;
input int              MaxTP= 500;
input double           ProfitMultiply = 0.8;
input int              MinTarget=100;
input int              StopLoss=300;
```

Также я выбрал упрощенный подход к использованию модели. Я не стал создавать и обучать модель внутри советника, а зашел с другой стороны. Во всех создаваемых нами скриптах для тестирования архитектурных решений нейронных слоев мы сохраняли обученные модели. Так почему бы нам просто не загрузить одну из обученных моделей? Вы можете создать и обучить свою модель, а потом просто указать имя файла с обученной моделью во внешнем параметре *Model* и использовать ее. Остается лишь указать место хранения файла *Common*, количество баров описания одного паттерна *BarsToPattern* и используемый таймфрейм *TimeFrame*. Также для принятия решения мы укажем минимальную прогнозируемую вероятность получения прибыли *TradeLevel*.

Чтобы повысить вероятность закрытия сделки по тейк-профиту, добавим параметр *ProfitMultiply,* в котором укажем коэффициент доверия прогнозируемой силы движения. Иными словами, при указании тейк-профита открываемой позиции мы будем корректировать размер ожидаемого движения на данный коэффициент.

Использование параметра *Common* для указания места хранения обученной модели является довольно важным показателем, как бы это ни показалось странным. Дело в том, что доступ к файлам в *MetaTrader 5* ограничен песочницей. Каждый терминал, установленный на компьютере, имеет свою песочницу. Таким образом, каждый из двух терминалов, установленных на одном компьютере, работает в своей песочнице и не мешает второму. Для случаев, когда необходим обмен данными между терминалами на одном компьютере, используется отдельная общая папка. Именно на ее использование и указывает значение *true* в параметре *Common*.

При использовании режима оптимизации тестера стратегий каждый тестовый агент работает в своей отдельной песочнице, даже в рамах одного терминала. Поэтому, чтобы обеспечить равный доступ к обученной модели всем агентам тестирования, нужно поместить ее в общую папку терминалов и указать соответствующее значение флага.

После объявления внешних параметров нашего советника мы в глобальном пространстве подключаем нашу библиотеку для работы с моделями нейронных сетей *neuronnet.mqh* и стандартную библиотеку для совершения торговых операций *Trade\Trade.mqh.*

```
#include "..\..\Include\NeuroNetworksBook\realization\neuronnet.mqh"
#include <Trade\Trade.mqh>

CNet *net;
CTrade *trade;
datetime lastbar = 0;
int h_RSI;
int h_MACD;
```

Следом мы объявляем глобальные переменные:

- *net* — указатель на объект модели;
- *trade* — указатель на объект совершения торговых операций;
- *lastbar* — время последнего анализируемого бара, используется для проверки события открытия новой свечи;
- *h_RSI* — хендл индикатора *RSI*;
- *h_MACD* — хендл индикатора *MACD*.

Наш шаблон будет содержать минимальный набор функций. Но это отнюдь не означает, что ваш советник должен содержать их ровно столько же.

В функции *OnInit* осуществляем инициализацию советника. В начале функции создадим новый экземпляр объекта нейронной сети и сразу проверим результат операции. В случае успешного создания нового объекта загрузим модель из указанного файла. И, конечно, проверим результат выполнения операций.

```
int OnInit()
  {
//---
   if(!(net = new CNet()))
     {
      PrintFormat("Error creating Net: %d", GetLastError());
      return INIT_FAILED;
     }
   if(!net.Load(Model, Common))
     {
      PrintFormat("Error loading mode %s: %d", Model, GetLastError());
      return INIT_FAILED;
     }
   net.UseOpenCL(UseOpenCL);
```

После загрузки модели подгружаем используемые индикаторы. Напомню, что в рамках этой книги мы обучали модели на выборке исторических данных двух индикаторов — *RSI* и *MACD*. Как всегда, проверяем результат проведения операции.

```
   h_RSI = iRSI(_Symbol, TimeFrame, 12, PRICE_TYPICAL);
   if(h_RSI == INVALID_HANDLE)
     {
      PrintFormat("Error loading indicator %s", "RSI");
      return INIT_FAILED;
     }
   h_MACD = iMACD(_Symbol, TimeFrame, 12, 48, 12, PRICE_TYPICAL);
   if(h_MACD == INVALID_HANDLE)
     {
      PrintFormat("Error loading indicator %s", "MACD");
      return INIT_FAILED;
     }
```

Следующим шагом мы создаем экземпляр объекта для совершения торговых операций. Проверяем результат создания объекта и устанавливаем тип исполнения ордеров.

```
void OnDeinit(const int reason)
  {
   if(!!net)
      delete net;
   if(!!trade)
      delete trade;
   IndicatorRelease(h_RSI);
   IndicatorRelease(h_MACD);
  }
```

В заключение функции мы установим начальное значение для времени последнего бара и выходим из функции.

Сразу за функцией инициализации мы создаем функцию деинициализации *OnDeinit*, в которой мы удаляем созданные в программе объекты. Также закрываем индикаторы.

```
void OnDeinit(const int reason)
  {
   if(CheckPointer(net) == POINTER_DYNAMIC)
      delete net;
   if(CheckPointer(trade) == POINTER_DYNAMIC)
      delete trade;
   IndicatorRelease(h_RSI);
   IndicatorRelease(h_MACD);
  }
```

Весь алгоритм работы советника мы пропишем в функции *OnTick*. Терминал вызывает эту функцию при наступлении события нового тика на графике с запущенной программой. В начале функции проверяем, открылся ли новый бар. Если свеча уже отработана, выходим из функции в ожидании нового тика. Суть данного действия проста: мы будем подавать на вход нашей модели информацию только закрытых свечей, а чтобы информация была максимально актуальной, совершать мы это будем на открытии новой свечи.

```
void OnTick()
  {
   if(lastbar >= iTime(_Symbol, TimeFrame, 0))
      return;
   lastbar = iTime(_Symbol, TimeFrame, 0);
```

В нашем шаблоне нет функций, отрабатывающих каждый тик, поэтому мы будем выполнять действия только на открытии новой свечи. Если же вы включите в свою программу функции, которые должны будут обрабатывать каждый тик, такие как трейлинг-стоп, перевод ордеров в безубыток или что-то еще, то вызов таких функций нужно будет поместить до проверки события нового бара.

При наступлении события открытия новой свечи загружаем информацию с наших индикаторов в локальные динамические массивы. Обязательно проверяем результат выполнения операций.

```
   double macd_main[], macd_signal[], rsi[];
   if(h_RSI == INVALID_HANDLE || CopyBuffer(h_RSI, 0, 1, BarsToPattern, rsi) <= 0)
     {
      PrintFormat("Error loading indicator %s data", "RSI");
      return;
     }
   if(h_MACD == INVALID_HANDLE || CopyBuffer(h_MACD, MAIN_LINE, 1, BarsToPattern, mac
      CopyBuffer(h_MACD, SIGNAL_LINE, 1, BarsToPattern, macd_signal) <= 0)
     {
      PrintFormat("Error loading indicator %s data", "MACD");
      return;
     }
```

Когда данные индикаторов загружены, мы создаем экземпляр объекта буфера данных для сбора текущего состояния и организуем цикл заполнения буфера данных текущим состоянием индикаторов. Здесь следует организовать точно такую же последовательность значений описания текущего состояния, как мы заполняли в файле обучающей выборки. В противном случае результат работы модели будет непредсказуем.

```
   CBufferType *input_data = new CBufferType();
   if(!input_data)
     {
      PrintFormat("Error creating Input data array: %d", GetLastError());
      return;
     }
   if(!input_data.BufferInit(BarsToPattern, 4, 0))
      return;

   for(int i = 0; i < BarsToPattern; i++)
     {
      if(!input_data.Update(i, 0, (TYPE)rsi[i]))
        {
         PrintFormat("Error adding Input data to array: %d", GetLastError());
         delete input_data;
         return;
        }

      if(!input_data.Update(i, 1, (TYPE)macd_main[i]))
        {
         PrintFormat("Error adding Input data to array: %d", GetLastError());
         delete input_data;
         return;
        }

      if(!input_data.Update(i, 2, (TYPE)macd_signal[i]))
        {
         PrintFormat("Error adding Input data to array: %d", GetLastError());
         delete input_data;
         return;
        }

      if(!input_data.Update(i, 3, (TYPE)(macd_main[i] - macd_signal[i])))
        {
         PrintFormat("Error adding Input data to array: %d", GetLastError());
         delete input_data;
         return;
        }
     }
   if(!input_data.Reshape(1,input_data.Total())
     return;
```

Когда в буфере данных мы полностью собрали описание текущего состояния, переходим к работе над нашей моделью. Вначале проверяем действительность указателя на модель и вызываем метод прямого прохода. После успешного завершения метода прямого прохода получаем его результаты в локальный буфер. Мы не создаем новый экземпляр объекта для буфера результатов — для этих целей мы используем буфер исходных данных.

```
   if(!net)
     {
      delete input_data;
      return;
     }
   if(!net.FeedForward(input_data))
     {
      PrintFormat("Error of Feed Forward: %d", GetLastError());
      delete input_data;
      return;
     }
```

Далее следует блок принятия решения по сигналам от нашей модели. В результате прямого прохода моделью возвращается два числа. Первое число обучалось на определение направления предстоящего движения, а второе определяет расстояние до ближайшего экстремума. Таким образом, для совершения операций мы будем ориентироваться на оба сигнала. Они должны быть одинаково направлены.

Сначала мы проверяем сигнал на покупку. Прежде всего параметр, отвечающий за направление движения, должен быть положительным. Сразу же проверим и открытые позиции. При наличии открытых позиций на покупку мы не открываем новую позицию, а выходим из функции до следующего тика.

Обратите внимание, что мы не проверяем наличие открытой позиции на продажу. В нашей упрощенной версии советника мы доверяем прогнозам нашей модели и ожидаем закрытие всех открываемых позиций по тейк-профиту или стоп-лоссу. Тем самым мы исключили из нашего советника блок сопровождения позиции. Как результат, мы ожидаем возможность одновременного существования двух разнонаправленных позиций, а это возможно только при хеджинговом учете позиций. Поэтому тестирование подобного советника возможно только на соответствующих счетах.

Такой подход позволит оценить результативность прогнозов нашей модели. Но при построении советников для использования на реальных рынках я бы рекомендовал продумать и добавить в советник блок сопровождения позиций.

```
   if(!net.GetResults(input_data))
     {
      PrintFormat("Error of Get Result: %d", GetLastError());
      delete input_data;
      return;
     }
   if(input_data.At(0) > 0.0)
     {
      bool opened = false;
      for(int i = 0; i < PositionsTotal(); i++)
        {
         if(PositionGetSymbol(i) != _Symbol)
            continue;
         if(PositionGetInteger(POSITION_TYPE) == POSITION_TYPE_BUY)
            opened = true;
        }

      if(opened)
        {
         delete input_data;
         return;
        }
```

Если нет открытых позиций на покупку, проверяем силу сигнала (вероятность движения в нужном направлении) и ожидаемое движение до предстоящего экстремума. Если хотя бы один из параметров не удовлетворяет требованиям, выходим из функции до следующего тика.

```
      if(input_data.At(0) < TradeLevel ||
         input_data.At(1) < (MinTarget * SymbolInfoDouble(_Symbol, SYMBOL_POINT)))
        {
         delete input_data;
         return;
        }
```

Если все же принято решение об открытии позиции, то мы определяем уровни стоп-лосса и тейк-профита и отправляем ордер на покупку.

```
      double tp = SymbolInfoDouble(_Symbol, SYMBOL_BID) + MathMin(input_data.At(1) *
                  ProfitMultiply, MaxTP * SymbolInfoDouble(_Symbol, SYMBOL_POINT));
      double sl = SymbolInfoDouble(_Symbol, SYMBOL_BID) -
                  StopLoss * SymbolInfoDouble(_Symbol, SYMBOL_POINT);
      trade.Buy(Lot, _Symbol, 0, sl, tp);
     }
```

Аналогично организован алгоритм принятия решения на продажу.

```
   if(input_data.At(0) < 0)
     {
      bool opened = false;
      for(int i = 0; i < PositionsTotal(); i++)
        {
         if(PositionGetSymbol(i) != _Symbol)
            continue;
         if(PositionGetInteger(POSITION_TYPE) == POSITION_TYPE_SELL)
            opened = true;
        }

      if(opened)
        {
         delete input_data;
         return;
        }

      if(input_data.At(0) > -TradeLevel ||
         input_data.At(1) > -(MinTarget * SymbolInfoDouble(_Symbol, SYMBOL_POINT)))
        {
         delete input_data;
         return;
        }

      double tp = SymbolInfoDouble(_Symbol, SYMBOL_BID) + MathMax(input_data.At(1) *
                  ProfitMultiply, -MaxTP * SymbolInfoDouble(_Symbol, SYMBOL_POINT));
      double sl = SymbolInfoDouble(_Symbol, SYMBOL_BID) +
                 StopLoss * SymbolInfoDouble(_Symbol, SYMBOL_POINT);
      trade.Sell(Lot, _Symbol, 0, sl, tp);
     }
    delete input_data;
  }
```

После выполнения всех операций по описанному алгоритму удаляем буфер текущего состояния и выходим из функции.

Советник сделан весьма упрощенным, но и он позволит проверить работу нашей модели в тестере стратегий *MetaTrader 5*.

7.3 Создание модели для тестирования

В предыдущем разделе мы создали шаблон советника, чтобы протестировать возможность использования наших моделей нейронных сетей для совершения торговых операций на финансовых рынках. Это универсальный шаблон, который может работать с любыми моделями. В нем лишь ограничены параметры описания одной свечи и конфигурации слоя результатов. В результате работы модель должна вернуть тензор значений, который блок принятия решений в шаблоне может однозначно трактовать.

Для тестирования я решил собрать новую модель, в которой будет задействовано несколько типов нейронных слоев. Создавать и обучать модель будем с помощью скрипта. Формат таких скриптов нам уже знаком из многочисленных тестов, которые мы рассмотрели в рамках этой

книги. Новый скрипт создадим в файле *gpt_not_norm.mq5*. Файл нового скрипта сохраним в подкаталоге *gpt* нашей книги в соответствии со структурой файлов.

В глобальной части скрипта объявим две константы:

- *BarsInHistory* — количество баров в обучающей выборке;
- *ModelName* — имя файла для сохранения обученной модели.

Обозначим внешние параметры скрипта. Прежде всего, это, конечно, название файла с обучающей выборкой *StudyFileName*. Хочу заметить, что здесь мы используем выборку без предварительной нормализации данных. В предыдущем разделе в нашем шаблоне советника мы не настраивали предварительную обработку данных, поэтому весь расчет на использование слоя пакетной нормализации. Тесты, которые мы провели ранее, подтверждают возможность такой замены.

Следующий внешний параметр *OutputFileName* содержит имя файла для записи динамики изменения ошибки модели в процессе обучения.

Мы планируем использовать блок с архитектурой *GPT*. Для такой архитектуры характерно использование параметра для указания длины накопительной внутренней последовательности паттерна. Для запроса данного параметра у пользователя создадим внешний параметр *BarsToLine.*

Далее идет ставший уже стандартным для подобных скриптов набор параметров:

- *NeuronsToBar* — количество нейронов входного слоя на один бар;
- *UseOpenCL* — флаг использования технологии *OpenCL*;
- *BatchSize* — размер пакета между обновлениями матрицы весов;
- *LearningRate* — коэффициент обучения;
- *HiddenLayers* — количество скрытых слоев;
- *HiddenLayer* — количество нейронов в скрытом слое;
- *Epochs* — количество итераций обновления матрицы весов до прекращения процесса обучения.

```
#define HistoryBars           40
#define ModelName             "gpt_not_norm.net"
//+------------------------------------------------------------------+
//| Внешние параметры для работы скрипта                             |
//+------------------------------------------------------------------+
// Имя файла с обучающей выборкой
input string   StudyFileName  = "study_data_not_norm.csv";
// Имя файла для записи динамики ошибки
input string   OutputFileName = "loss_study_gpt_not_norm.csv";
// Глубина анализируемой истории
input int      BarsToLine     = 60;
```

```
// Количество нейронов входного слоя на 1 бар
input int      NeuronsToBar   = 4;
// Использовать OpenCL
input bool     UseOpenCL      = false;
// Размер пакета для обновления матрицы весов
input int      BatchSize      = 10000;
// Коэффициент обучения
input double   LearningRate   = 0.0003;
// Количество скрытых слоев
input int      HiddenLayers   = 2;
// Количество нейронов в одном скрытом слое
input int      HiddenLayer    = 60;
// Количество итераций обновления матрицы весов
input int      Epochs         = 5000;
```

После объявления внешних параметров добавляем в скрипт нашу библиотеку работы с моделями нейронных сетей.

```
//+------------------------------------------------------------------+
//| Подключаем библиотеку нейронной сети                             |
//+------------------------------------------------------------------+
#include "..\..\..\Include\NeuroNetworksBook\realization\neuronnet.mqh"
```

На этом работа в глобальной области заканчивается. Продолжим написание кода скрипта уже в теле функции *OnStart*. В теле функции мы используем структурированный подход вызова отдельных функций, каждая из которых выполняет определенные действия.

```
void OnStart(void)
  {
   VECTOR loss_history;
//--- подготовим вектор для хранения истории ошибок сети
   if(!loss_history.Resize(0, Epochs))
     {
      Print("Not enough memory for loss history");
      return;
     }
   CNet net;
//--- 1. инициализация сети
   if(!NetworkInitialize(net))
      return;
//--- 2. загрузка данных обучающей выборки
   CArrayObj data;
   CArrayObj result;
   if(!LoadTrainingData(StudyFileName, data, result))
      return;
//--- 3. обучение сети
   if(!NetworkFit(net, data, result, loss_history))
      return;
//--- 4. сохранение истории ошибок сети
   SaveLossHistory(OutputFileName, loss_history);
   Print("Done");
  }
```

Первой в нашем скрипте идет функция инициализации модели *NetworkInitialize*. В параметрах данная функция получает указатель на объект модели, которую предстоит инициализировать.

В теле функции предусмотрено два варианта инициализации модели. Сначала мы пытаемся загрузить предварительно обученную модель из файла, указанного во внешних параметрах скрипта, и проверяем результат выполнения операции. Если модель успешно загружена, пропускаем блок создания новой модели и продолжаем работу с загруженной моделью. Благодаря этой возможности мы можем при необходимости остановить и возобновить процесс обучения.

```
bool NetworkInitialize(CNet &net)
  {
   if(net.Load(ModelName))
     {
      printf("Loaded pre-trained model %s", ModelName);
      net.SetLearningRates((TYPE)LearningRate,(TYPE)0.9, (TYPE)0.999);
      net.UseOpenCL(UseOpenCL);
      net.LossSmoothFactor(BatchSize);
      return true;
     }
```

Если же не удалось загрузить предварительно обученную модель, создаем новую нейронную сеть. Сначала мы создаем динамический массив для записи указателей на объекты описания нейронных слоев и тут же вызываем функцию создания описания архитектуры нашей модели *CreateLayersDesc*.

```
   CArrayObj layers;
//--- создаем описание слоев сети
   if(!CreateLayersDesc(layers))
      return false;
```

Когда в нашем динамическом массиве объектов собрано полное описание создаваемой модели, вызываем метод генерации модели с указанием в параметрах указателя на динамический массив описания модели, функции потерь и параметров оптимизации модели.

```
//--- инициализируем сеть
   if(!net.Create(&layers, (TYPE)LearningRate, (TYPE)0.9, (TYPE)0.999, LOSS_MSE,
                                       0, (TYPE)0))
     {
      PrintFormat("Error of init Net: %d", GetLastError());
      return false;
     }
```

Обязательно проверяем результат выполнения операции.

После создания модели устанавливаем заданный пользователем флаг использования технологии *OpenCL* и диапазон сглаживания ошибки.

```
   net.UseOpenCL(UseOpenCL);
   net.LossSmoothFactor(BatchSize);
   return true;
  }
```

На этом мы заканчиваем функцию инициализации модели. Предлагаю сразу познакомиться с алгоритмом функции создания описания архитектуры модели *CreateLayersDesc*. В параметрах функция получает указатель на объект динамического массива описания архитектуры модели. В теле функции сразу очищаем полученный массив.

```
bool CreateLayersDesc(CArrayObj &layers)
  {
   layers.Clear();
```

Первым мы создаем слой исходных данных. Алгоритм для создания всех нейронных слоев будет один, поэтому сначала мы создаем новый экземпляр объекта описания нейронного слоя. Как всегда, проверяем результат операции по созданию нового объекта.

```
   CLayerDescription *descr;
//--- создаем слой исходных данных
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
```

После того, как объект описания нейронного слоя создан, заполняем его данными, достаточными для однозначного понимания архитектуры создаваемого нейронного слоя.

```
   descr.type         = defNeuronBase;
   int prev_count = descr.count = NeuronsToBar * GPT_InputBars;
   descr.window       = 0;
   descr.activation   = AF_NONE;
   descr.optimization = None;
```

На вход создаваемой модели мы будем подавать информацию всего о трех последних свечах. На самом деле этого действительно мало как с токи зрения объема информации для принятия решения нейронной сетью, так и с практической точки зрения трейдинга. Но не будем забывать, что мы будем использовать в своей модели блоки с архитектурой *GPT*. Данная архитектура предполагает накопление исторических данных внутри блока. Это компенсирует недостаток информации. В то же время использование малого объема исходных данных позволит значительно сократить объем вычислительных операций на каждой итерации. Таким образом, размер слоя исходных данных определяется как произведение количества элементов для описания одной свечи на количество анализируемых свечей. В нашем случае количество элементов описания одной свечи задается во внешнем параметре *NeuronsToBar*, а количество анализируемых свечей задается константой *GPT_InputBars*.

Слой исходных данных не использует ни функцию активации, ни оптимизацию параметров. Напомню, что исходные данные мы записываем напрямую в буфер результатов.

```
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
```

После того, как мы заполнили объект описания архитектуры нейронного слоя необходимым набором достаточных данных, добавляем его в наш динамический массив указателей на объекты.

Напомню, что мы предварительно не обрабатывали исходные данные. Поэтому в архитектуре нейронной сети мы предусмотрели создание слоя пакетной нормализации данных сразу за слоем исходных данных. Согласно приведенному выше алгоритму, мы создаем новый экземпляр объекта описания нейронного слоя. Не забываем проверить результат проведения операции создания объекта, так как на следующем этапе мы будем заполнять элементы данного объекта необходимым описанием архитектуры создаваемого нейронного слоя, а обращение к элементам объекта по недействительному указателю вызовет критическую ошибку.

```
//--- создаем слой нормализации данных
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
```

В описании создаваемого нейронного слоя мы укажем тип нейронного слоя *defNeuronBatchNorm*, что соответствует слою пакетной нормализации. Размер нейронного слоя и окна исходных данных укажем равным размеру предыдущего нейронного слоя исходных данных.

Размер пакета укажем на уровне размера пакета между обновлениями матрицы весов, которые пользователь указал во внешнем параметре *BatchSize.*

Как и предыдущий слой, слой пакетной нормализации не использует функцию активации. Но уже появляется метод оптимизации обучаемых параметров *Adam*.

```
descr.type         = defNeuronBatchNorm;
descr.count        = prev_count;
descr.window       = prev_count;
descr.activation   = AF_NONE;
descr.optimization = Adam;
descr.batch        = BatchSize;
```

После указания всех необходимых параметров описания создаваемого нейронного слоя мы добавляем указатель на объект в динамический массив указателей на описание архитектуры нашей модели.

```
if(!layers.Add(descr))
  {
   PrintFormat("Error adding layer: %d", GetLastError());
   delete descr;
   return false;
  }
```

Мы уже говорили, что четырех параметров в описании одной свечи довольно мало. Следовательно, было бы не плохо добавить еще несколько параметров. Для использования методов машинного обучения в условиях дефицита параметров разработан ряд подходов, которые были объединены в целое направление *Feature Engineer*. Одним из таких способов является использование сверточных слоев, в которых количество фильтров превышает размер исходного окна. Логика такого подхода заключается в том, что вектор описания одного элемента рассматривается как координаты некой точки текущего состояния в *N*-мерном пространстве, где *N* — длина вектора описания одного элемента. Осуществляя свертку, мы проецируем эту точку на вектор свертки. Именно этим свойством мы пользуемся при сжатии данных и понижении размерности данных. Именно этим свойством мы и воспользуемся для увеличения размерности данных. Как можно заметить, здесь нет противоречия с ранее изученным подходом к использованию сверточного слоя. Только теперь мы берем количество фильтров превышающее вектор описания одного элемента и тем самым увеличиваем размерность пространства. Воспользуемся описанным методом и сделаем следующий сверточный слой с числом фильтров в два раза больше числа элементов описания одной свечи. Надо сказать, что в данном случае мы делаем сверточный слой в рамках описания одной свечи, поэтому размер окна исходных данных и размер его шага будут равны размеру вектора описания одной свечи.

Алгоритм создания описания нейронного слоя остается прежним. Сначала мы создаем новый экземпляр объекта описания нейронного слоя и проверяем результат выполнения операции.

```
//--- Сверточный слой
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
```

Затем заполняем необходимые данные.

```
descr.type = defNeuronConv;
prev_count = descr.count = prev_count / NeuronsToBar;
descr.window = NeuronsToBar;
int prev_window = descr.window_out = 2 * NeuronsToBar;
descr.step = NeuronsToBar;
descr.activation = AF_SWISH;
descr.optimization = Adam;
descr.activation_params[0] = 1;
```

Предаем указатель на заполненный экземпляр объекта в динамический массив описания архитектуры создаваемой модели.

```
if(!layers.Add(descr))
  {
   PrintFormat("Error adding layer: %d", GetLastError());
   delete descr;
   return false;
  }
```

После того, как поток информации пройдет через сверточный слой, мы ожидаем получить тензор с восемью элементами описания состояния одной свечи. Но мы помним, что полносвязные модели не оценивают зависимость между элементами. А при анализе временных рядов такая зависимость обычно довольно сильная.

Поэтому на следующем этапе мы попробуем проанализировать такие зависимости. Как вы помните, именно о таком анализе мы говорили при знакомстве со сверточными сетями. Каким бы странным это ни казалось, но для решения двух, на первый взгляд, разных задач мы используем один тип нейронных слоев. На самом деле, мы выполняем схожую задачу, но с разными данными. В предыдущем сверточном слое мы раскладывали вектор описания одной свечи на большее количество элементов. Можно посмотреть на эту задачу и с другой стороны. Как мы говорили при изучении сверточного слоя, процесс свертки — это определение схожести двух функций. То есть в каждом фильтре мы выявляем схожесть исходных данных с некой эталонной функцией. Каждый фильтр использует свою эталонную функцию. Осуществляя операции свертки в размере одного бара, мы искали схожесть каждого бара с неким эталонным.

Теперь же мы хотим проанализировать динамику изменения параметров свечей. Для этого нам надо провести свертку между одинаковыми элементами векторов описания разных свечей. После проведения свертки предыдущий слой нам вернул последовательно по три значения (количество анализируемых свечей) от каждого фильтра. Значит, следующим шагом мы создаем сверточный слой с окном исходных данных и его шагом равным количеству анализируемых свечей. В данном сверточном слое мы тоже будем использовать восемь фильтров.

Создадим описание сверточного нейронного слоя согласно приведенному ранее алгоритму.

```
//--- Сверточный слой 2
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }

   descr.type = defNeuronConv;
   descr.window = prev_count;
   descr.step = prev_count;
   prev_count = descr.count = prev_window;
   prev_window = descr.window_out = 8;
   descr.activation = AF_SWISH;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;

   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
```

Таким образом, после предварительной обработки данных в одном слое пакетной нормализации и двух последовательных сверточных слоях мы получили тензор из 64 элементов (8 * 8). Напомню, что на вход нейронной сети мы подали тензор из 12 элементов: 3 свечи по 4 элемента на каждую свечу.

Дальше сигнал будем обрабатывать в блоке с архитектурой *GPT*. В нем мы создадим четыре последовательных нейронных слоя с восемью головами внимания в каждом. Размер глубины анализируемых данных мы вывели во внешние параметры скрипта, что позволит провести обучение с различной глубиной и выбрать оптимальный параметр по соотношению затрат на обучение к качеству работы модели. Алгоритм создания описания нейронного слоя остается прежним.

```
//--- GPT-слой
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }

   descr.type = defNeuronGPT;
   descr.count = BarsToLine;
   descr.window = prev_count * prev_window;
   descr.window_out = prev_window;
   descr.step = 8;
   descr.layers = 4;
   descr.activation = AF_NONE;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;

   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
```

За блоком *GPT* создадим блок полносвязных нейронных слоев. Все слои в блоке будут идентичными. Количество слоев и нейронов в каждом мы вынесли во внешние параметры скрипта. Согласно предложенному выше алгоритму, мы создаем новый экземпляр объекта описания нейронного слоя и проверяем результат выполнения операции.

```
//--- Скрытые полносвязные слои
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
```

После успешного создания нового экземпляра объекта заполним его необходимыми данными. Как уже было сказано выше, количество нейронов в слое мы возьмем из внешнего параметра *HiddenLayer*. Функцию активации я выбрал *Swish*. Конечно, для большей гибкости скрипта в процессе обучения больше параметров можно вынести во внешние и провести больше циклов обучения модели с различными параметрами. Такой подход потребует больше времени и затрат на обучение модели, но позволит найти наиболее оптимальные значения для параметров модели.

```
   descr.type = defNeuronBase;
   descr.count = HiddenLayer;
   descr.activation = AF_SWISH;
   descr.optimization = Adam;
   descr.activation_params[0] = 1;
```

Так как у нас планируется создание идентичных нейронных слоев, далее мы создаем цикл с количеством итераций равным количеству создаваемых нейронных слоев. В теле цикла будем добавлять созданное описание нейронного слоя в динамический массив описания архитектуры

создаваемой модели. И, конечно, проверяем результат выполнения операций на каждой итерации цикла.

```
for(int i = 0; i < HiddenLayers; i++)
  {
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
  }
```

В завершение модели создадим слой результатов. Это полносвязный слой, который содержит два нейрона с функцией активации *tanh*. Выбор в пользу данной функции активации был сделан по совокупности оценки целевых значений обучаемой модели:

- Первый элемент целевого значения принимает 1 для целей на покупку и −1 для целей на продажу, что наилучшим образом настраивается функцией гиперболического тангенса *tanh;*
- Обучение моделей мы осуществляли на паре *EURUSD*, следовательно, значение ожидаемого движения до ближайшего экстремума должно быть в диапазоне от −0.05 до 0.05. В данном диапазоне значений график функции гиперболического тангенса *tanh* близок к линейному.

Если же вы планируете использование модели на инструментах с абсолютным значением ожидаемого движения до ближайшего экстремума более 1, можно применить масштабирование целевого результата. Затем использовать обратное масштабирование при трактовке сигнала модели. Также можно подумать об использовании другой функции активации.

Для создания описания нейронного слоя в архитектуре создаваемой модели мы используем тот же алгоритм. Сначала создаем новый экземпляр объекта описания нейронного слоя и проверяем результат выполнения операции.

```
//---  Слой результатов
   if(!(descr = new CLayerDescription()))
     {
      PrintFormat("Error creating CLayerDescription: %d", GetLastError());
      return false;
     }
```

Затем наполняем созданный объект необходимой информацией: тип нейронного слоя, количество нейронов, функция активации и метод оптимизации параметров модели.

```
descr.type         = defNeuronBase;
descr.count        = 2;
descr.activation   = AF_TANH;
descr.optimization = Adam;
```

Указатель на заполненный объект добавляем в динамический массив описания архитектуры создаваемой модели. При этом не забываем проверить результат выполнения операции.

```
   if(!layers.Add(descr))
     {
      PrintFormat("Error adding layer: %d", GetLastError());
      delete descr;
      return false;
     }
   return true;
  }
```

Завершаем работу функции.

Следующей в алгоритме скрипта идет функция загрузки обучающей выборки *LoadTrainingData*. В параметрах функция получает строковую переменную с названием файла для загрузки и указатели на два объекта динамических массивов: паттернов *data* и целевых значений *result*.

```
bool LoadTrainingData(string path, CArrayObj &data, CArrayObj &result)
  {
```

Напомню, что загружать мы будем обучающую выборку без предварительной нормализации исходных данных из файла *study_data_not_norm.csv*, так как мы планируем использовать модель в режиме реального времени, а для подготовки исходных данных будем использовать слой пакетной нормализации.

Алгоритм загрузки исходных данных будет полностью повторять тот, что мы рассмотрели для выполнения этой же задачи ранее, в скрипте тестирования архитектуры *GPT*. Но повторение — мать учения. Поэтому кратко освежим его в памяти. Для загрузки обучающей выборки мы объявляем две новые переменные для хранения указателей на объекты буферов данных, в которые мы и будем непосредственно считывать из файла по одному паттерну и его целевому значению (*pattern* и *target*, соответственно). Сами же экземпляры объектов мы будем создавать позже. Это связано с тем, что нам потребуются новые экземпляры объектов для загрузки каждого паттерна. Поэтому создавать объекты будем уже в теле цикла перед непосредственным процессом загрузки данных из файла.

```
   CBufferType *pattern;
   CBufferType *target;
```

После завершения подготовительной работы открываем файл с обучающей выборкой для чтения данных. При открытии файла среди прочих флагов указываем *FILE_SHARE_READ* — этот флаг открывает общий доступ к файлу для чтения данных. То есть, добавляя данный флаг, мы не блокируем доступ к файлу из других приложений для чтения файла. Это позволит запустить параллельно несколько скриптов с разными параметрами, и они не будут блокировать доступ к файлу друг другу. Конечно, запускать параллельно несколько скриптов можно только в том случае, если это позволяют мощности оборудования.

```
//--- открываем файл с обучающей выборкой
   int handle = FileOpen(path, FILE_READ | FILE_CSV | FILE_ANSI | FILE_SHARE_READ,
                                                                  ",", CP_UTF8);
   if(handle == INVALID_HANDLE)
     {
      PrintFormat("Error opening study data file: %d", GetLastError());
      return false;
     }
```

Конечно, проверяем результат выполнения операции. В случае ошибки открытия файла информируем об ошибке пользователя, удаляем все ранее созданные объекты и выходим из программы.

После успешного открытия файла создаем цикл непосредственного чтения данных из файла. Операции в теле цикла будут повторяться до наступления одного из следующих событий:

- достижение конца файла;
- прерывание выполнения программы пользователем.

```
//--- выводим прогресс загрузки данных обучения в комментарий чарта
   uint next_comment_time = 0;
   uint OutputTimeout     = 250; // не чаще 1 раза в 250 миллисекунд
//--- организовываем цикл загрузки обучающей выборки
   while(!FileIsEnding(handle) && !IsStopped())
     {
      if(!(pattern = new CBufferType()))
        {
         PrintFormat("Error creating Pattern data array: %d", GetLastError());
         return false;
        }
      if(!pattern.BufferInit(1, NeuronsToBar * GPT_InputBars))
         return false;
```

В теле цикла первым делом создаем новые экземпляры объектов для записи текущего паттерна и его целевых значений. И, конечно, сразу проверяем результат выполнения операции. В случае возникновения ошибки информируем пользователя об ошибке, удаляем ранее созданные объекты, закрываем файл и выходим из программы. Очень важно удалить все объекты и закрыть файл перед выходом из программы.

```
      if(!(target = new CBufferType()))
        {
         PrintFormat("Error creating Pattern Target array: %d", GetLastError());
         return false;
        }
      if(!target.BufferInit(1, 2))
         return false;
```

После этого организуем вложенный цикл с количеством итераций равным полному паттерну данных. Мы создавали обучающие выборки из 40 свечей на паттерн и, соответственно, должны последовательно считать все данные. В то же время для обучения нашей модели не требуется столь большое описание паттерна. Поэтому организуем пропуск лишних данных — в буфер будем записывать только последние необходимые данные.

```
      int skip = (HistoryBars - GPT_InputBars) * NeuronsToBar;
      for(int i = 0; i < NeuronsToBar * HistoryBars; i++)
        {
         TYPE temp = (TYPE)FileReadNumber(handle);
         if(i < skip)
            continue;
         pattern.m_mMatrix[0, i - skip] = temp;
        }
```

После полной загрузки данных текущего паттерна организуем аналогичный цикл для загрузки целевых значений. На этот раз число итераций цикла будет равно числу целевых значений в обучающей выборке, то есть в нашем случае двум. Перед запуском цикла проверяем состояние флага сохранения паттерна. В цикл заходим только в том случае, если описание паттерна было загружено в полном объеме.

```
      for(int i = 0; i < 2; i++)
         target.m_mMatrix[0, i] = (TYPE)FileReadNumber(handle);
```

После отработки циклов загрузки данных переходим к блоку добавления информации о паттерне в наши динамические массивы. Добавим указатели на объекты в динамический массив описания паттернов и целевых результатов. Не забываем проверять результат выполнения операций.

```
      if(!data.Add(pattern))
        {
         PrintFormat("Error adding study data to array: %d", GetLastError());
         return false;
        }
      if(!result.Add(target))
        {
         PrintFormat("Error adding study data to array: %d", GetLastError());
         return false;
        }
```

После успешного добавления в динамические массивы проинформируем пользователя о количестве загруженных паттернов и перейдем к загрузке следующего паттерна.

```
      //--- выводим прогресс загрузки в комментарий чарта (не чаще 1 раза в 250 милли
      if(next_comment_time < GetTickCount())
        {
         Comment(StringFormat("Patterns loaded: %d", data.Total()));
         next_comment_time = GetTickCount() + OutputTimeout;
        }
     }
   FileClose(handle);
   Comment(StringFormat("Patterns loaded: %d", data.Total()));
   return(true);
  }
```

После успешной загрузки всех данных обучающей выборки закрываем файл и завершаем функцию загрузки данных.

Далее переходим к процедуре обучения нашей модели в функции *NetworkFit*. В параметрах функция получает указатели на три объекта:

- обучаемая модель,

- динамический массив описаний состояния системы,
- динамический массив целевых результатов.

```
bool NetworkFit(CNet &net, const CArrayObj &data, const CArrayObj &result, VECTOR &lo
  {
```

В теле метода вначале выполним небольшую подготовительную работу. Подготовим локальные переменные.

```
   int patterns = data.Total();
   int count = -1;
   TYPE min_loss = FLT_MAX;
```

После завершения подготовительной работы организуем вложенные циклы обучения нашей модели. Внешний цикл будет отсчитывать количество обновлений матриц весовых коэффициентов, а во вложенном мы будем перебирать паттерны нашей обучающей выборки.

Напомню, в архитектуре *GPT* исторические данные накапливаются в стеке. Поэтому для корректной работы модели очень важна историческая последовательность подачи данных на вход модели, аналогично рекуррентным моделям. По этой причине мы не можем перемешивать обучающую выборку в рамках одного пакета обучения и будем подавать на вход модели паттерны в хронологическом порядке. В то же время для обучения модели модно использовать случайные пакеты из общей совокупности обучающей выборки. Стоит отметить, что при определении пакета для обучения нужно увеличить его размер на величину внутренней накопительной последовательности блока *GPT*, так как для корректного определения зависимостей между элементами необходимо соблюдение их хронологической последовательности.

Таким образом, перед запуском вложенного цикла мы определяем границы текущего пакета для обучения.

В теле вложенного цикла перед дальнейшем продолжением операций проверяем флаг принудительного завершения работы программы и при необходимости прерываем работу функции.

```
//--- цикл по эпохам
   for(int epoch = 0; epoch < Epochs; epoch++)
     {
      ulong ticks = GetTickCount64();
      //--- обучаем батчами
      //--- выбор случайного паттерна
      int k = (int)((double)(MathRand() * MathRand()) / MathPow(32767.0, 2) * (patter
      k = fmax(k, 0);
      for(int i = 0; (i < (BatchSize + BarsToLine) && (k + i) < patterns); i++)
        {
         //--- проверим на остановку обучения
         if(IsStopped())
           {
            Print("Network fitting stopped by user");
            return true;
           }
```

Сначала осуществляем прямой проход модели, вызвав метод *net.FeedForward*. В параметрах метода прямого прохода мы передаем указатель на объект описания текущего состояния паттерна и проверяем результат выполнения операции. В случае возникновения ошибки выполнения

метода информируем пользователя об ошибке, удаляем созданные объекты и выходим из программы.

```
      if(!net.FeedForward(data.At(k + i)))
        {
         PrintFormat("Error in FeedForward: %d", GetLastError());
         return false;
        }
```

После успешного выполнения метода прямого прохода проверяем заполненность буфера нашего блока *GPT*. Если буфер еще не заполнен, переходим к следующей итерации цикла.

```
      if(i < BarsToLine)
         continue;
```

Метод обратного прохода *net.Backpropagation* вызываем только после заполнения накопительной последовательности блока *GPT*. На этот раз в параметрах метода мы передаем указатель на объект целевых значений. Обязательно проверяем результат операции — если возникла ошибка, выполняем операции как при ошибке в прямом методе.

```
      if(!net.Backpropagation(result.At(k + i)))
        {
         PrintFormat("Error in Backpropagation: %d", GetLastError());
         return false;
        }
     }
```

Методами прямого и обратного проходов мы выполнили соответствующие алгоритмы процесса обучения нашей модели. На данном этапе градиент ошибки уже распространен до каждого обучаемого параметра. Осталось лишь обновить матрицы весовых коэффициентов. Но данную операцию мы выполняем не на каждой итерации обучения, а только после набора пакета. В нашем случае обновлять матрицы весовых коэффициентов будем после завершения итераций вложенного цикла.

```
      //--- перенастраиваем веса сети
      net.UpdateWeights(BatchSize);
      printf("Use OpenCL %s, epoch %d, time %.5f sec", (string)UseOpenCL, epoch, (Get
```

Как вы видели на графиках тестирования моделей, динамика ошибки модели практически никогда не идет плавной линией. Сохранение модели с минимальной ошибкой позволит нам сохранить наиболее подходящие параметры для нашей модели. Поэтому мы сначала проверяем текущую ошибку модели и сравниваем ее с минимальной достигнутой в процессе обучения. Если ошибка снизилась, сохраняем текущую модель и обновляем переменную минимальной ошибки.

```
      //--- сообщим о прошедшей эпохе
      TYPE loss = net.GetRecentAverageLoss();
      Comment(StringFormat("Epoch %d, error %.5f", epoch, loss));
      //--- запомним ошибку эпохи для сохранения в файл
      loss_history[epoch] = loss;
      if(loss < min_loss)
         //--- сохраняем модель с минимальной ошибкой
         if(net.Save(ModelName))
           {
            min_loss = loss;
            count = -1;
           }
```

Дополнительно мы ввели счетчик *count*. В нем мы будем отсчитывать количество итераций обновления от последнего минимального значения ошибки. И если его значение превысит указанный порог (в примере указано 10 итераций), то мы прерываем процесс обучения.

```
      if(count >= 10)
         break;
      count++;
     }
   return true;
  }
```

После завершения полного цикла обучения нам предстоит сохранить в файл накопленную динамику изменения ошибки модели в процессе обучения. Для этого мы создали функцию *SaveLossHistory*. В параметрах функция получает строковую переменную с именем файла для записи данных и вектор ошибок в процессе обучения модели.

В теле функции мы открываем файл для записи. При этом используем имя файла, которое пользователь указал в параметрах. Сразу проверяем результат. Если произошла ошибка при открытии файла, информируем пользователя и выходим из функции.

```
void SaveLossHistory(string path, const VECTOR &loss_history)
  {
   int handle = FileOpen(OutputFileName, FILE_WRITE | FILE_CSV | FILE_ANSI,
                                                              ",", CP_UTF8);
   if(handle == INVALID_HANDLE)
     {
      PrintFormat("Error creating loss file: %d", GetLastError());
      return;
     }
   for(ulong i = 0; i < loss_history.Size(); i++)
      FileWrite(handle, loss_history[i]);
   FileClose(handle);
   PrintFormat("The dynamics of the error change is saved to a file %s\\MQL5\\Files\\
                                  TerminalInfoString(TERMINAL_DATA_PATH), OutputFileNa
  }
```

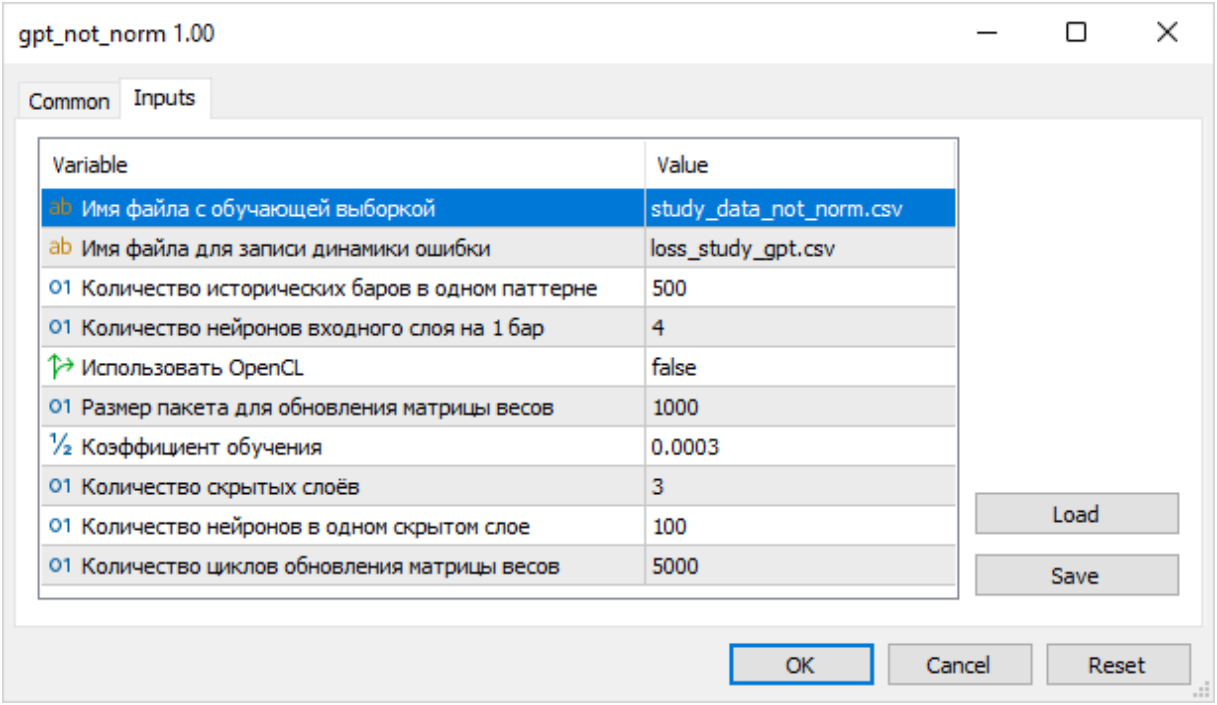
Если файл открылся успешно, организуем цикл, в котором поочередно запишем все значения вектора накопления ошибки модели при обучении. После завершения полного цикла записи данных закрываем файл и информируем пользователя о месте нахождения файла.

На этом наш скрипт создания и обучения модели завершен, и мы можем начать обучение модели на ранее созданной выборке не нормированных обучающих данных из файла *study_data_not_norm.csv*.

Следующим шагом будет запуск процесса обучения модели. Здесь надо запастись терпением, ведь процесс обучения довольно длительный. Его длительность зависит от используемого оборудования. К примеру, я запустил обучение модели с параметрами, приведенными на скриншоте ниже.

На моем ноутбуке с Intel Core i7-1165G7 вычисление одного пакета между обновлениями матриц весовых коэффициентов составляет 35–36 секунд. То есть полное обучение модели с 5000 итераций обновления матриц коэффициентов займет около 2 суток непрерывной работы. В то же время, если вы видите что обучение остановилось и минимальная ошибка долгое время не изменяется, вы можете в ручном режиме прервать обучение модели. Если достигнутые показатели не удовлетворяют требования, можно продолжить обучение модели с другими значениями коэффициента обучения и размера пакета для обновления матрицы весов. Общий подход в подборе параметров следующий:

- Коэффициент обучения — обучение начинается с большего коэффициента, и в процессе обучения мы постепенно снижаем коэффициент обучения;
- Размера пакета обновления матрицы весов — обучение начинается с малого пакета и постепенно увеличивается.



Параметры обучения модели

Указанные приемы позволяют на начальном этапе проводить максимально быстрое грубое обучение модели с последующей более тонкой настройкой. При этом, если в процессе обучения модели ошибка постоянно растет, то это свидетельствует о завышенном коэффициенте обучения. Использование большого пакета обновления матрицы весов помогает изменять матрицы весов в наиболее приоритетном направлении, но при этом требует больше времени на проведение операций между обновлениями весовых коэффициентов. В то же время малый пакет ведет к более быстрому и более хаотичному обновлению параметров модели, при этом общая тенденция сохраняется. Но при использовании малых размеров пакета рекомендуется уменьшать

и коэффициент обучения, чтобы снизить переобучение модели под конкретные участки обучающей выборки.

7.4 Определение параметров советника

После того, как мы обучили модель, перед ее применением в торговой стратегии, необходимо научиться ее использовать. Прежде всего, нужно научиться понимать ее сигналы. Как вы знаете, после осуществления прямого прохода на выходе наша модель возвращает два значения:

- Вероятное направление движения (абсолютное значение показывает вероятность движения, а знак — направление);
- Ожидаемую силу движения (абсолютное значение показывает силу движения, а знак — направление).

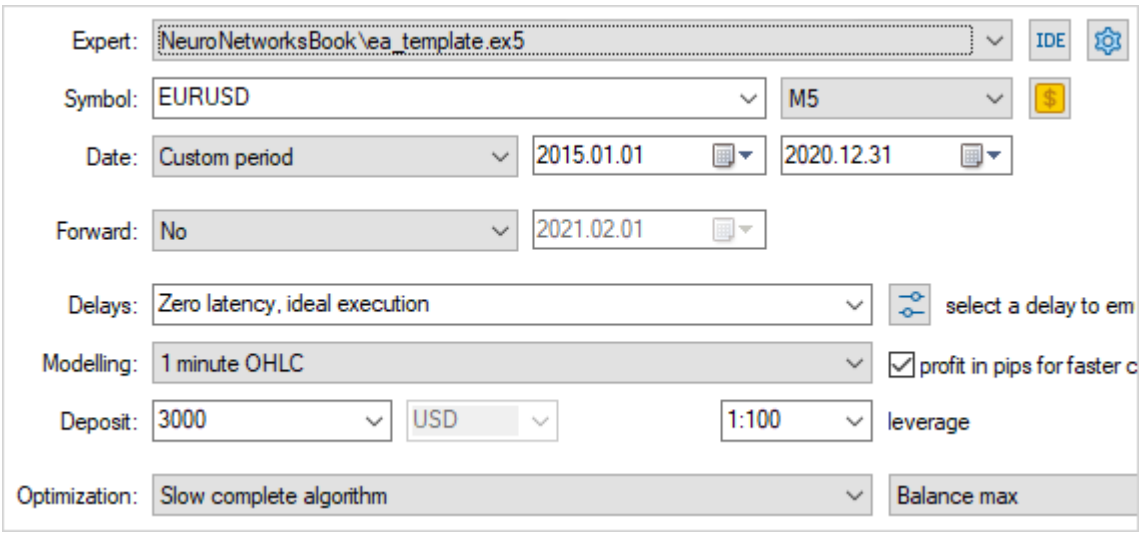
Для каждого параметра нужно найти пороговое значение принятия решения. Слишком большое значение может отсечь большое количество прибыльных сделок или вовсе не дать ни одного сигнала на совершение торговых операций. Слишком маленькое значение может привести к большому количеству ложных сигналов и вовсе сделать торговлю убыточной. Поэтому сейчас очень важно найти оптимальные параметры советника для работы с нашей обученной моделью.

Для выполнения этой работы нам идеально подходит тестер стратегий *MetaTrader 5*. В терминале нажимаем клавиши *Ctrl+R* и переходим в тестер. На вкладке *Settings* в поле *Expert* выбираем свой советник и выставляем параметры тестирования.

Мы обучали модель на исторических данных *EURUSD* за 2015–2020 годы на таймфрейме M5. На этих же исторических данных мы и будем определять оптимальные параметры советника. В общем правиле обучения нейронных сетей проверка работоспособности модели проверяется на валидационной выборке. Но в данном случае мы просто определяем оптимальные параметры советника при работе модели на обучающей выборке.

Мы знаем, что наш советник анализирует точки входа только на открытии свечи, поэтом для проверки наличия сигналов от модели можно было бы провести тестирование только по ценам открытия. Но нам также надо понимать качество этих сигналов. При этом хочется провести первый грубый подбор параметров с минимальными затратами времени и ресурсов. Поэтому выберем режим тестирования по контрольным точкам таймфрейма M1.

Нам нужно оптимизировать параметры, поэтому выберем режим оптимизации. Для выбора режима оптимизации желательно понимать количество предстоящих итераций. Их подсчет не требует усилий, так как они автоматически считаются при выборе параметров советника для оптимизации. Перейдем на вкладку параметров и в столбце *Value* установим начальные значения параметров. Хочу обратить внимание, что такие параметры, как имя файла модели и количество свечей текущего паттерна, не оптимизируются, поскольку они должны соответствовать используемой модели.



Оптимизация параметров советника

На первом этапе мы будем грубо оптимизировать только один параметр — пороговое значение принятия решения *TradeLevel*. Отметим чек-бокс этого параметра. На выходе нашей модели мы использовали гиперболический тангенс в качестве функции активации. Поэтому значения на выходе нейронов нормализованы в диапазоне от −1 до 1. Знак показывает направление движения. Значит, уровень принятия решения может быть в диапазоне от 0 до 1. Конечно, осуществление сделок с вероятностью получения прибыли менее 50% выглядит по меньшей мере рискованно. Поэтому попробуем выбрать уровень принятия решения в диапазоне от 0,5 до 1,0. Напомню, что это первая и грубая подборка параметра, поэтому мы будем использовать шаг 0,05. Тестер стратегий нам сразу посчитал 11 итераций. Как видите, их довольно мало. Вернемся на вкладку *Settings* и выберем тип оптимизации *Slow complete algorithm* (медленная, перебор всех параметров). Также выбираем оптимизацию по максимальному балансу и запускаем процесс оптимизации, нажав на кнопку *Start*.

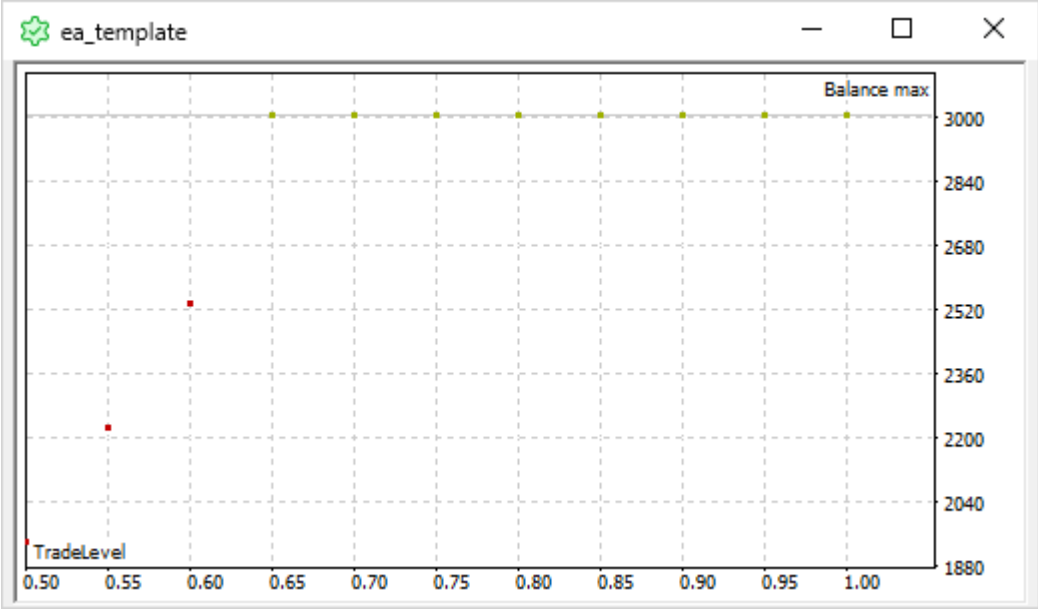
Variable	Value	Start	Step	Stop	Steps
☑ Model	gpt_not_norm.net				
☑ BarsToPattern	3				
☑ Common	true				
☐ TimeFrame	5 Minutes	current		1 Month	
☑ TradeLevel	**1**	**0.5**	**0.05**	**1**	**11**
☐ Lot	0.01	0.01	0.001	0.1	
☐ MaxTP	1000	200	100	1000	
☐ ProfitMultiply	0.9	0.5	0.05	1	
☐ MinTarget	100	50	50	500	
☐ StopLoss	100	50	50	500	
					11

Оптимизация параметров советника

На скриншоте ниже показаны результаты оптимизации. Как можно заметить, при уровне принятия решения 0,65 и выше советник не совершает ни одной сделки. Из этого можно сделать вывод, что в процессе обучения наша нейронная сеть не обнаружила паттернов с вероятностью однонаправленного движения равной 65% или более. Не следует пугаться убыточности работы советника на данном этапе, ведь мы провели только первичную грубую оптимизацию с грубым определением уровня принятия решения. Далее нам предстоит оптимизировать еще несколько параметров нашего советника.

Pass	Res...	Profit	Total trad...	Profit fac...	Expected...	Drawdo...	TradeLevel
10	3000.00	0.00	0		0.00	0.00	1.00
9	3000.00	0.00	0		0.00	0.00	0.95
8	3000.00	0.00	0		0.00	0.00	0.90
7	3000.00	0.00	0		0.00	0.00	0.85
6	3000.00	0.00	0		0.00	0.00	0.80
5	3000.00	0.00	0		0.00	0.00	0.75
4	3000.00	0.00	0		0.00	0.00	0.70
3	3000.00	0.00	0		0.00	0.00	0.65
2	2532.23	-467.77	5198	0.88	-0.09	16.38	0.60
1	2222.02	-777.98	9101	0.88	-0.09	26.60	0.55
0	1935.31	-1064.69	12399	0.87	-0.09	35.79	0.50

Результаты первой оптимизации



Результаты первой оптимизации

Сначала попробуем оптимизировать параметр минимальной силы предстоящего движения *MinTarget*, чтобы отсеять незначительные колебания. Цель такой итерации — отобрать наиболее сильных движений. Это связано с тем, что вероятность срабатывания таких паттернов на практике выше, а незначительные колебания могут не обладать достаточной силой импульса для достижения цели или вовсе не сработать. К тому же, использование ордеров с низким уровнем доходности снижает соотношение доходности к возможному риску.

Оптимизацию параметра мы будем осуществлять в диапазоне от 50 до 600 пунктов с шагом 50 пунктов. На данной итерации нам потребуется проверить 12 прогонов советника.

Variable	Value	Start	Step	Stop	Steps
Model	gpt_not_norm.net				
BarsToPattern	3				
Common	true				
TimeFrame	5 Minutes	current		1 Month	
TradeLevel	0.6	0.5	0.05	1	
Lot	0.01	0.01	0.001	0.1	
MaxTP	1000	200	100	1000	
ProfitMultiply	0.9	0.5	0.05	1	
MinTarget	100	**50**	**50**	**600**	**12**
StopLoss	100	50	50	500	
					12

Подбор порогового значения принятия решения

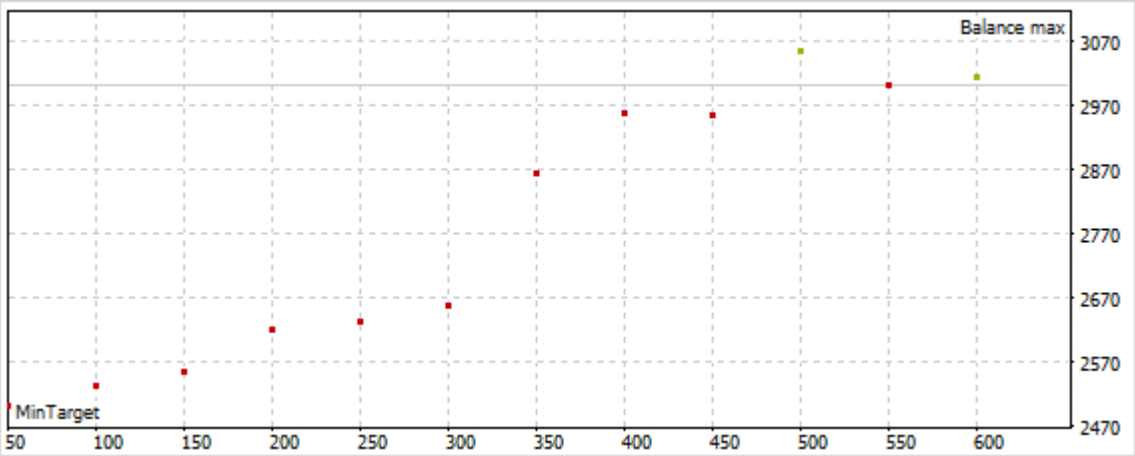
По результатам данной оптимизации параметра мы можем заметить появление первых прибыльных проходов с уровнем принятия решения 500 и 600 пунктов. Однако при таком выборе параметров сильно падает количество совершенных торговых операций. Мы же хотим выжать максимум из нашей модели. Наиболее близкими к точке безубыточности и количеством торговых операций более 1000 можно выделить значения параметра принятия решения на уровне 350–400 пунктов. Позволим себе небольшую авантюру и продолжим оптимизацию параметров с указанным диапазоном параметра.

Pass	Res...	Profit	Total trad...	Profit fac...	Expected...	Drawdo...	MinTarget
9	3052.69	52.69	297	1.22	0.18	0.93	500
11	3010.71	10.71	42	1.31	0.25	0.47	600
10	2999.95	-0.05	123	1.00	-0.00	0.77	550
7	2954.46	-45.54	1188	0.95	-0.04	2.21	400
8	2952.48	-47.52	635	0.91	-0.07	2.45	450
6	2862.93	-137.07	1954	0.91	-0.07	5.16	350
5	2655.56	-344.44	2763	0.84	-0.12	11.93	300
4	2630.25	-369.75	3509	0.86	-0.11	13.22	250
3	2619.80	-380.20	4181	0.88	-0.09	13.52	200
2	2553.66	-446.34	4724	0.87	-0.09	15.68	150
1	2532.23	-467.77	5198	0.88	-0.09	16.38	100
0	2500.37	-499.63	5537	0.87	-0.09	17.54	50

Результаты оптимизации порогового значения принятия решения

Далее перейдем к оптимизации параметра стоп-лосса, который ограничивает риски на каждую сделку. Данный параметр будем оптимизировать в диапазоне от 50 до 500 пунктов с шагом 50 пунктов.

Как было сказано выше, мы не определили четкое значение параметра *MinTarget*. Поэтому для текущего процесса оптимизации будем использовать два оптимизируемых параметра. При этом параметр порогового уровня прогнозной силы предстоящего импульса будет принимать только два допустимых значения.



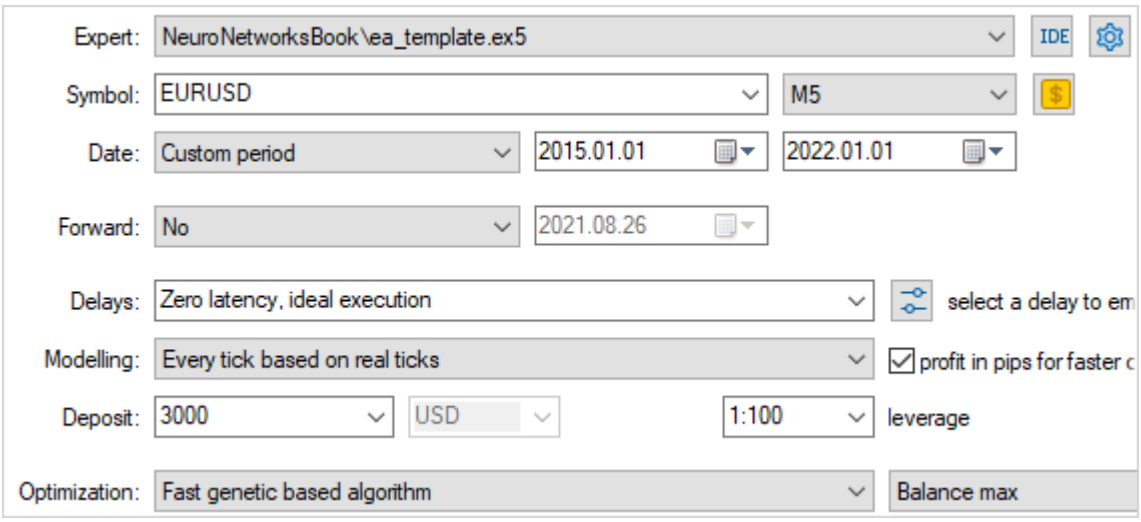
Результаты оптимизации порогового значения принятия решения

Таким образом, тестер стратегий посчитал 20 проходов текущего процесса оптимизации.

Стоит обратить внимание на еще одно обстоятельство. В предстоящем процессе оптимизации мы собираемся подобрать оптимальный уровень стоп-лосса. Здесь надо сказать, что в процессе реальной торговли уровни стоп-лосса и тейк-профита обрабатываются на стороне брокера на каждом тике. Чтобы при срабатывании стоп-лосса получить значения убытка, максимально приближенные к реальным уровням, необходимо будет проводить оптимизацию с отработкой каждого тика. Поэтому мы переходим на вкладку *Settings* и изменим режим моделирования на *Every tick based on real ticks*, что переведет тестер стратегий в режим обработки реальных исторических тиков. Также изменим режим оптимизации на *Fast genetic based algorithm.* Это позволит тестеру отсеять проходы, результаты которых будут заведомо хуже уже проведенных.

Variable	Value	Start	Step	Stop	Steps
☑ Model	gpt_not_norm.net				
☑ BarsToPattern	3				
☑ Common	true				
☐ TimeFrame	5 Minutes	current		1 Month	
☐ TradeLevel	0.6	0.5	0.05	1	
☐ Lot	0.01	0.01	0.001	0.1	
☐ MaxTP	1000	200	100	1000	
☐ ProfitMultiply	0.9	0.5	0.05	1	
☑ **MinTarget**	100	**350**	**50**	**400**	**2**
☑ **StopLoss**	100	**50**	**50**	**500**	**10**
					20

Оптимизация параметров стоп-лосса



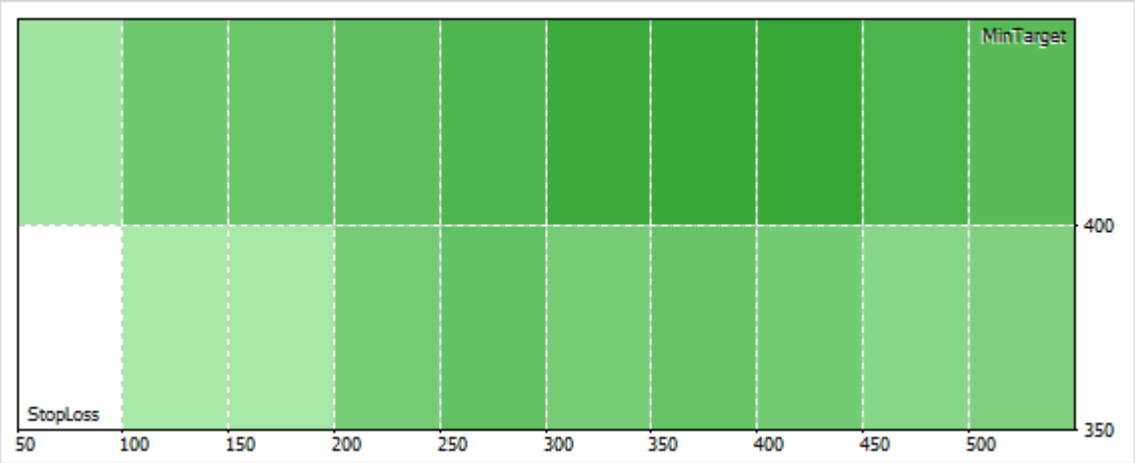
Оптимизация параметров стоп-лосса

По результатам оптимизации параметров, к сожалению, мы пока не видим прибыльных проходов. Но в то же время видно явное превосходство большего параметра принятия решения на основе силы предстоящего импульса *MinTarget*. При этом получены довольно близкие результаты для трех уровней стоп-лосса в диапазоне 300–400 пунктов.

Таким образом для последующей оптимизации параметров мы принимаем *MinTarget* на уровне 400 пунктов, а стоп-лосс продолжим оптимизировать в диапазоне 300–400 пунктов.

Pass	Result	Profit	Total tr...	Profit f...	Expect...	Drawd...	MinTar...	StopLoss
15	2970.63	-29.37	718	0.98	-0.04	3.13	400	400
13	2967.65	-32.35	762	0.98	-0.04	3.74	400	350
11	2962.85	-37.15	799	0.97	-0.05	3.98	400	300
17	2928.43	-71.57	700	0.95	-0.10	3.82	400	450
9	2924.67	-75.33	860	0.95	-0.09	4.17	400	250
19	2907.33	-92.67	679	0.94	-0.14	3.98	400	500
7	2895.35	-104.65	921	0.92	-0.11	4.49	400	200
8	2889.63	-110.37	1364	0.95	-0.08	4.59	350	250
12	2884.78	-115.22	1206	0.95	-0.10	5.48	350	350
5	2876.44	-123.56	1025	0.90	-0.12	5.03	400	150
3	2868.69	-131.31	1205	0.87	-0.11	4.86	400	100
14	2858.75	-141.25	1139	0.94	-0.12	6.38	350	400
10	2855.77	-144.23	1280	0.93	-0.11	5.98	350	300

Результаты подбора стоп-лосс



Результаты подбора стоп-лосс

Следующий параметр, который мы будем оптимизировать, это коэффициент доверия к прогнозируемой моделью силе ожидаемого импульса. Это тот коэффициент, на который мы будем умножать значение второго параметра, возвращаемого нашей моделью, при расчете тейк-профита открываемой позиции. Вполне логично, что мы не будем завышать ожидаемое значение импульса. Поэтому верхняя граница оптимизации параметра будет равна единице. Нижнюю границу оптимизации мы возьмем на уровне 0,5, что эквивалентно 50% прогнозируемого импульса. С шагом 0,05 получаем 11 проходов оптимизации. Помножим данное количество на 3 варианта стоп-лосса и получим 33 прохода предстоящей оптимизации параметров.

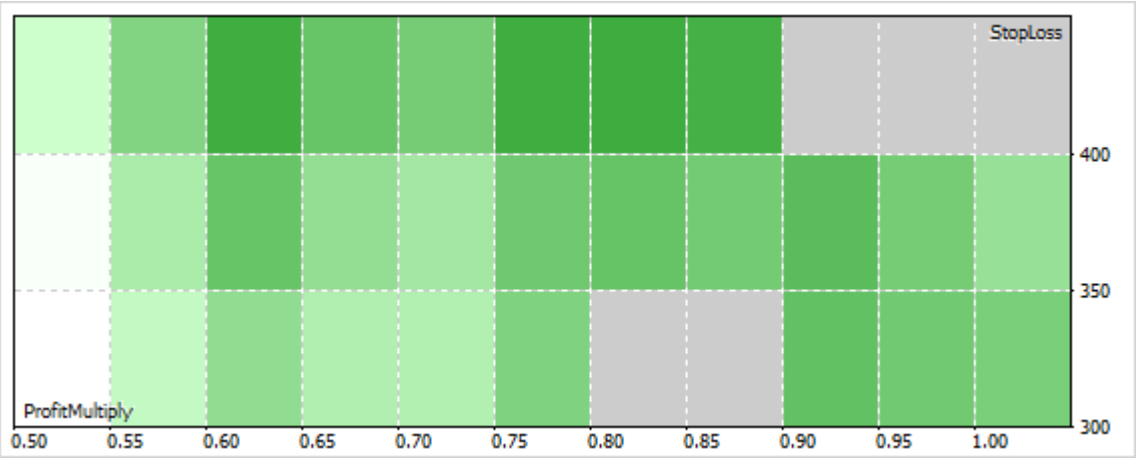
Variable	Value	Start	Step	Stop	Steps
☑ Model	gpt_not_norm.net				
☑ BarsToPattern	3				
☑ Common	true				
☐ TimeFrame	5 Minutes	current		1 Month	
☐ TradeLevel	0.6	0.56	0.01	0.64	
☐ Lot	0.01	0.01	0.001	0.1	
☐ MaxTP	1000	300	100	900	
☑ **ProfitMultiply**	0.5	**0.5**	**0.05**	**1.0**	**11**
☐ MinTarget	400	400	50	500	
☑ **StopLoss**	300	**300**	**50**	**400**	**3**
					33

Оптимизация коэффициента доверия

В результате оптимизации мы делаем однозначный выбор параметра стоп-лосса на уровне 400 пунктов и коэффициент доверия на уровне 0,8.

Pass	Result	Profit	Total tr...	Profit f...	Expect...	Drawd...	Profit...	StopLoss
28	2985.39	-14.61	763	0.99	-0.02	3.44	0.80	400
24	2984.49	-15.51	873	0.99	-0.02	2.89	0.60	400
27	2984.22	-15.78	784	0.99	-0.02	3.57	0.75	400
29	2980.48	-19.52	736	0.99	-0.03	3.27	0.85	400
19	2967.65	-32.35	762	0.98	-0.04	3.74	0.90	350
8	2962.85	-37.15	799	0.97	-0.05	3.98	0.90	300
17	2960.05	-39.95	809	0.97	-0.05	4.55	0.80	350
25	2959.68	-40.32	841	0.97	-0.05	3.71	0.65	400
13	2959.35	-40.65	912	0.97	-0.04	3.34	0.60	350

Результаты оптимизации коэффициента доверия



Результаты оптимизации коэффициента доверия

К сожалению, мы должны признать провал нашей авантюры. Мы так и не получили прибыльную комбинацию параметров. Вернемся к параметру порогового значения принятия решения по силе прогнозируемого импульса движения *MinTarget*. Напомню, что при прошлой оптимизации данного параметра мы получили максимальную прибыль на уровне 500 пунктов. Проведем небольшую повторную оптимизацию данного параметра в диапазоне от 400 до 500 пунктов с шагом в 50 пунктов.

Variable	Value	Start	Step	Stop	Steps
☑ Model	gpt_not_norm.net				
☑ BarsToPattern	3				
☑ Common	true				
☐ TimeFrame	5 Minutes	current		1 Month	
☐ TradeLevel	0.6	0.56	0.01	0.64	
☐ Lot	0.01	0.01	0.001	0.1	
☐ MaxTP	1000	300	100	900	
☐ ProfitMultiply	0.8	0.5	0.05	1.0	
☑ **MinTarget**	400	**400**	**50**	**500**	**3**
☐ StopLoss	400	300	50	400	
					3

Повторная оптимизация параметра принятия решения

В результате оптимизации мы получаем прибыльную комбинацию параметров на уровне 500 пунктов. Надо сказать, что уровень полученной прибыли практически вдвое больше полученной

ранее. При этом количество торговых операций сократилось, а фактора прибыли сохранился на уровне 1,22.

Pass	Res...	Profit	Total trad...	Profit fac...	Expected...	Drawdo...	MinTarget
2	3089.06	89.06	215	1.22	0.41	2.10	500
0	2985.39	-14.61	763	0.99	-0.02	3.44	400
1	2906.30	-93.70	438	0.90	-0.21	4.30	450

Результаты оптимизации параметра принятия решения

Variable	Value	Start	Step	Stop	Steps
Model	gpt_not_norm.net				
BarsToPattern	3				
Common	true				
TimeFrame	5 Minutes	current		1 Month	
TradeLevel	0.6	0.56	0.01	0.64	
Lot	0.01	0.01	0.001	0.1	
MaxTP	300	**300**	**100**	**900**	**7**
ProfitMultiply	0.8	0.5	0.05	1.0	
MinTarget	500	400	50	500	
StopLoss	400	300	50	400	
					7

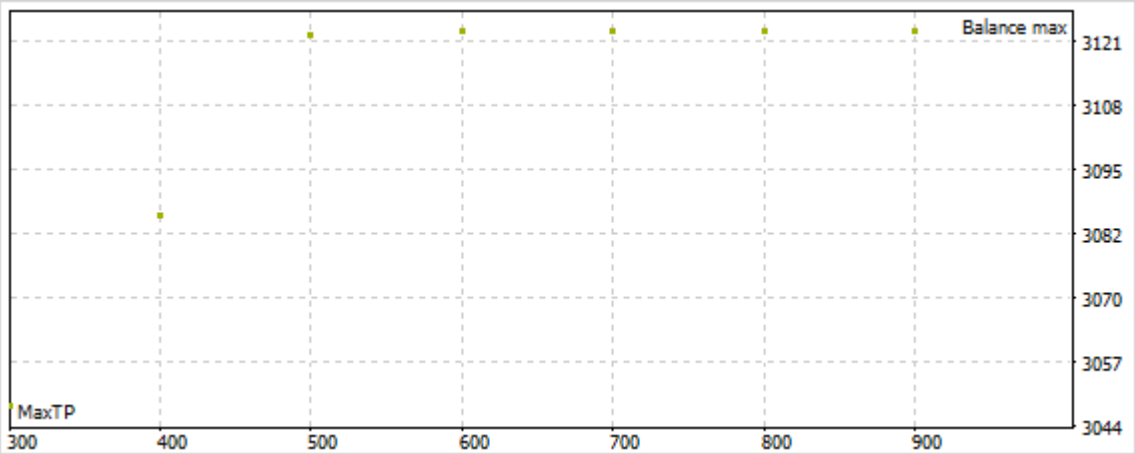
Оптимизация параметра ограничения доходности

Далее мы проведем оптимизацию параметра ограничения максимального уровня тейк-профита *MaxTP*. Данный параметр сложит некой блокировкой завышенных ожиданий. Если на новых значениях модель спрогнозирует завышенное ожидаемое движение, советник ограничит уровень тейк-профита данным значением, которое мы определим из статистики обучающей выборки. Значение параметра *MaxTP* будем оптимизировать в диапазоне от 300 до 900 пунктов с шагом в 100 пунктов.

По результатам оптимизации параметров можно заметить, что при изменении параметра выше 600 результат работы советника не изменяется. Следовательно, на всей обучающей выборке уровень ожидаемого движения не превышает 600 пунктов. Поэтому мы смело можем ограничить максимальный уровень прибыли в 600 пунктов.

Pass	Res...	Profit	Total trad...	Profit fac...	Expected...	Drawdo...	MaxTP
6	3122.69	122.69	215	1.32	0.57	1.46	900
5	3122.69	122.69	215	1.32	0.57	1.46	800
4	3122.69	122.69	215	1.32	0.57	1.46	700
3	3122.69	122.69	215	1.32	0.57	1.46	600
2	3121.93	121.93	215	1.32	0.57	1.46	500
1	3085.97	85.97	222	1.21	0.39	1.48	400
0	3047.89	47.89	243	1.12	0.20	1.55	300

Результаты оптимизации параметра ограничения доходности



Результаты оптимизации параметра ограничения доходности

Variable	Value	Start	Step	Stop	Steps
Model	gpt_not_norm.net				
BarsToPattern	3				
Common	true				
TimeFrame	5 Minutes	current		1 Month	
TradeLevel	0.56	**0.56**	**0.01**	**0.64**	**9**
Lot	0.01	0.01	0.001	0.1	
MaxTP	600	300	100	900	
ProfitMultiply	0.8	0.5	0.05	1.0	
MinTarget	500	400	50	500	
StopLoss	400	300	50	400	
					9

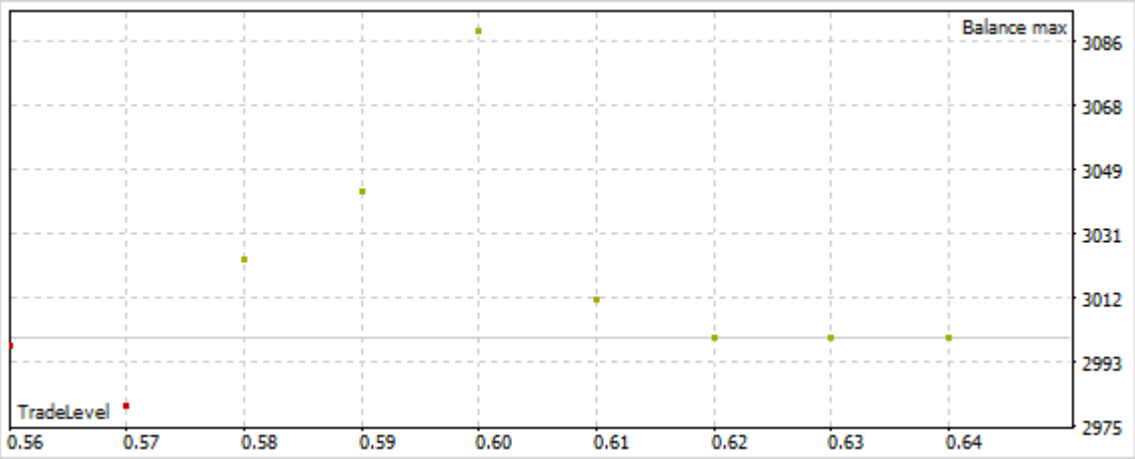
Тонкая настройка параметра принятия решения

Напоследок проведем тонкую оптимизацию параметра принятия решения по уровню вероятности движения *TradeLevel*. Ранее мы проводили грубую оптимизацию данного параметра с шагом 0,05 и остановились на уровне 0,6. Сейчас мы попробуем провести оптимизацию параметра с шагом 0,01 в окрестностях ранее выбранного уровня. Таким образом, мы проведем оптимизацию параметра в диапазоне 0,56–0,64.

Как ни странно, но проведенная оптимизация лишь подтвердила правильность ранее сделанного выбора значения параметра принятия решения на уровне 0,6. Любое отклонение параметра от данного значения отрицательно сказывается на доходности работы нашего советника.

Pass	Res...	Profit	Total trad...	Profit fac...	Expected...	Drawdo...	TradeLevel
4	3089.06	89.06	215	1.22	0.41	2.10	0.60
3	3042.58	42.58	365	1.06	0.12	3.66	0.59
2	3022.75	22.75	446	1.02	0.05	4.51	0.58
5	3011.26	11.26	72	1.08	0.16	1.51	0.61
8	3000.00	0.00	0		0.00	0.00	0.64
7	3000.00	0.00	0		0.00	0.00	0.63
6	3000.00	0.00	0		0.00	0.00	0.62
0	2997.80	-2.20	555	1.00	-0.00	4.11	0.56
1	2980.18	-19.82	506	0.98	-0.04	5.03	0.57

Результаты тонкой настройки параметра принятия решения



Результаты тонкой настройки параметра принятия решения

Итак, в результате проведенной работы по оптимизации, мы получили ниже следующий набор параметров, который позволяет получить прибыль на обучающей выборке.

Надо сказать, что для определения оптимального набора параметров мы провели довольно много итераций по их оптимизации. В то же время тестер стратегий терминала *MetaTrader 5* позволяет проводить оптимизацию любого количества параметров за один запуск процесса оптимизации. Но за это придется платить временем и вычислительными ресурсами. Если подсчитать суммарное количество совершенных проходов за все итерации оптимизации, получим порядка 95 проходов. А если мы запустим одновременную оптимизацию всех указанных выше параметров, то совокупное количество возможных вариантов параметров для осуществления проходов превысит 100 000. Можно надеяться на снижение количества проходов благодаря использованию генетических алгоритмов, но все равно их количество значительно превысит проведенные нами. Следовательно, и времени на оптимизацию параметров потребуется значительно больше.

Variable	Value	Start	Step	Stop	Steps
Model	gpt_not_norm.net				
BarsToPattern	3				
Common	true				
TimeFrame	5 Minutes	current		1 Month	
TradeLevel	0.6	0.56	0.01	0.64	9
Lot	0.01	0.01	0.001	0.1	
MaxTP	600	300	100	900	
ProfitMultiply	0.8	0.5	0.05	1	
MinTarget	500	400	50	500	
StopLoss	400	300	50	400	

Набор оптимизированных параметров

Ну а теперь, после определения оптимального набора параметров, проведем проверку работоспособности модели на новых данных.

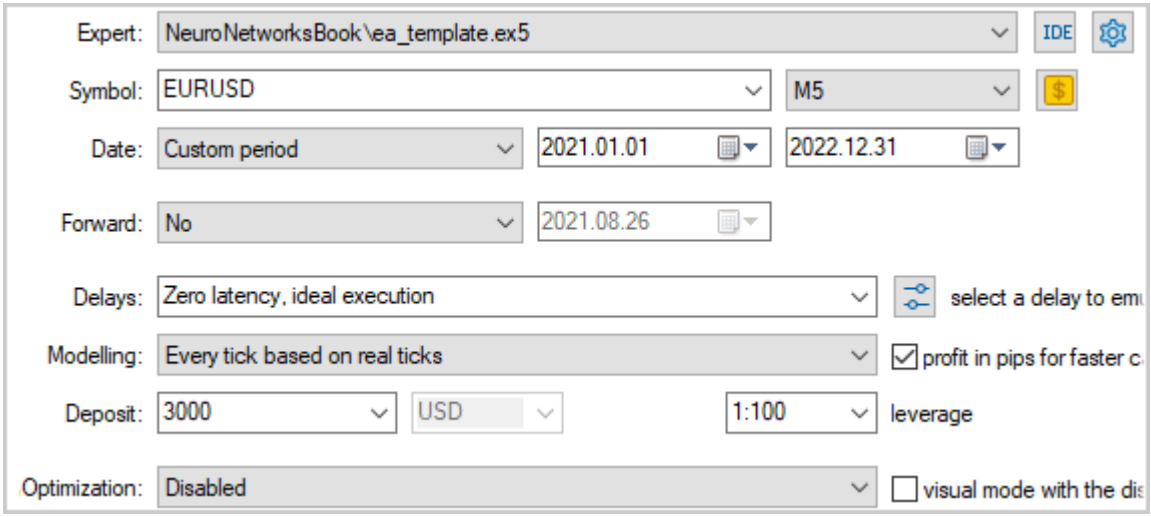
7.5 Тестирование модели на новых данных

В предыдущем разделе мы провели оптимизацию параметров нашего советника на обучающей выборке и определили оптимальный набор параметров. Теперь нам предстоит проверить работоспособность нашей модели на новых данных. Согласитесь, мы создаем модель для

возможности заработать на финансовом рынке. Пока мы только обучили модель и оптимизировали параметры советника на исторических данных за период с 2015 по 2020 год включительно. Мы определили оптимальный набор параметров, позволяющий получить прибыль на исторических данных. К сожалению, мы не можем отправиться в прошлое и заработать на исторических данных. Мы лишь можем запустить наш советник на торговом счете и надеяться на соизмеримую доходность советника в будущем. Чтобы подтвердить или опровергнуть возможность получения прибыли в будущем, проведем тестирование нашего советника с обученной моделью и оптимизированными параметрами на исторических данных вне обучающей выборки — на данных за 2021 год. Таким образом, мы проверим доходность модели на новых данных.

Как и в случае с оптимизацией параметров, мы переходим в тестер стратегий *MetaTrader 5* и на вкладке *Settings* указываем период тестирования 2021 год, выбираем тип моделирования на основе реальных тиков и отключаем оптимизацию параметров. Также не забудем указать корректный финансовый инструмент и таймфрейм.

После этого переходим на вкладку параметров советника и указываем значения параметров, которые мы определили в предыдущем разделе. Запускаем процесс тестирования кнопкой *Start*.



Форвард-тестирование модели

Variable	Value	Start	Step	Stop	Steps
Model	gpt_not_norm.net				
BarsToPattern	3				
Common	true				
TimeFrame	5 Minutes	current		1 Month	
TradeLevel	0.6	0.56	0.01	0.64	9
Lot	0.01	0.01	0.001	0.1	
MaxTP	600	300	100	900	
ProfitMultiply	0.8	0.5	0.05	1	
MinTarget	500	400	50	500	
StopLoss	400	300	50	400	

Форвард-тестирование модели

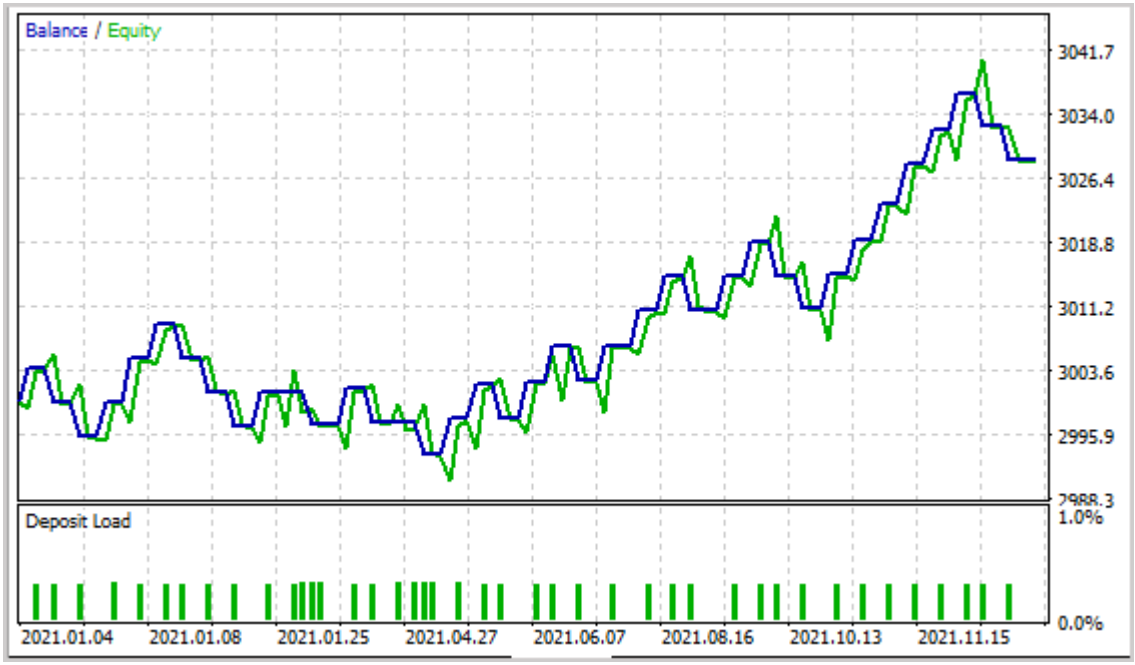
На периоде тестирования советник получал прибыль на длительном временном интервале. В целом год был закрыт с положительным результатом. Надо сказать, что для тестирования модели мы используем довольно упрощенный алгоритм советника без использования мани-менеджмента

и блока сопровождения позиции. Но и в таком варианте советник демонстрирует прибыль. Это свидетельствует об общей прибыльности торговых сигналов, генерируемых моделью. Потенциально добавление мани-менеджмента и блока сопровождения позиции позволит повысить доходность работы советника.

History Quality	100%				
Bars	74453	Ticks	20348987	Symbols	1
Initial Deposit	3 000.00				
Total Net Profit	28.64	Balance Drawdo...	6.39	Equity Drawdow...	9.18
Gross Profit	88.74	Balance Drawdo...	15.52 (0.5...	Equity Drawdow...	18.46 (0.61%)
Gross Loss	-60.10	Balance Drawdo...	0.52% (15....	Equity Drawdow...	0.61% (18.46)

Результаты форвард-тестирования модели

График изменения баланса демонстрирует боковое движение в первой половине года, но с мая наблюдается четкая тенденция к росту капитала.



Результаты форвард-тестирования модели

Анализ работы советника на новых данных показал, что по некоторым показателям он даже превосходит значения, полученные на обучающей выборке. К примеру, фактор доходности на новых данных составил 1,48, в то время как на обучающей выборке при оптимизации параметров данный показатель был на уровне 1,22. Уровень маржинальности в данном случае не показателен, так как все торговые операции осуществлялись с минимальным объемом, что сильно завысило данный показатель.

Profit Factor	1.48	Expected Payoff	0.80	Margin Level	29908.20%
Recovery Factor	1.55	Sharpe Ratio	2.35	Z-Score	0.00 (0.00%)
AHPR	1.0003 (0....	LR Correlation	0.83	OnTester result	0
GHPR	1.0003 (0....	LR Standard Error	6.53		

Результаты форвард-тестирования модели

В целом за весь 2021 год советник открыл 36 позиций, 21 из которых была закрыта с прибылью. Это составило 58.33% от общего числа позиций. Полученное значение очень близко к 60%

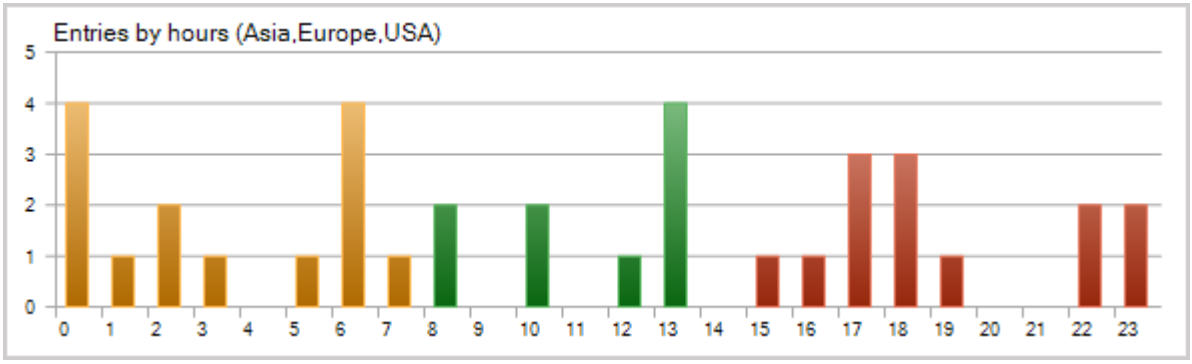
ожидаемой доходности от сигналов модели. Напомню, что пороговым уровнем совершения торговых операций является 60% вероятность движения цены в прогнозируемом направлении (параметр *TradeLevel=0.6*).

Максимальное количество идущих подряд убыточных сделок равно трем, а максимальное количество прибыльных — шести.

Total Trades	36	Short Trades (wo...	34 (55.88%)	Long Trades (wo...	2 (100.00%)
Total Deals	72	Profit Trades (% ...	21 (58.33%)	Loss Trades (% o...	15 (41.67%)
	Largest	profit trade	5.05	loss trade	-4.03
	Average	profit trade	4.23	loss trade	-4.01
	Maximum	consecutive win...	6 (25.67)	consecutive loss...	3 (-12.02)
	Maximal	consecutive prof...	25.67 (6)	consecutive loss ...	-12.02 (3)
	Average	consecutive wins	2	consecutive losses	2

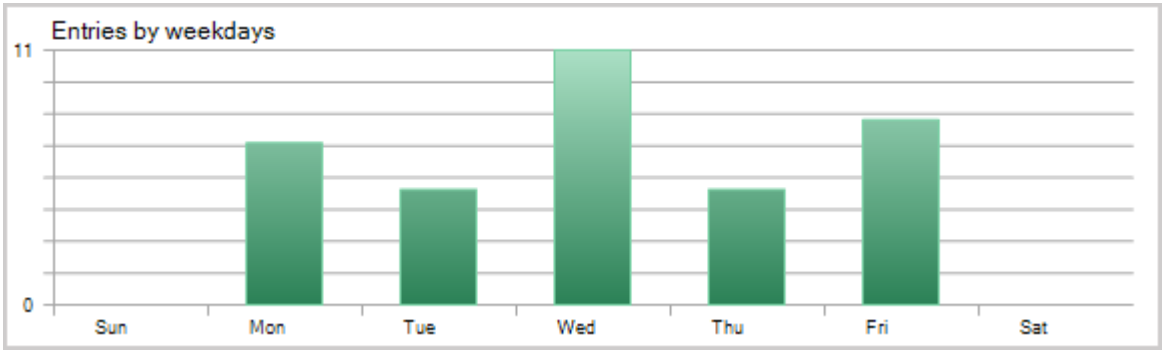
Результаты форвард-тестирования модели

Мы не встраивали в наш советник фильтрацию сделок по времени выполнения торговых операций, как и не давали временных ориентиров для обучаемой модели. В результате мы видим, что советник открывает позиции более или мене равномерно в течение всех торговых сессий.



Результаты форвард-тестирования модели

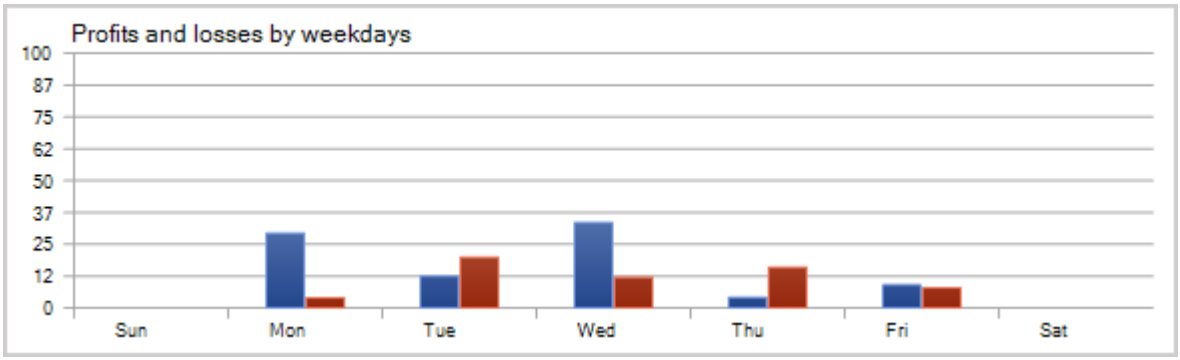
В течение же недели мы видим значительное превосходство по открытию позиций в среду (около 30%). Далее идут пятница и понедельник. Меньше всего открывается позиций во вторник и четверг.



Результаты форвард-тестирования модели

Максимальную же доходность советник получает в среду и понедельник. При этом показатель отношения прибыли к убытку лучше в понедельник. В пятницу полученные прибыль и убыток балансируют в районе точки безубыточности. А вот во вторник и четверг убытки превосходят полученную прибыль. Такой анализ позволяет потенциально увеличить доходность советника, исключив заведомо убыточные сделки. К примеру, если мы добавим фильтр открытия позиций по

дням недели, то можем увеличить общую доходность советника за счет совершения сделок только в понедельник и среду.



Результаты форвард-тестирования модели

В целом же результат получения прибыли советником на новых данных позволяет сделать следующие выводы:

1. В процессе проведения технического анализа возможно выделение неких паттернов, способных генерировать довольно устойчивые сигналы для совершения торговых операций с доходностью не менее 60%.
2. Использование моделей нейронных сетей позволяет выявлять такие паттерны.
3. Создание советника на базе нейронных сетей позволяет добиться устойчивой доходности на протяжении длительного временного интервала.

Заключение

Вот мы и добрались до конца книги. В ней мы рассмотрели несколько самых популярных архитектурных решений из разных областей. Это и сверточные модели из задач распознавания изображений, и рекуррентные модели из обработки временных последовательностей, и трансформер с механизмом *Self-Attention*, разработанный для решения языковых задач.

Я не просто так продемонстрировал вам абсолютно разные архитектурные решения. Никогда не бойтесь экспериментировать, ведь по проторенной дорожке всегда легче идти, но можно лишь повторить уже достигнутые результаты, как бы хороши они не были. И это не плохо. Порой, это даже больше, чем нужно. Но если вы хотите создать что-то новое, что-то свое, тогда придется сойти с дороги и направиться прямиком в неизвестность. И никто не знает, что нас там ждет — слава и успех или забвение. Но я верю, что усилия не бывают напрасными. Хочу пожелать вам, чтобы на своем пути вы достигли поставленных целей.

В данной книге мы построили библиотеку, которая поможет вам легко и просто реализовать собственные модели нейронных сетей, обучить их на исторических данных и проверить их работоспособность в тестере стратегий с помощью предложенного шаблона советника. Вам я желаю найти свою модель, которая принесет вам прибыль и благосостояние. Важно запомнить: тщательно проверьте и всесторонне протестируйте советник, прежде чем доверить ему свои сбережения.

Все программы из данной книги доступны прямо в среде разработки MetaEditior, не нужно ничего скачивать и копировать. Просто зайдите в Навигаторе в публичные проекты и найдите проект \MQL5\Shared Projects\NN_Book_Project. Все примеры разложены по соответствующим папкам проекта.

До скорых встреч. Вы всегда можете найти дополнительную информацию на сайте mql5.com.