

**ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ**

**Федеральное государственное автономное  
образовательное учреждение высшего образования**

**Национальный исследовательский университет  
"Высшая школа экономики"**

Факультет экономических наук  
Образовательная программа "Экономика"

**КУРСОВАЯ РАБОТА**

На тему "Применение нейронных сетей для анализа макроэкономических  
показателей"

Студент группы БЭК-162  
Цвигун Аким Олегович

Научный руководитель:  
Старший преподаватель Демешев Борис Борисович

Москва 2018

# Содержание

<b>1</b>	<b>Введение</b>	<b>3</b>
1.1	Актуальность . . . . .	3
1.2	Основы нейронных сетей . . . . .	3
1.3	Структура . . . . .	4
1.4	Методы обучения . . . . .	7
1.5	Метод обратного распространения . . . . .	8
<b>2</b>	<b>Нейронные сети RNN</b>	<b>10</b>
2.1	Особенности модели . . . . .	10
2.2	Принцип работы . . . . .	10
2.3	Недостатки RNN . . . . .	11
<b>3</b>	<b>Сети LSTM</b>	<b>11</b>
3.1	Введение в модель LSTM . . . . .	11
3.2	Механизм действия . . . . .	12
3.3	Пошаговый разбор LSTM . . . . .	13
3.4	Гиперпараметры . . . . .	15
<b>4</b>	<b>Программа на Python</b>	<b>16</b>
4.1	План работы . . . . .	16
4.2	Импорт данных . . . . .	16
4.3	Преобразование данных . . . . .	18
4.4	Модель экспоненциального скользящего среднего . . . . .	21
4.5	Модель LSTM . . . . .	22
<b>5</b>	<b>Заключение</b>	<b>25</b>
	<b>Литература</b>	<b>27</b>

# 1 Введение

## 1.1 Актуальность

Исследования по нейронным сетям вызваны тем, что способ обработки информации человеческим мозгом сильно отличается от методов, применяемых обычными цифровыми компьютерами. Мозг является сложной, нелинейной и параллельной системой обработки поступающей информации. При помощи своих структурных единиц, называемых нейронами, он способен выполнять различного рода задачи быстрее, чем это делают сверхмощные современные компьютеры. Примером подобной обработки поступающей информации может служить обоняние. Функцией системы обоняния является создание представления об окружающем мире таким образом, чтобы человек имел возможность взаимодействовать с ним. Благодаря этой способности (определять запах) мозг способен выполнять простейшие задачи (например, узнавать запах любимого продукта) в доли секунды. В то же время на выполнение даже более простых задач у компьютера могут уйти часы, а то и дни.

Аналогичным примером может служить орган термоллокации у некоторых разновидностей змей, таких как удавы, питоны и ямкоголовые гадюки. С его помощью змеи могут засечь местоположение своей добычи даже в полной темноте. Более того, сравнивая полученные сигналы, они (змеи) точно рассчитывают расстояние до объекта и затем атакуют. Подобных результатов мозгу человека или змее позволяет добиться совершенная структура, дающая возможность обучаться на основании того, что принято называть «опытом». Он накапливается с течением времени, моделируя свою картину восприятия мира в сознании животного посредством передачи определенных сигналов в определенные ячейки мозга — нейроны.

Понятие нейронов связано с понятием пластичности мозга — способности формирования нервной системы в соответствии с внешними эффектами. Именно это свойство — пластичность — играет ключевую роль в функционировании нейронов в качестве юнитов обработки информации в мозге. Подобно этому, в искусственных нейронных сетях аналогичную роль играют искусственные нейроны. Нейронная сеть, в свою очередь, представляет собой машину, моделирующую способ обработки данных мозгом и реализуемую посредством компьютеров. Таким образом, нейронная сеть — это большой параллельный и распределенный процессор, который состоит из простейших единиц обработки информации — нейронов. Они накапливают эмпирические знания и передают их для дальнейшей обработки.

## 1.2 Основы нейронных сетей

Нейронные сети, как уже было отмечено, очень схожи с мозгом. У них есть по крайней мере два общих признака<sup>1</sup>:

- данные поступают в нейронную сеть извне и используются во время обучения;
- процесс накопления знаний происходит посредством связей между нейронами — синаптических весов.

Процедура, с помощью которой нейронная сеть настраивается, называется алгоритмом обучения. Алгоритм в определенном порядке составляет синаптические веса модели для обеспечения требуемого устройства взаимосвязей нейронов.

Самыми важными инструментами нейронных сетей являются распараллеливание обработки данных и способность самообучаться, то есть создавать обобщения.

---

<sup>1</sup>[1], Neural Networks and Learning Machines Second Edition

Обобщение является способностью получать обоснованный результат по данным, не встречавшимся в процессе обучения. Эти свойства дают нейронным сетям возможность решать масштабные и трудно разрешимые на данный момент задачи. Кроме того, к свойствам нейронных сетей можно отнести следующие<sup>2</sup>:

- Нелинейность – очень важная характеристика нейронных сетей, особенно если механизм, отвечающий за формирование входного сигнала, также нелинейный: например, фотография или речь.
- Адаптивность – способность приспосабливать свои синаптические веса к изменениям внешней среды.
- Масштабируемость в рамках технологии VLSI (с английского “very large scale integrated” – сверхбольшая степень интеграции). Технология VLSI позволяет представить достаточно сложное поведение через иерархическую структуру.
- Отказоустойчивость – способность продуктивно работать даже при неблагоприятных условиях. Распределенный характер хранения данных в нейронной сети позволяет ей сохранять свою работоспособность даже при повреждении какого-либо нейрона или синапса.

### 1.3 Структура

В модели нейрона, лежащего в основе искусственных нейронных сетей, можно выделить четыре основных элемента:

- Набор синапсов (“synapse”) или связей (“connection link”), каждый из которых имеет только одну характеристику – вес или силу. Так, сигнал  $x_j$ , идущий в нейрон  $k$  через синапс  $j$ , умножается на вес  $w_{kj}$ . Следует отметить, что индексы синаптического веса  $w_{kj}$  идут в обратном порядке – сначала индекс выходного нейрона, а затем входного.
- Сумматор (“adder”), который, как следует из названия, складывает все входные сигналы, умноженные на соответствующий синаптический вес. Данной операции присуща линейность, поэтому ее можно описать как линейную комбинацию.
- Функция активации, которая ограничивает область значений выходного сигнала. Важно помнить, что нейроны оперируют с числами в диапазоне  $[0; 1]$  или  $[-1; 1]$ , и в этой связи числа, выходящие за рамки данной области, необходимо нормализовывать. Этим и занимается функция активации, которую иногда называют функцией сжатия (“squashing function”).
- Пороговый элемент или отклонение (“bias”)  $b_k$ , отражающий уменьшение или увеличение сигнала, обрабатываемого функцией активации.

Соответственно, можно описать архитектуру нейронной сети. Имеется три типа слоев: входной, скрытый и выходной, причем количество скрытых слоев в сети устанавливается в процессе создания нейросети: у небольших сетей обычно присутствует лишь один скрытый слой. У мощных и сложных нейросетей это количество неограниченно, но как правило не больше пяти — иначе сеть будет работать очень долго. Каждый слой, в свою очередь, состоит из нейронов, которые суммируют в них информацию, обрабатывают ее с помощью функции активации и передают дальше

---

<sup>2</sup>[1], Neural Networks and Learning Machines Second Edition

посредством синапсов, умножая результат на синаптический вес и прибавляя к нему пороговые элементы.

Математически работу нейрона  $k$  можно описать следующей системой уравнений: Система  $u_k = \sum_{j=1}^m w_{kj} \cdot x_j$ ,  $y_k = \phi(u_k + b_k)$ , где  $x_1, x_2, \dots, x_m$  — входные сигналы;  $w_{k1}, w_{k2}, \dots, w_{km}$  — веса у синапсов, идущих в нейрон  $k$ ,  $u_k$  — линейная комбинация входных воздействий (“linear combiner output”);  $b_k$  — пороговый элемент. Использование элемента смещения  $b_k$  обеспечивает эффект аффинного преобразования (“affine transformation”) выхода линейного сумматора  $u_k$ . В модели, представленной на рис. 1<sup>3</sup>, постсинаптический потенциал рассчитывается по следующей формуле:  $v_k = u_k + b_k$ .

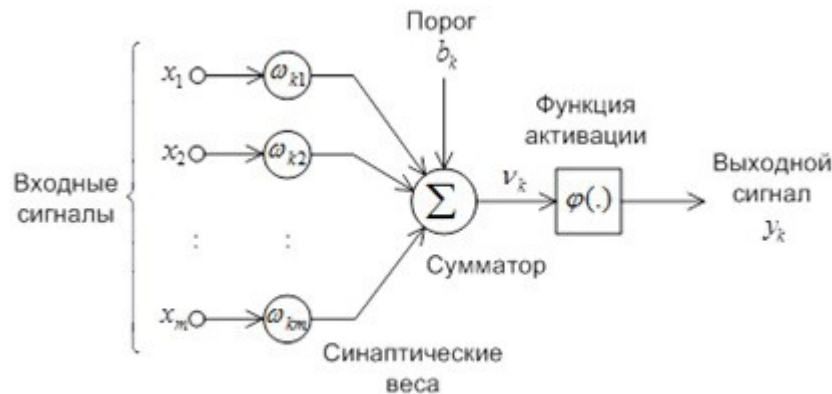


Рис. 1: Процесс работы RNN

Рассмотрим основные используемые разновидности функций активации (Таблица 1).

- Функция RELU (Rectified linear unit) — довольно простая функция, в силу чего используется реже двух других. В то же время благодаря своей простоте очень удобна в построении небольших моделей.
- Сигмоидальная функция (“sigmoid function”) — самая популярная функция при создании нейронных сетей. Данная функция является быстро возрастающей, поддерживая баланс между линейным и нелинейным поведением <sup>\*\*сноска\*\*</sup> (более подробное описание сигмоидальной функции содержится в [2]). Параметр  $\alpha$  является параметром наклона (“slope parameter”) и служит для того, чтобы строить сигмоидальные функции с разной крутизной. У сигмоидальной функции также есть очень удобные ключевые значения: при  $v \rightarrow -\infty : q(x) = 0$ ; при  $v = 0 : q(x) = \frac{1}{2}$ , а при  $v \rightarrow +\infty : q(x) = 1$ . Ее частным случаем является логистическая функция, у которой параметр  $\alpha$  равен 1<sup>4</sup>.
- Гиперболический тангенс — функция, главным преимуществом которой является ее область значений: в отличие от большинства функций активаций ее  $E(f) = [-1; 1]$ . В то же время использование гиперболического тангенса только с положительными значениями нецелесообразно, так как это значительно снижает результаты нейронной сети.

<sup>3</sup>[2], Прогнозирование экономических временных рядов с использованием нейросетевых технологий

<sup>4</sup>[3], Characterization of a class of sigmoid functions with applications to neural networks

Таблица 1: Различные функции активации

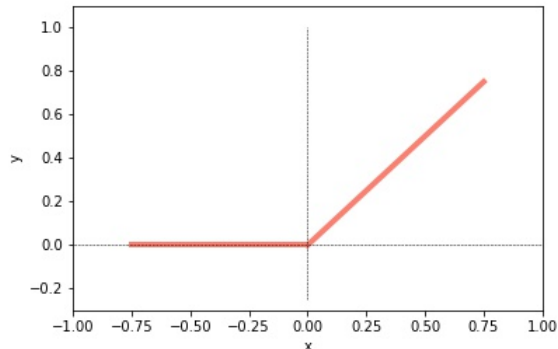
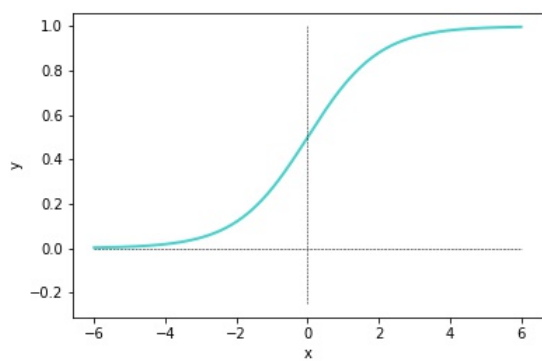
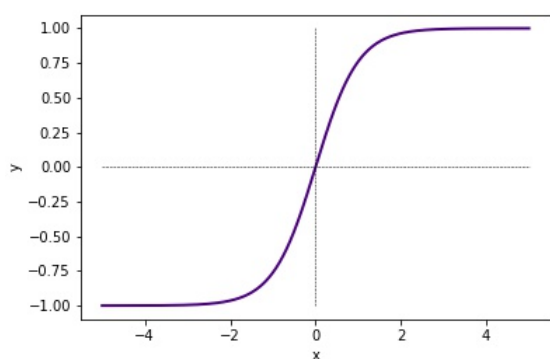
Название функции	Мат. запись	График функции
Ф-я RELU	$f(x) = \begin{cases} 0, & x < 0 \\ x, & x > 0 \end{cases}$	
Сигмоидальная ф-я	$\sigma = \frac{1}{1 + e^{-x}}$	
Гиперболический тангенс	$\tanh = \frac{e^{2x} - 1}{e^{2x} + 1}$	

Таблица 2: My caption

Название ф-и	Математическая запись
MSE	$\frac{1}{l} \sum_{i=1}^l (a(x_i) - y_i)^2$
RMSE	$\sqrt{\frac{1}{l} \sum_{i=1}^l (a(x_i) - y_i)^2}$
MAE	$\frac{1}{l} \sum_{i=1}^l  a(x_i) - y_i $

## 1.4 Методы обучения

Существует множество различных способов обучения нейронных сетей. Они отличаются целью обучения, наличием или отсутствием памяти, необходимостью учителя (то есть необходимостью какого-то набора данных, по которому сеть «настроится») и многим другим. Для обучения нейросети, предсказывающей значение на основе временного ряда требуется минимизировать ошибку – то есть расставить веса и элементы смещения таким образом, чтобы значение на выходе нейронной сети имело наименьшее отклонение от действительного значения. Методов вычисления ошибки также очень много. Среди основных стоит выделить среднеквадратическую ошибку MSE (“mean squared error”), ее корень RMSE (“root mean squared error”) и среднюю абсолютную ошибку MAE (“mean absolute error”) (см. также таблица 2). При обучении нейронной сети я использовал MSE. Несмотря на различающуюся размерность вводимых данных и результатов MSE, она является более предпочтительной вследствие, во-первых, возможностью дифференцирования, а во-вторых, такой функционал ошибки сильнее штрафует за крупные отклонения, а посему более чувствителен к выбросам.

Таким образом, ключевой задачей в обучении нашей нейросети является минимизация функции ошибки. Это представляет собой решение задачи оптимизации: по данной функции надо найти аргументы, в которых она имеет глобальный экстремум. Подробно про задачи и методы можно найти в следующих источниках ([3], [4]). В своей работе я остановлюсь на градиентном спуске – методе, который в разных формах используется повсеместно в машинном обучении и, в частности, в нейронных сетях.

Градиентом функции  $f : R^d \rightarrow R$  называется вектор его частных производных:  $\nabla f(x_1, \dots, x_d) = (\frac{\partial f}{\partial x_i})_{i=1}^d$ . Используя знания математического анализа, вспомним, что градиент функции – направление ее наискорейшего роста. Соответственно, антиградиент – направление ее наискорейшего убывания. Для минимизации функции ошибки будет логичным стартовать из некоторой точки, сдвинуться в сторону антиградиента, пересчитать его, снова двинуться в его сторону и так далее. Формализуя данный алгоритм, получаем, что нам требуется выполнять следующее преобразование до сходимости:  $w^{(k)} = w^{(k-1)} - \eta_k \nabla Q(w^{(k-1)})$ , где  $Q(w)$  – значение функционала ошибки для набора  $w$ , а  $\eta_k$  – длина шага, контролирующая скорость движения. Иногда длину шага можно сделать константой:  $\eta_k = c$ . При этом если значение длины шага слишком маленькое, то мы рискуем либо не выбраться из какого-либо локального минимума, либо затратить слишком много итераций для достижения точки минимума. В случае слишком большого значения шага существует вероятность постоянно «перепрыгивать» через эту точку. Зачастую длину шага монотонно уменьшают по мере приближения к точке минимума:  $\eta_k = \frac{1}{k}$ . Останавливать итерационный процесс

можно, когда изменение вектора весов на итерации меньше заданного порога или при близости градиента к нулю<sup>5</sup>.

Но даже у такого всеобъемлющего метода есть свои недостатки. В частности, основной проблемой градиентного спуска является необходимость подсчета градиента всей суммы на каждом шаге. Данный процесс, несомненно, является очень трудоемким даже при небольшом размере выборки. Между тем, точное вычисление градиента не так уж и необходимо ввиду того, что размер шага по направлению к точке минимума, как правило, не очень большой, а поэтому наличие небольших неточностей не скажется на общей траектории. Поэтому существуют различные модификации метода градиентного спуска. Первой из них является метод стохастического градиентного спуска (далее – SGD)<sup>6</sup>. Суть его заключается в том, что на каждом шаге мы ищем градиент не всей выборки, а отдельно взятого слагаемого. Соответственно, время расчета сильно снижается, но вместе с ней падает и точность. Кроме того, в памяти на каждом шаге необходимо держать лишь один объект из выборки. Эта особенность стохастического метода позволяет обучать модели на очень больших выборках: объекты можно считывать по одному, и по каждому делать один шаг метода SGD. В нашей работе мы будем использовать mini-batch метод (далее – MBGD), принцип работы которого заключается в том, что на каждой итерации мы ищем градиент случайного подмножества элементов из выборки. Таким образом, MBGD выступает в качестве золотой середины между SGD и классическим градиентным спуском: в MBGD сохраняет баланс между скоростью и точностью вычислений.

## 1.5 Метод обратного распространения

На методе обратного распространения (далее – МОР) целиком зависит нейронная сеть, которую мы будем строить. МОР, в свою очередь, основывается на методе градиентного спуска, в частности, на MBGD<sup>7</sup>. Рассмотрим механизм действия МОР. Сначала введем обозначения:

- $x$  — вектор входных обучающих данных.  $x = (x_1, x_2, \dots, x_i, \dots, x_n)$ .
- $t$  — вектор обучающих выходных значений:  $t = (t_1, t_2, \dots, t_k, \dots, x_n)$ .
- $X_i$  —  $i$ -тый входной нейрон. Для всех  $X_i$  входной сигнал равен выходному.
- $Z_j$  —  $j$ -тый скрытый нейрон.
- $z_j^{in}$  — суммарное значение, поступающее на вход скрытому нейрону  $Z_j$ .  $z_j^{in} = v_{0j} + \sum_i^n x_i \cdot v_{ij}$ .
- $z_j$  — сигнал на выходе  $Z_j$  — результат применения функции активации к  $z_j^{in}$ :  $z_j = f(z_j^{in})$ .
- $Y_k$  —  $k$ -тый выходной нейрон.
- $y_k^{in}$  — суммарное значение, поступающее на вход выходному нейрону  $Y_k$ .  $y_k^{in} = w_{0k} + \sum_j z_j \cdot w_{jk}$ .
- $y_k$  — сигнал на выходе  $Y_k$  — результат применения функции активации к  $y_k^{in}$ :  $y_k = f(y_k^{in})$ .

---

<sup>5</sup>[?], Learning TensorFlow

<sup>6</sup>[4], Optimization Models

<sup>7</sup>[?], A Gentle Introduction to Optimization



- $v_{ij}$  — вес синапса из  $X_i$  в  $Z_j$ .  $v_{0j}$  — отклонение нейрона  $Z_j$ .
- $w_{jk}$  — вес синапса из  $Z_j$  в  $Y_k$ .  $w_{0k}$  — отклонение нейрона  $Y_k$ .
- $\sigma_j$  — составляющая корректировки синаптических весов  $v_{ij}$ . Соответствует ошибке, распространяемой  $Z_j$ .
- $\sigma_k$  — составляющая корректировки синаптических весов  $w_{jk}$ . Соответствует ошибке, распространяемой  $Y_k$ .
- $\alpha$  — "learning rate" — параметр скорости обучения.

Алгоритм обучения выглядит следующим образом:

- I) Каждому синаптическому весу присваиваем случайное небольшое значение.
- II) Цикл III – X повторяется до тех пор, пока условие прекращения работы алгоритма неверно.
- III) Для каждой пары «Данные – целевое значение» выполняются шаги IV – IX  
Распространение данных от входов к выходам
- IV) Каждый входной нейрон ( $X_i, I = 1, 2, \dots, n$ ) отправляет полученный сигнал  $x_i$  всем нейронам в следующий (скрытый) слой.
- V) Каждый скрытый нейрон ( $Z_j, j = 1, 2, \dots, p$ ) суммирует взвешенные входящие сигналы (то есть входящие сигналы, умноженные на веса синапсов, по которым эти сигналы попадают в нейрон):  $z_j^{in} = v_{0j} + \sum_i x_i \cdot v_{ij}$ . Далее к значению в каждом скрытом нейроне применяется функция активации:  $z_j = f(z_j^{in})$ . После этого результат нейрона посылается в следующий слой: либо в скрытый, либо в выходной, в зависимости от количества скрытых слоев в нейронной сети. Этот шаг повторяется до тех пор, пока слой, в который поступают результаты нейронов, не будет выходным.
- VI) Каждый выходной нейрон ( $Y_k, k = 1, 2, \dots, m$ ) суммирует взвешенные входящие сигналы:  $y_k^{in} = w_{0k} + \sum_j z_j \cdot w_{jk}$ . После применения функции активации вычисляется выходной сигнал:  $y_k = f(y_k^{in})$ . Обратное распространение ошибки
- VII) Каждый выходной нейрон ( $Y_k, k = 1, 2, \dots, m$ ) получает целевое значение (то, которое является правильным для данного входного сигнала) и вычисляет ошибку:  $\sigma_k = (t_k - y_k) * f'(y_k^{in})$ . Далее вычисляется изменение синаптического веса:  $w_{jk} : \Delta w_{jk} = \alpha \cdot \sigma_k \cdot z_j$ . Также вычисляется величина корректировки смещения:  $\Delta w_{0k} = \alpha \cdot \sigma_k$ . После этого нейрон посылает  $\sigma_k$  по синапсам нейронам предыдущего слоя.
- VIII) Суммирование каждым скрытым нейроном ( $Z_j, j = 1, 2, \dots, p$ ) входящих ошибок (исходящих от нейронов следующего слоя):  $\sigma_j = \sum_k \sigma_k \cdot w_{jk}$ , а затем вычисляет величину ошибки (умножая полученное из следующего слоя значение на производную функции активации):  $\sigma_j = \sigma_j \cdot f'(z_j^{in})$ . Помимо этого, опять же, вычисляется изменение синаптического веса  $v_{ij} : \Delta v_{ij} = \alpha \cdot \sigma_j \cdot x_i$  и величина корректировки смещения:  $\Delta v_{0j} = \alpha \cdot \sigma_j$ . Этот шаг повторяется до тех пор, пока предыдущий слой не будет входным. Соответственно, в нейронных сетях с одним скрытым слоем этот шаг выполняется один раз.

- IX) Каждый выходной нейрон ( $Y_k, k = 1, 2, \dots, m$ ) изменяет веса своих синапсов, исходящих от элемента смещения и пороговых элементов:  $w_{jk}(new) = w_{jk}(old) + \Delta w_{jk}$ . Аналогично для скрытых нейронов ( $Z_k, k = 1, 2, \dots, p$ ):  $v_{ij}(new) = v_{ij}(old) + \Delta v_{ij}$ .
- X) Проверяем условие прекращения действия алгоритма. Это может быть как достижение суммарной квадратичной ошибкой заранее предустановленного минимума в ходе обучения, так и выполнение определенного количества итераций алгоритма.

Все вышеперечисленное в совокупности строит нейронную сеть, готовую давать очень точные предсказания. Однако процессе обучения никак не использовалась связь с предыдущим элементом — а стало быть, подобные нейросети не подойдут для анализа временных рядов, где каждый последующий элемент зависит от предыдущих. Для анализа объектов такого рода были изобретены рекуррентные нейронные сети (далее — RNN) — Recurrent Neural Networks.

## 2 Нейронные сети RNN

### 2.1 Особенности модели

Отличие RNN от базовой модели заключается в том, что на вход в скрытый слой подается не только информация из входных нейронов, но и предыдущее состояние сети ( $h_t$ ). В этой связи результат, получаемый на выходе, напрямую зависит от предыдущего результата. Чтобы лучше понять это, рассмотрим простейшую модель такой нейронной сети.

Пусть в нашей нейронной сети имеется только по одному входному, скрытому и выходному нейрону. Если ранее на вход в скрытый слой в момент  $t$  поступало  $x_t \cdot w_t$ , где  $x_t$  — информация, поступающая во входной нейрон,  $w_h$  — вес синапсов в скрытый слой, то теперь информация на входе в скрытый нейрон равна  $x_t \cdot w_h + h_{t-1} \cdot v$ , где  $h_{t-1}$  — состояние сети в предыдущий момент ( $t - 1$ ), а  $v$  — вес этого состояния (то, с каким коэффициентом оно учитывается). Тот факт, что теперь при расчете скрытого состояния нейрона мы используем предыдущее состояние означает, что текущее значение будет зависеть от предшествующих. Это и является особенностью RNN — расчет новых значений происходит с учетом предыдущих.

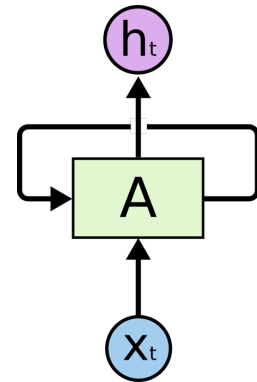


Рис. 2: Простейшая модель RNN

### 2.2 Принцип работы

Модель RNN представлена на рис. 2<sup>8</sup>. В целом механизм работы нейронов в RNN состоит в следующем: на вход в момент  $t$  поступает вектор значений  $x_t$  с весом  $w$ . Параллельно с ним в нейрон передается предыдущее состояние сети с весом  $v$ . Далее все происходит, как в базовой модели нейронной сети — значение в скрытом

<sup>8</sup>[5], Understanding LSTM Networks

слое рассчитывается по формуле  $f(x_t \cdot w_h + h_{t-1} \cdot v)$ , где  $f(x)$  - функция активации. По умолчанию в RNN используется гиперболический тангенс. После этого значения скрытого слоя умножаются на вес синапсов в выходном слое и поступают на выход. Подобная структура является универсальной в силу своей простоты с одной стороны и связью с предыдущими значениями с другой<sup>9</sup>.

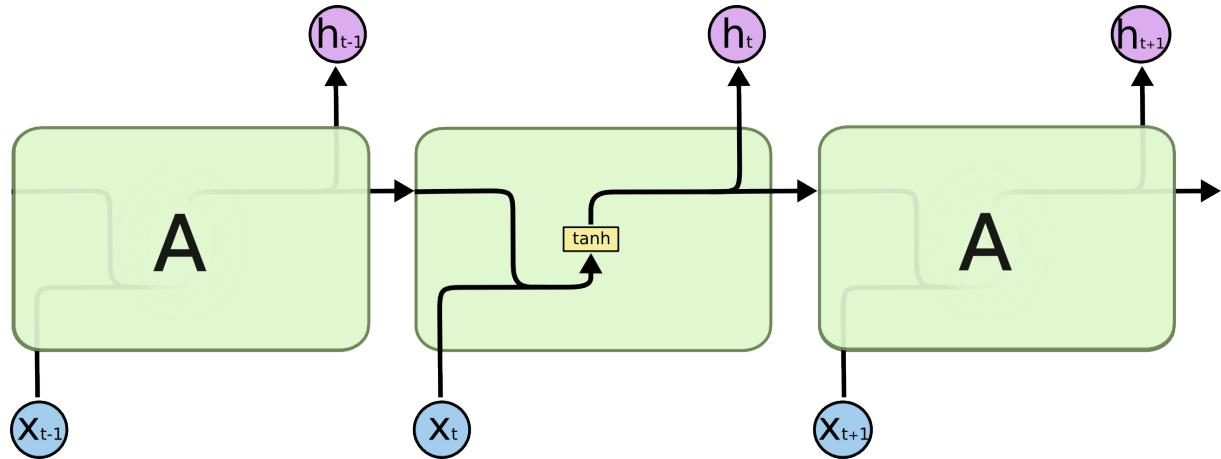


Рис. 3: Процесс работы каждого нейрона в RNN

## 2.3 Недостатки RNN

Однако даже у такой модели существуют свои серьезные недостатки и пробелы. Проблема долгосрочной зависимости в RNN проявляется в двух вариантах — взрывающиеся и затухающие градиенты. Рассмотрим их несколько подробнее. Пусть у нас есть состояние сети в момент  $t$ . Оно равно  $x_t \cdot w_h + h_{t-1} \cdot v$ . С каждым шагом оно умножается на вектор весов  $v$ . Таким образом, к моменту  $t$  состояние  $t - 1$  умножается на  $v$ , к моменту  $t + 1$  — на  $v^2$ , к моменту  $t + 2$  — на  $v^3$ , и так далее, а к моменту  $t + n$  — на  $v^{n+1}$ . Если вектор весов  $v > 1$ , то состояние в момент  $t - 1$  ( $h_{t-1}$ ) будет экспоненциально расти, и к моменту  $n$  вырастет в  $v^{n+1}$  раз, "затмив" при этом последние состояния сети, которые будут поступать с весами  $v$ ,  $v^2$  и так далее. Аналогично при  $v < 1$ , к моменту  $n$   $h_{t-1}$  умножится на  $v$  столько раз, что оно будет близко к нулю и практически не будет нести в себе какую-либо информацию<sup>10</sup>.

Для решения проблемы взрывающихся и затухающих градиентов существует модификация RNN — нейронные сети с долгой краткосрочной памятью — Long Short Term Memory — LSTM. Они имеют более сложную структуру, нежели RNN, однако благодаря дополнительным ячейкам памяти они позволяют предсказывать информацию гораздо точнее.

## 3 Сети LSTM

### 3.1 Введение в модель LSTM

Само название данного типа рекуррентных сетей — Long Short Term Memory — говорит само за себя. Это нейронные сети, в которых присутствует как краткосрочная,

<sup>9</sup>[6], Глубокое обучение

<sup>10</sup>[1], Neural Networks and Learning Machines Second Edition

так и долгосрочная память. Это свойство позволяет им работать с последовательностями любого рода. Структура LSTM приведена на рис. 3<sup>11</sup>. Рассмотрим более подробно принцип работы таких сетей<sup>12</sup>.

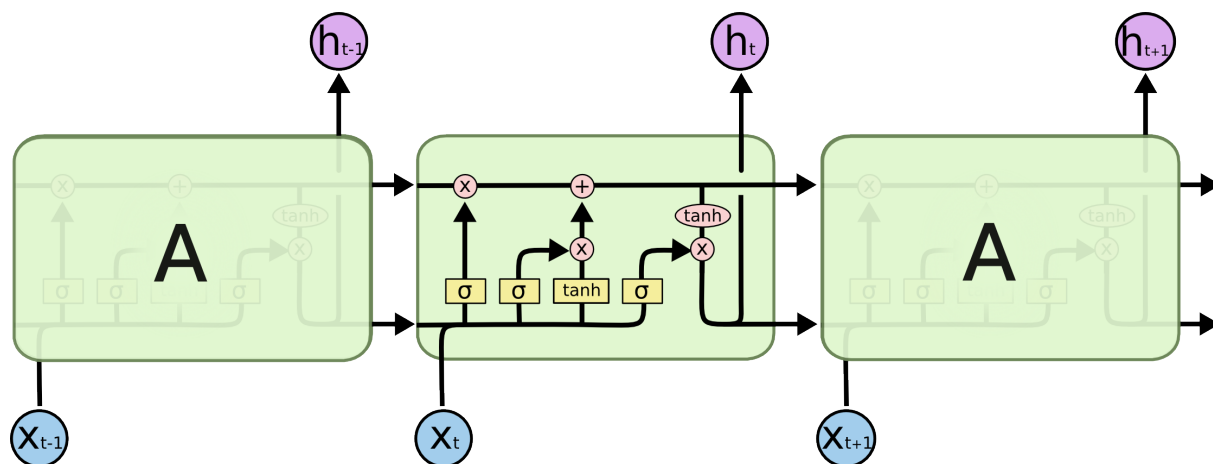


Рис. 4: Процесс работы LSTM

### 3.2 Механизм действия

Ключевым элементом в LSTM является состояние ячейки — cell state. Оно представляет собой горизонтальную линию, которая идет вдоль всей сети. Состояние ячейки напоминает конвейерную ленту — она проходит напрямую через всю цепочку, время от времени подвергаясь небольшим линейным преобразованиям. Информация словно течет по ней, не подвергаясь изменениям извне (рис. 4)<sup>13</sup>

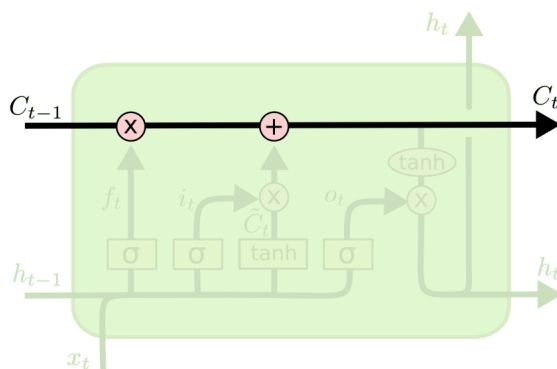


Рис. 5: Строение состояния ячейки

В LSTM существует возможность добавлять или удалять данные из состояния ячейки. Внесение изменений в информацию происходит посредством особых компонентов, называемых фильтрами или гейтами (gates). Они состоят из сигмоидального слоя и операции поточечного умножения (рис. 5)<sup>14</sup>.

<sup>11</sup>[5], Understanding LSTM Networks

<sup>12</sup>[7], GRU и LSTM: современные рекуррентные нейронные сети

<sup>13</sup>[5], Understanding LSTM Networks

<sup>14</sup>[5], Understanding LSTM Networks

Сигмоидальный слой возвращает числа в пределах от нуля до единицы, показывая, какую долю каждого блока информации нужно пропустить далее по сети. Такой диапазон значений достигается областью значений сигмоидальной функции, лежащей в основе фильтров. Ноль означает "полностью забыть единица — "полностью пропустить". LSTM содержит три таких фильтра, которые позволяют хранить и контролировать информацию в состоянии ячейки.

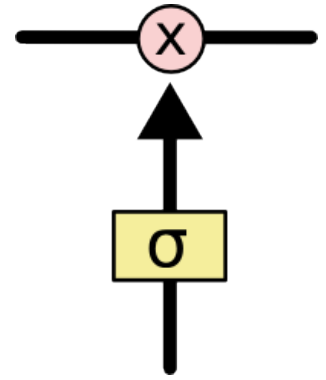


Рис. 6: Структура фильтра с сигмоидальным слоем

### 3.3 Пошаговый разбор LSTM

Первый шаг в LSTM — решить, какую информацию из состояния ячейки необходимо "забыть" — выкинуть из памяти. Это определяет сигмоидальный слой под названием "фильтр забывания". Он смотрит на предыдущее состояние ячейки  $h_{t-1}$  и входной вектор  $x_t$  и возвращает число в пределах от нуля до единицы для каждого числа состояния ячейки  $C_{t-1}$ . Как уже было отмечено, 1 означает полностью оставить, 0 — полностью забыть (рис. 6)<sup>15</sup>.  $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$

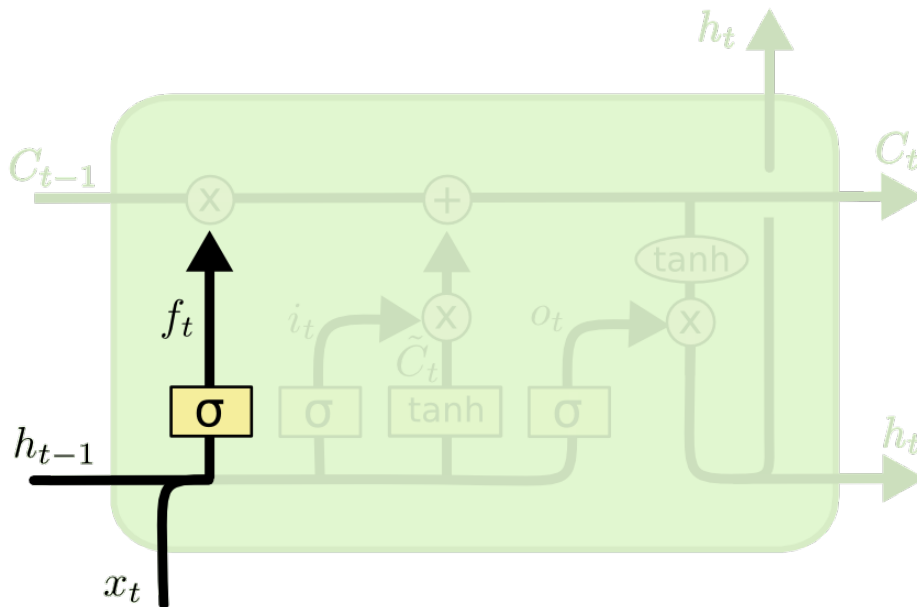


Рис. 7: Строение фильтра забывания

Следующим шагом является принятие решения о том, какая доля новой информации будет храниться в состоянии ячейки. Это решение состоит из двух частей. Сначала сигмоидальный слой, называющийся "входной фильтр определяет, какие значения мы хотим обновить:  $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$ . Затем tanh-слой создает вектор новых значений (рис. 7)<sup>16</sup>:  $\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$ .

Настало время обновить старое состояние ячейки  $C_{t-1}$  в новое  $C_t$ . Старое состояние умножается на фильтр забывания, тем самым избавляясь от информации,

<sup>15</sup>[5], Understanding LSTM Networks

<sup>16</sup>[5], Understanding LSTM Networks

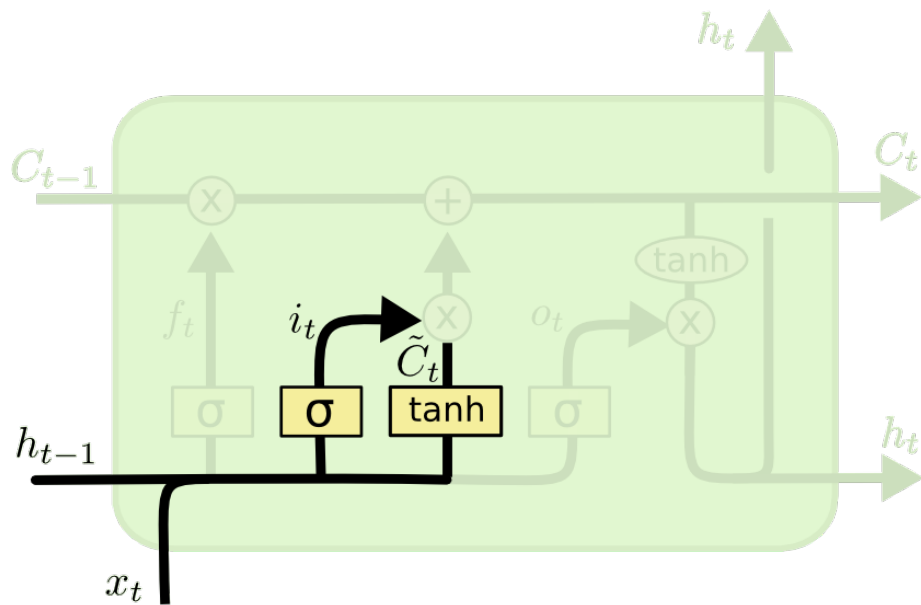


Рис. 8: Строение входного фильтра и tanh слоя

которая более не является актуальной. Затем к результату прибавляется вектор новых значений, умноженный на входной фильтр, добавляя в состояние ячейки новую информацию, умноженную на ее "важность" (рис. 8)<sup>17</sup>:  $C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$ .

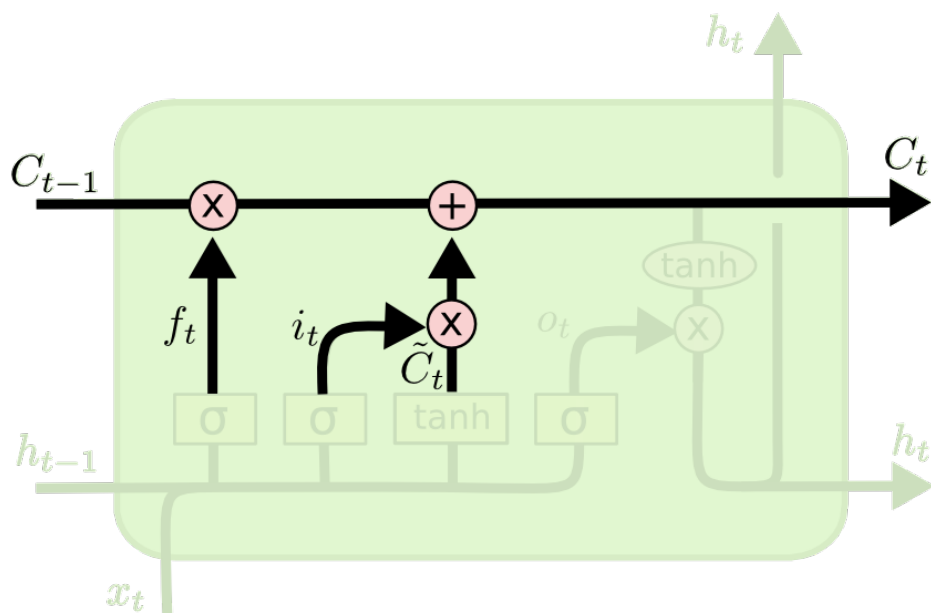


Рис. 9: Механизм обновления состояния ячейки

Последний шаг — принятие решения о выходных данных. На выход поступает все то же состояние ячейки, но только обработанное несколькими функциями. Сначала сигмоидальный слой выходного фильтра определяет, какую часть информации мы собираемся подать на выход. Затем состояние ячейки обрабатывается через tanh, чтобы значения лежали в диапазоне  $[-1; 1]$ , и умножается на результат выходного фильтра, чтобы на выходе была только интересующая нас информация. Выглядит

<sup>17</sup>[5], Understanding LSTM Networks

это следующим образом:  $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$ ;  $h_t = o_t * \tanh(C_t)$ .

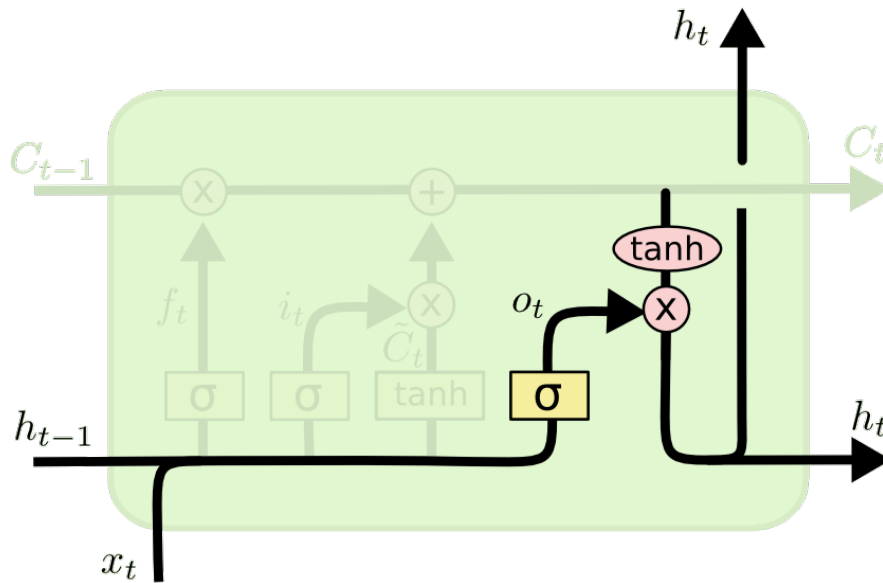


Рис. 10: Работа выходного фильтра

### 3.4 Гиперпараметры

Во всех нейронных сетях, как и в любой модели, можно выделить гиперпараметры - элементы, оптимальные значения которых зависят от самой модели, и поэтому подбираются методом проб и ошибок. В нейронной сети LSTM можно выделить следующие гиперпараметры:

- Количество скрытых нейронов в слое. С этим параметром все просто - чем больше нейронов, тем лучше качество модели и тем дольше она работает.
- Количество эпох обучения - аналогичный гиперпараметр. Чем его значение выше, тем лучше качество и медленнее скорость.
- Метод нормализации данных - зависит от данных, по которым проводится обучение сети. Как правило, используется стандартный метод - из каждого объекта вычитается его математическое ожидание, полученный результат делится на стандартное отклонение. Однако в случае, если временной ряд имеет очень маленькую амплитуду колебания, данный метод приведет данные к примерно одинаковым значениям, что, в свою очередь, сильно затруднит процесс обучения сети. Поэтому иногда стоит прибегнуть к иным способам нормализации данных.
- Функция ошибки - MSE, MAE, RMSE и другие. Оптимальная, опять же, зависит от данных и от того, какая цель поставлена перед нейронной сетью. Например, для регрессии, как правило, используется MSE, потому что она более чувствительна к выбросам, а для задач классификации зачастую выбирается MAE, потому что важность представляет направление тренда, а не конкретные значения.

- Скорость обучения - learning rate. Этот параметр очень сильно варьируется от модели к модели. В работе представлена адаптивная скорость обучения - она тем выше, чем ближе оптимизатор к точке глобального минимума.
- Метод оптимизации - как уже было сказано, в работе будет использован метод обратного распространения.
- Величина 'дропаута' - какая доля старой, входящей и выходящей информации забывается на каждом шаге LSTM.
- Функции активации - гиперпараметр, который подбирается исключительно методом проб и ошибок.

В разных моделях существуют другие гиперпараметры — выше приведены основные в сетях LSTM.

Таким образом, сети LSTM позволяют решать задачи с использованием долгосрочной зависимости. Именно эта характеристика дает этому типу нейронных сетей колоссальное преимущество перед другими разновидностями. Поэтому при анализе временных рядов, в которых долгосрочная зависимость показателей является неотъемлемой составляющей, логично использовать именно LSTM, что я и проделал в своей работе.

## 4 Программа на Python

### 4.1 План работы

Мы приближаемся к основной части работы - к написанию кода для прогнозирования макроэкономических данных. Модель LSTM, как уже было отмечено, является очень мощным инструментом в машинном обучении, вследствие чего в коде я решил использовать именно ее. Для начала определимся с планом.

- 1) Импорт требуемых библиотек и данных для обучения и тестирования модели - показателей акций фондового рынка с Yahoo-finance.
- 2) Деление данных на тренировочные и тестовые с последующей нормализацией.
- 3) Построение простейших моделей прогнозирования и оценка качества их работы.
- 4) Построение и дальнейшее предсказание показателей с помощью модели LSTM.

### 4.2 Импорт данных

Итак, приступим. Первое, что требуется сделать - импортировать все необходимые библиотеки (строчки 1-9). Вкратце рассмотрим каждую из них.

- 1) pandas - библиотека, необходимая для чтения файлов "csv", а также для представления данных в табличной форме.
- 2) matplotlib.pyplot - модуль всеобъемлющей библиотеки matplotlib, посредством которого происходит визуализация данных.
- 3) os - библиотека для чтения файлов с компьютера (в нашем случае необходима для загрузки скачанных файлов).



- 4) numpy - пакет для проведения некоторых математических операций.
- 5) urllib.request, json - модули для работы в интернете.
- 6) tensorflow - основная библиотека в нашей программе - с его помощью происходит непосредственно построение модели LSTM.
- 7) MinMaxScaler - функция из пакета sklearn для нормализации данных.
- 8) datetime - библиотека для обработки времени.

```
import matplotlib.pyplot as plt
import pandas as pd
import datetime as dt
import urllib.request, json
import os
import numpy as np
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
```

Следующий шаг - загрузка данных, на основе которых будут строиться предсказания. Данные, которые мы будем загружать, представляют собой выборку из более чем трех тысяч котировок акций скромной компании Apple. [Скачать данные](#) - по этой ссылке данные можно и нужно скачать и поместить в директорию с файлом, в котором будет запускаться программа. Котировки акций рассматриваются в четырех категориях:

- Open - начальная цена акций за день;
- Close - финальная цена акций за день;
- High - самая высокая цена акций за день;
- Open - самая низкая цена акций за день;

Соответственно, загрузим данные в таблицу, отсортируем их по дате и выведем их описание, количество и статистику.

```
# Загружаем данные в таблицу
df = pd.read_csv('AAPL.csv')

# Сортируем данные по дате
df = df.sort_values('Date')

# Выводим описание наших данных
df.describe()
```

Как уже было отмечено, цены представлены в четырех категориях. Для того, чтобы отслеживать общую динамику акций, достаточно взять среднюю цену за день - арифметическое среднее минимальной и максимальной.

	Open	High	Low	Close	Adj Close	Volume
count	3089.000000	3089.000000	3089.000000	3089.000000	3089.000000	3.089000e+03
mean	54.408180	54.926004	53.841993	54.397401	45.305461	1.413024e+08
std	39.924409	40.219641	39.625588	39.930786	39.034974	1.031620e+08
min	4.556428	4.636428	4.471428	4.520714	3.048571	1.147590e+07
25%	17.458570	17.755714	17.185715	17.434286	11.756919	6.662250e+07
50%	47.828571	48.127144	47.411430	47.805714	32.238071	1.154377e+08
75%	89.692856	90.665718	89.048569	89.807144	68.650002	1.885058e+08
max	144.289993	145.460007	143.809998	144.770004	141.968033	8.432424e+08

Рис. 11: Резюме по данным

```
# Выбираем столбец с максимальной ценой за день и
# представляем его в форме массива
high = df.loc[:, 'High'].as_matrix()

# Аналогично для минимальной цены
low = df.loc[:, 'Low'].as_matrix()

# Берем среднее арифметическое
average = (high + low) / 2
```

Визуализируем наши данные: рассмотрим их в динамике.

```
# Определяем размер нашей фигуры
plt.figure(figsize = (18,9))

# По оси абсцисс - числа от нуля до последнего индекса
# массива со средними, по оси ординат - сами показатели
plt.plot(range(len(average)), average)

# Настраиваем, как будет выглядеть ось абсцисс
plt.xticks(range(0, len(average), 500),
            df['Date'].loc[::500], rotation=45)

# Именуем оси
plt.xlabel('Дата', fontsize=18)
plt.ylabel('Средняя цена', fontsize=18)

# Выводим получившийся график
plt.show()
```

## 4.3 Преобразование данных

Для начала инициализируем все необходимые нам переменные.

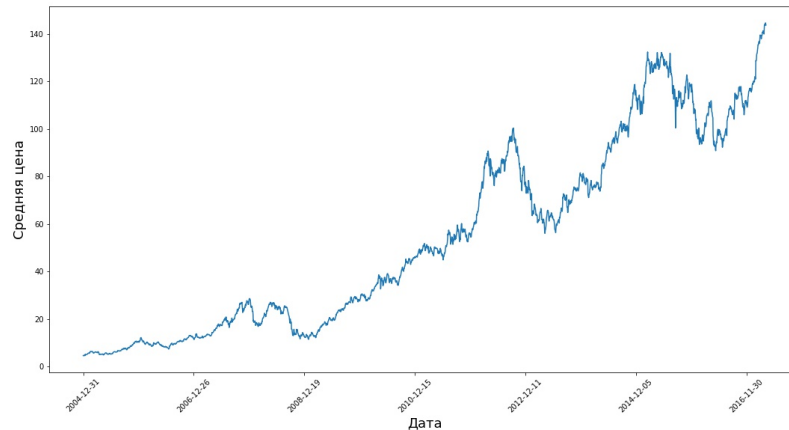


Рис. 12: График имеющихся данных

```
# Количество "шагов вперед", то есть на сколько
# шагов вперед мы будем делать предсказания
f_horizon = 1

# Размер каждой выборки с данными, которые будут подаваться на вход
batch_size = 300

# Количество клеток в каждом скрытом слое LSTM
num_hidden = [200, 200, 150]

# Количество слоев (равно 3)
n_layers = len(num_hidden)

# Количество нейронов в последнем (выходном) слое
num_last = num_hidden[-1]

# Размерность входных данных: так как работаем
# с одномерными массивами, размерность равна 1
num_input = 1

# Размерность выходных данных: так как работаем
# с одномерными массивами, размерность равна 1
num_output = 1

# Количество эпох - сколько раз будет проводиться операция обучения
num_epochs = 10**(3) + 1

# Доля выбрасываемой на каждом шаге информации
dropout = 0.2
```

У нас имеется информация о чуть более, чем трех тысячах наблюдений. Пусть первые 2700 будут тренировочными, то есть теми, на которых мы нашу модель будем обучать, а остальные — тестовыми.

```
# Тренировочным данным присваиваем первые 2700 значений,
# остальные делаем тестовыми
train_data = average[:2700]
test_data = average[2700:]
```

Теперь, когда мы разделили данные на обучающие и тестовые, проведем нормализацию. Во-первых, нормализацию мы будем проводить по группам (батчам). В противном случае первая половина данных практически обнулится, и пользы от нее будет мало. Во-вторых, мы нормализуем все данные в соответствии с тренировочными, потому что предполагается, что у нас нет доступа к тестовым данным. Функция "MinMaxScaler" нормализует данные по следующему принципу:

$$X_{std} = \frac{(X - X_{min})}{X_{max} - X_{min}};$$

$$X_{scaled} = X_{std} \cdot (X_{max}^{new} - X_{min}^{new}) + X_{min}^{new},$$
 где  $X_{min}^{new}/X_{max}^{new}$  - минимальное / максимальное значение в группе после применения "MinMaxScaler": в нашем случае это 0 и 1.

```
# Объявляем переменную scaler нашим "нормализатором";
# также преобразовываем данные в нужную нам размерность
scaler = MinMaxScaler()
train_data = train_data.reshape(-1, 1)
test_data = test_data.reshape(-1, 1)
```

Разделим наши данные на 10 групп, по 300 элементов в каждой. Выполним процедуру нормализации для каждой группы и приведем данные к одномерному массиву. Затем обрежем данные до требуемого размера.

```
# Инициализируем размер окна и производим нормализацию для
# каждой группы, настраивая scaler на тех же данных.
# После этого приводим данные к нужному нам виду
window_size = 300
for period in range(0, 2700, window_size):
    train_data[period:period + window_size, :] = scaler.fit_transform(
        train_data[period:period + window_size, :])

# Приводим тренировочные данные к одномерному массиву
train_data = train_data.reshape(-1)

# Нормализуем тестовые и приводим их к одномерному массиву
test_data = scaler.transform(test_data).reshape(-1)

# Сохраним нормализованные данные целиком для последующих операций
new_data = np.concatenate([train_data, test_data])

# Обрезаем массив со всеми данными до количества
# "сколько раз batch_size полностью помещается
# в new_data" + f_horizon. Иными словами, мы отбрасываем
# ту часть new_data, в которой не сможет поместиться
# очередной batch_size, прибавляя f_horizon элементов new_data
new_data = new_data[:-(len(new_data) % batch_size) + f_horizon]
```

## 4.4 Модель экспоненциального скользящего среднего

Чтобы иметь возможность сравнить нашу будущую LSTM модель с какой-либо более простой, построим модель экспоненциального скользящего среднего (Exponential moving average - ЕМА).

Будем рассчитывать значение показателя в периоде  $t + 1$  как  $x_{t+1} = EMA_t = \gamma \cdot EMA_{t-1} + (1 - \gamma) \cdot x_t$ ,  $EMA_0 = 0$ , и значение ЕМА сохраняется на протяжении всех вычислений. Составим такую модель с  $\gamma = 0.5$ .

```
# Инициализируем переменные
gamma = 0.5
ema = 0
ema_length = len(new_data)
ema_pred = [ema] # В этот список добавляем значения ЕМА
ema_x = [] # В этот список добавляем даты
ema_mse = [] # В этот список добавляем среднеквадратичные ошибки

# Циклом предсказываем значение для периода t+1 на основе периода t
for pred_by_ema in range(0, ema_length - 1):
    ema = ema * gamma + (1 - gamma) * new_data[pred_by_ema]
    ema_pred.append(ema)
    ema_mse.append((ema_pred[-1] - new_data[pred_by_ema + 1])**2)

    date = df.loc[pred_by_ema + 1, 'Date']
    ema_x.append(date)

# Выведем MSE для ЕМА-прогнозирования:
print(
    'MSE для предсказания ЕМА равно %.5f'%(
        np.mean(ema_mse)))
```

Для того, чтобы иметь возможность вручную оценить качество предсказания, визуализируем реальные и предсказанные с помощью ЕМА данные.

```
# По аналогии с предыдущим графиком
plt.figure(figsize=(18,9))
plt.plot(range(len(new_data)), new_data,
         color='Indigo', label='True')
plt.plot(range(0, ema_length), ema_pred,
         color='orange', label='Prediction')
plt.xlabel('Дата')
plt.ylabel('Средняя цена')
plt.legend(fontsize=18)
plt.show()
```

Из графика видно, что ЕМА очень качественно прогнозирует данные. Однако у такой модели есть свои недостатки - она может выдавать предсказания лишь на один шаг вперед. Если мы захотим, чтобы такая модель дала прогноз уже на два или более шагов, она выдаст то же значение, что и для предыдущего шага. Проверим:

$$x_{t+1} = EMA_t;$$

$$x_{t+2} = EMA_{t+1} = \gamma \cdot EMA_t + (1 - \gamma) \cdot x_{t+1} \Rightarrow x_{t+2} = \gamma \cdot EMA_t + (1 - \gamma) \cdot EMA_t =$$

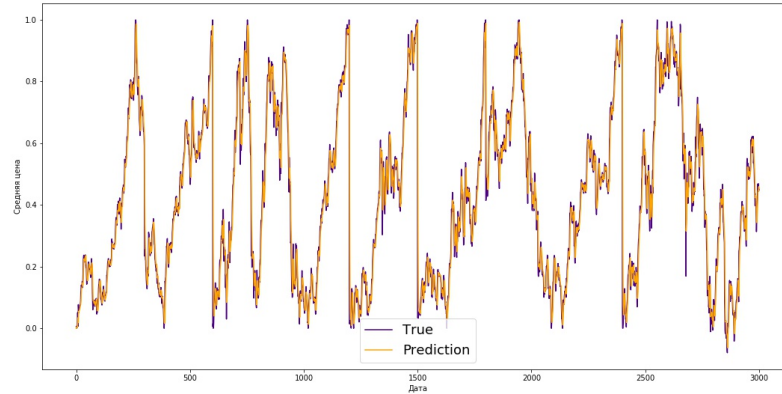


Рис. 13: График предсказаний методом ЕМА и истинных значений

$$EMA_t = x_{t+1}$$

Соответственно, для предсказания динамики показателей в долгосрочном периоде необходима более качественная модель - LSTM.

## 4.5 Модель LSTM

Прежде чем мы начнем создавать нашу модель в TensorFlow, необходимо создать тренировочную и тестовую выборки с целевыми переменными и требуемой формой для передачи их в будущую модель.

```
# Создаем тренировочную выборку и выводим
# размерность получившихся данных для проверки
def f_train_data(series, forecast, batch_size):
    x_data = series[-(batch_size + forecast)]
    x_final = x_data.reshape(-1, batch_size, 1)
    y_data = series[forecast:-(batch_size)]
    y_final = y_data.reshape(-1, batch_size, 1)
    return x_final, y_final

x_batches, y_batches = f_train_data(new_data, f_horizon, batch_size)

# Создаем тестовую выборку и выводим размерность
# получившихся данных для проверки
def f_test_data(series, forecast, batch_size):
    test_X = series[-(batch_size + forecast):-forecast].reshape(
        -1, batch_size, 1)
    test_Y = series[-batch_size:].reshape(-1, batch_size, 1)
    return test_X, test_Y

X_test, y_test = f_test_data(new_data, f_horizon, batch_size)
```

Приступим к созданию модели. Создадим пустой граф. Это действие становится обязательным, если мы планируем перезапускать нашу модель.

```
# Необходимое действие при многократном запуске модели
tf.reset_default_graph()
```

Плейсхолдеры (placeholders) - важные элементы в TensorFlow. Они хранят данные определенного типа и размерности, причем передать им эти данные можно позже. Объявим требуемые нам плейсхолдеры.

```
# Инициализируем плейсхолдеры с входными данными,
# истинными значениями для них и скоростью обучения
X = tf.placeholder(tf.float32, [None, batch_size, num_input])
y = tf.placeholder(tf.float32, [None, batch_size, num_output])
learning_rate = tf.placeholder(tf.float32, shape=[])
```

Теперь можно создавать нашу LSTM-сеть. Инициализируем скрытые слои и количество нейронов в каждом, а затем объединим их.

```
# Определяем три скрытых слоя в LSTM. Аргумент initializer
# приравниваем к указанному, потому что он помогает
# качественнее и быстрее настроить начальные веса
lstm_cells = [
    tf.contrib.rnn.LSTMCell(
        num_units=num_hidden[layer],
        state_is_tuple=True,
        initializer=tf.contrib.layers.xavier_initializer()
    )
    for layer in range(n_layers)]

# Настраиваем фильтры - забывания, входной и
# выходной в каждом слое
drop_lstm_cells = [tf.contrib.rnn.DropoutWrapper(
    cell,
    input_keep_prob=1.0,
    output_keep_prob=1.0 - dropout,
    state_keep_prob=1.0 - dropout
) for cell in lstm_cells]

# Объединяем слои
drop_multi_cell = tf.contrib.rnn.MultiRNNCell(drop_lstm_cells)
```

Все необходимые компоненты модели построены, нужно объявить переменные, ответственные за хранение состояния ячейки и скрытого слоя.

```
# Создаем переменные состояния ячейки и
# скрытого состояния, которые будут хранить
# состояние LSTM.
c, h = [], []
initial_state = []
for layer in range(n_layers):
    c.append(tf.Variable(tf.zeros([batch_size, num_hidden[layer]]),
        trainable=False))
    h.append(tf.Variable(tf.zeros([batch_size, num_hidden[layer]]),
```

```

                                trainable=False))
initial_state.append(
                                tf.contrib.rnn.LSTMStateTuple(
                                    c[layer], h[layer]))

```

Теперь, когда у нас созданы все необходимые переменные и элементы, объявляем саму модель.

```

# Создание непосредственно рекуррентной сети с заданными клетками
lstm_output, state = tf.nn.dynamic_rnn(
    drop_multi_cell, X, initial_state=tuple(initial_state),
    time_major=True, dtype=tf.float32)

```

Добавим линейный слой в нашу LSTM-сеть для правильной выходной формы данных.

```

# Приводим вывод сети к требуемому формату
lstm_output_new = tf.reshape(lstm_output, [-1, num_last])

# Добавляем линейный слой
# для правильного вывода данных
lstm_output_stacked = tf.layers.dense(lstm_output_new, num_output)

# Снова приводим данные к нужной форме
outputs = tf.reshape(lstm_output_stacked, [-1, batch_size, num_output])

```

Последним шагом перед запуском сети является расчет размер потерь. Для каждой пары "предсказание — истинное значение" вычисляем MSE, а затем суммируем все получившиеся значения ошибки. Затем определяем оптимизатор (optimizer) — AdamOptimizer, который использует метод обратного распространения, подробно описанный ранее и минимизируем ошибку с его помощью.

```

# Определяем функцию ошибки
loss = tf.losses.mean_squared_error(outputs, y)

# Инициализируем оптимизатор
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)

# Создаем переменную, которая будет минимизировать
# функцию ошибки с помощью данного оптимизатора
training_op = optimizer.minimize(loss)

```

Приступаем к запуску модели. Объявляем глобальные переменные.

```

# Инициализируем глобальные переменные
init = tf.global_variables_initializer()

```

Производим непосредственно запуск LSTM сети. Каждые 100 итераций будем выводить значение ошибки, чтобы убедиться, что оно монотонно снижается.

```

# Строим график с истинными и предсказанными значениями
plt.figure(figsize=(18, 9))

```



```
plt.plot(range(300), np.ravel(y_pred), c='b',
         linewidth=1, label='Предсказанные значения')
plt.plot(range(300), np.ravel(y_test), c='g',
         linewidth=1, label='Истинные значения')
plt.legend()
plt.show()
```

Визуализируем полученные предсказания.

```
# Строим график с истинными и предсказанными значениями
plt.figure(figsize=(18, 9))
plt.plot(range(300), np.ravel(y_pred), c='b',
         linewidth=1, label='Предсказанные значения')
plt.plot(range(300), np.ravel(y_test), c='g',
         linewidth=1, label='Истинные значения')
plt.legend()
plt.show()
```

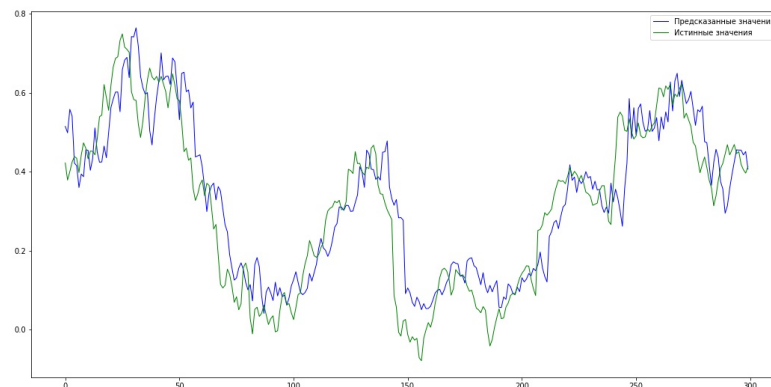


Рис. 14: График предсказаний LSTM и истинных значений

По графику можно сделать вывод, что полученная сеть не идеально прогнозирует данные. Однако она позволяет делать предсказания в любом, в том числе и долгосрочном периоде: для этого требуется всего лишь поменять значение переменной 'f horizon'. Кроме того, она может прогнозировать, как изменится направление тренда в перспективе - поднимется или опустится. Подобные свойства делают нашу сеть незаменимым оружием для людей, чья деятельность так или иначе связана с прогнозами — от игроков фондового рынка до работников в сфере консалтинга, от букмекеров до чиновников.

## 5 Заключение

Подводя итог, нейронные сети LSTM являются очень мощным инструментом предсказания, позволяющим анализировать будущие значения временных рядов в соответствии с их предыдущими значениями. Залогом качественной сети является, во-первых, репрезентативная обучающая выборка, демонстрирующая разнообразное поведение на всем периоде, а во-вторых — тщательный и корректный подбор гиперпараметров. Библиотека 'TensorFlow', в свою очередь, безусловно, очень сильно упрощает

задачу написания нейросетей благодаря широкому спектру доступных функций<sup>18</sup>. С развитием Tensorflow могут появляться новые функции и методы, которые будут способствовать повышению качества моделей. Нейронная сеть, представленная в работе, может быть использована для прогнозирования поведения макроэкономических показателей и допускает изменения и модернизацию для улучшения точности предсказаний.

---

<sup>18</sup>[8], TensorFlow library official site

## Список литературы

- [1] S.Haykin. Neural Networks and Learning Machines Second Rdition. Ontario, Canada: Prentice Hall, 2006. pp. 31-42, 89-90, 107, 230-240, 330, 921-925.
- [2] Головенко А.О. Прогнозирование экономических временных рядов с использованием нейросетевых технологий. Научно-аналитический экономический журнал, 2016. url: <http://sae-journal.ru/archives/284>.
- [3] A. M., K. M. Characterization of a class of sigmoid functions with applications to neural networks. 1996. pp. 819-835.
- [4] Clafiore G. Optimization Models. Cambridge University Press, 2014. pp.103-132.
- [5] Olah C. Understanding LSTM Networks. Colah's blog, 2015. url: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>.
- [6] Николенко С., Кадурин А., Архангельская Е. Глубокое обучение. СПб.: Питер, 2018. 480 с. Страницы 142-156, 231-248.
- [7] Викторович Будыльский Дмитрий. GRU и LSTM: современные рекуррентные нейронные сети. Казань: Научный журнал Молодой ученый, 2015. 3 с. Страницы 51-52.
- [8] Tensorflow library official site.