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 рекуррентных нейронных сетей для анализа макроэкономических показателей»

Студентка группы БЭК162
Сапельникова Мария Васильевна

Научный руководитель:
Старший преподаватель Демешев Борис Борисович

Москва 2018

Содержание

1 Введение

1.1 Мотивация

Произошедшая в начале 2000-х годов революция в машинном обучении, когда ученые из Университета Торонто научились обучать глубокие нейронные сети, послужила началом их активного применения для решения задач из самых разнообразных областей. Теперь перед исследователями открылись новые горизонты: появились алгоритмы, способные быстро, точно и эффективно предсказывать, классифицировать и моделировать.[1] С того момента ситуация частично изменилась. В современном мире существует множество задач, решение которых значительно бы упростилось с использованием машинного обучения. Игра на бирже, вычисление кредитоспособности компании, степень влияния внешних изменений на ключевые макроэкономические показатели, предсказание хода развития заболевания или какого-то химического процесса — каждая задача требует своего подхода, высокой точности и минимальной вероятности ошибки. По этой причине требуется непрерывное совершенствование методов машинного обучения, в частности разрешения многих возникающих в процессе разработки проблем.

1.2 О чем эта работа?

Основная цель нашей работы заключается в понимании базовых принципов работы нейронных сетей, в частности архитектуры RNN (Recurrent Neural Networks) и применении этих принципов на практике. Задачей будет прогнозирование временных рядов. Рассмотрим различные проблемы, возникающие при построении и обучении, а также попытаемся найти их наиболее оптимальное решение. В качестве источников используются учебники по машинному обучению, статьи, посвященные данной теме и онлайн-курсы для совершенствования (а иногда и приобретения) навыков программирования. Итогом работы станет инструкция по реализации рекуррентной нейронной сети для анализа временных рядов, дополненная необходимыми теоретическими сведениями. Предполагается, что читатель знаком с базовыми операциями программирования, курсом математического анализа и заинтересован в рассматриваемой теме.

2 Теоретическая часть

2.1 Общее представление о нейронных сетях

Прежде чем начать разговор о более узком понятии рекуррентных нейронных сетей, следует подробнее описать механизм обучения и работы нейронных сетей вообще. Как и для большинства изобретений человечества вдохновителем для их создания является природа, а именно человеческий мозг. До сих пор не до конца изученный, он невероятно быстро решает множество вопросов: «учит» новые языки, распознает звуки, генерирует речь и многое другое. С большинством задач мозг справляется гораздо лучше программ как по времени, так и по качеству. Нейронные сети, построенные по принципу человеческого мозга, призваны расширить спектр решаемых задач и увеличить производительность. В чем же заключается этот принцип? Мозг строится из специальных клеток — нейронов. Каждый нейрон формируется из тела

[1] [1], глава 1

клетки, коротких отростков — дендритов и длинных отростков — аксонов. Связи между дендритами и аксонами называются синапсами. см Рис.1[2]

Передаваемые по связям импульсы от одного нейрона к другому имеют разный характер, могут активировать следующий или наоборот снизить его активность. При этом нейроны отлично синхронизируются, достигая высокой степени параллелизации. Обобщенная схема работы человеческого мозга выглядит следующим образом: имеются связанные между собой клетки, каждая из которых находится в определенном состоянии, которое может быть изменено состоянием предыдущего. Одной из самых важных характеристик мозга считается пластичность — способность быстро «обучаться» и подстраиваться под изменения внешней среды. Именно этот принцип используется в нейронных сетях. При этом важно понимать, что основной задачей этих алгоритмов является не максимальное приближение к природному аналогу (это было бы слишком сложно, как минимум потому что мозг не до конца изучен), а несколько абстрактное применение общего принципа работы.

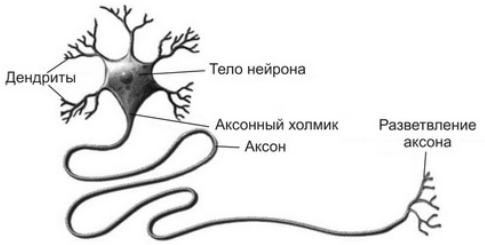

Рис. 1: Строение нейрона

2.2 Обучение нейронных сетей

Что значит «обучить» нейронную сеть? Обучение нейронных сетей происходит при помощи тренировочных данных, функции ошибки и алгоритма минимизации.Фактически, предсказания строятся с помощью учета предыдущих значений. Получаются матрицы весов, с которыми эти значения будут учитываться. В конечном итоге, мы получаем вектор предсказаний, те значения, которые построила модель. Чтобы узнать, насколько эти предсказания точны, необходимо вычислить разность между ними и истинными, затем минимизировать полученную функцию. В большинстве задач используются нелинейные функции ошибки, но так или иначе все они зависят от разность между предсказанием и истиной. Алгоритм, в роли которого в современном машинном обучении как правило выступает градиентный спуск, должен минимизировать функцию ошибки, зависящую от двух векторов: вектора предсказания и вектора истинных значений. Вследствие минимизации функции ошибки с помощью алгоритма происходит изменение весов и отклонений (или пороговых элементов) в модели.

Принцип работы этого метода будет описан в разделе «Проблемы рекуррентных нейронных сетей»

2.3 Рекуррентные нейронные сети

Основная область применения рекуррентных нейронных сетей — обработка различных последовательностей, где значение предыдущего элемента связано со следующим и каким-то образом влияет на него. Самым простым примером может стать предсказание последнего слова в предложении. Когда мы читаем предложение, его смысл постепенно становится понятнее с каждым прочитанным словом. То есть мозг не «удаляет» из памяти предыдущее прочитанное и «запоминает» новое, а «обновляет» значение всего предложения. Точно таким же образом действуют рекуррентные

[2]Источник: [2]

нейронные сети, учитывая все ранее произошедшие изменения и обновляя таким образом значение результата. Особенностью устройства рекуррентный нейронных сетей является наличие петель — связей нейрона с самим собой. Так формируются петли. Входящая последовательность разбивается по элементам, каждое внутреннее состояние получает на вход соответствующий элемент и результат предыдущего блока. Как видно на Рис.2 [3] входной вектор x_t обновляет состояние выходного вектора h_{t-1} до h_t.

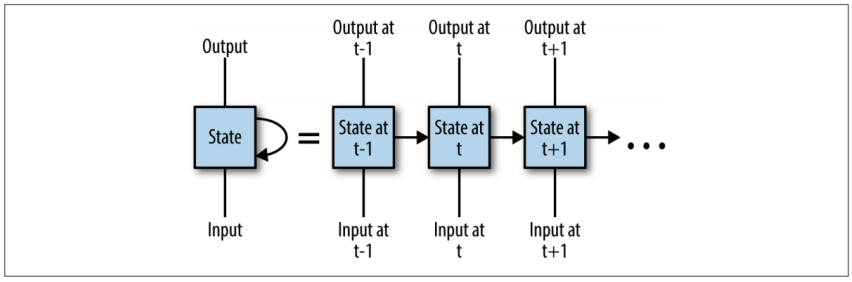

Рис. 2: Клетка рекуррентной нейронной сети

Так, мы можем развернуть любой узел рекуррентной нейронной сети в цепь, состояние которой обновляется с каждым шагом. Наша задача — описать этот шаг, а именно как каждое следующее «слово» влияет на смысл. Иными словами, нам нужно научить цепь «думать».[4] Разберем схему на примере ряда реального ВВП России с 2009 по 2011 год. Имеем входной вектор $x = (38807.2, 46308.5, 55967.2)$. Задача — предсказать значение ВВП в 2012 году. Соответственно, x_{t-1}, x_t и x_{t+1} будут элементами этого вектора, а требуемое предсказание — h_t. То, что происходит внутри каждого блока будет описано далее.

2.4 Математическая интерпретация происходящего

Опишем каждый блок рекуррентной нейронной сети[5]. Введем используемые обозначения:
W — матрица весов для перехода между слоями
U — матрица весов для входов
V — матрица весов для выходов
f — функция активации
h — функция для получения ответа s_t — состояние сети в момент времени t
b и c — отклонения
$x_t = (x_1, x_2, x_3)$ — вектор входящих значений
Тогда задача блока в момент времени t выглядит следующим образом:

$$a_t = b + Ws_{t-1} + Ux_t \tag{1}$$

$$o_t = c + Vs_t \tag{2}$$

$$s_t = f(a_t) \tag{3}$$

$$y_t = h(o_t) \tag{4}$$

[3]Источник: [3], стр. 77
[4]Источник:[3], стр. 77
[5][3], часть 2, глава 6

Поступающий на вход элемент x_t умножается на матрицу весов U, предыдущее состояние сети s_{t-1} — на матрицу весов W, добавляется свободный член (смещение, позволяющее «выровнять» значение)(1). Полученное выражение подставляется в функцию активации f, получается состояние сети в момент t — s_t(3). Далее это состояние передается на выход с весами V(2) и обрабатывается функцией h, приводящей полученный результат к требуемому задачей виду(4)и в следующий блок с весами W.
Отдельное внимание стоит уделить выбору функции активации.
Рассмотрим четыре самых часто используемых в машинном обучении: сигмоид (σ), гиперболический тангенс ($tanh$), ступенчатая функция и ReLU.

- Сигмоид: $\sigma = \dfrac{1}{1+e^x}$, Рис.3

  Основные свойства: дифференцируемая на всей области значений, при $x \to -\infty$ стремится к 0, при x $\to +\infty$ стремится к 1. Существенным недостатком является высокая скорость стремления к крайним точкам, что приводим к быстрому затуханию градиента при обучении. Важность этой проблемы станет более понятной в следующем разделе.

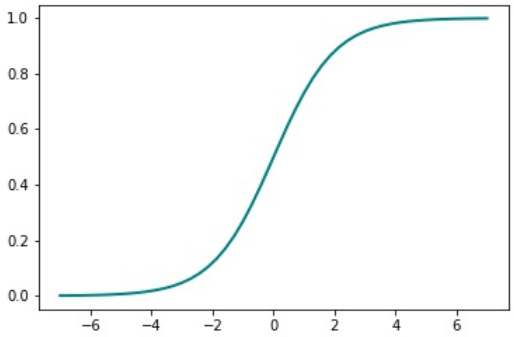

Рис. 3: Сигмоид

- Гиперболический тангенс: $tanh = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$, Рис.4

  Основные свойства: во многом похож на сигмоид, везде дифференцируемый, при $x \to -\infty$ стремится к -1, при x $\to +\infty$ стремится к 1. Скорость стремления к крайним точкам еще выше, однако среди них нет 0, который обнулял бы общий градиент при обучении. А значит, использование этой функции более разумно. Помимо этого, 0 является центральной точкой, то есть можно начать обучение в этой точек, смещаясь в любую сторону.

- Ступенчатая функция (функция Хевисайда), Рис.5 : $\begin{cases} 0, & x < 0 \\ 1, & x > 0 \end{cases}$

  Основные свойства: не определена в нуле, можем доопределить, получим грубое приближение вышеописанных, тогда сможем дифференцировать. Проблема совпадает с сигмоидом — нулевое крайнее значение обнуляет общий градиент при обучении.

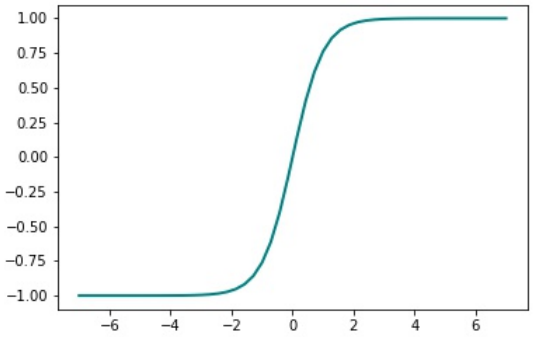

Рис. 4: Гиперболический тангенс

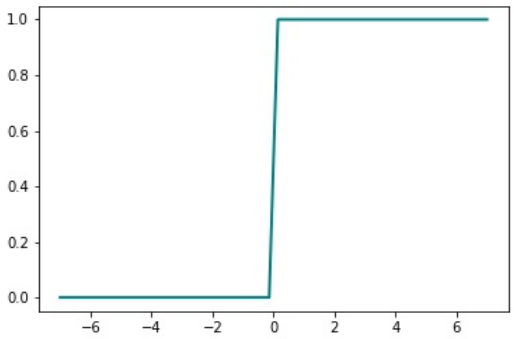

Рис. 5: Ступенчатая функция

- ReLU, Рис.6: $\begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$
  Основные свойства: Простая в дифференцировании, сокращает время на вычисления. Интересно, что именно эта функция наиболее схожа с тем, что происходит в человеческом мозгу [6]. В указанном источнике можно подробно прочитать про нее, мы же не будем вдаваться в глубокие математические подробности.

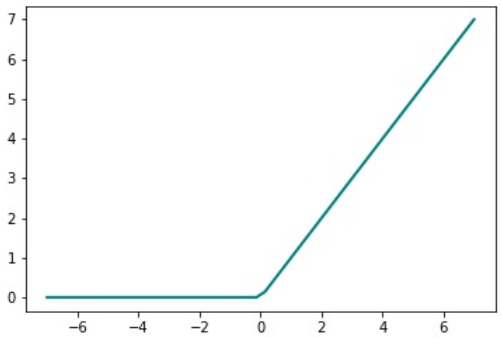

Рис. 6: ReLU

В своей работе я буду использовать функцию гиперболического тангенса, так как она наиболее понятная для примеров и не вызывает серьезных проблем при обучении.

[6] [1], часть 1, раздел 3.3

2.5 Проблемы рекуррентных нейронных сетей

Основным алгоритмом, используемым в обучении нейронных сетей, является метод градиентного спуска. Как известно, градиент функции – вектор, демонстрирующий направление её наискорейшего роста. Соответственно, если взять антиградиент (все элементы вектора-градиента с противоположным знаком), можно узнать направление наискорейшего убывания функции. В задачах обучения, как правило, минимизируется функция ошибки, то есть величина от разности истинного значения и полученного предсказания. Если мы будем перемещаться в направлении антиградиента в каждой точке функции ошибка с каким-то шагом, то рано или поздно достигнем её глобального минимума. Так как шаг является настраиваемым параметром, с ним связаны основные проблемы данного метода. Что будет, если шаг будет выбран слишком маленьким? Слишком большим? Появляется возможность либо «не дойти» до глобального минимума и «застрять» в локальном, либо «перепрыгнуть» его, продвинувшись слишком далеко. Природа проблем с обучением рекуррентных нейронных сетей несколько схожа с вышеописанным принципом, но прежде всего возникает вопрос об обучении сетей с такой архитектурой. Не будем вдаваться в подробности метода обратного распространения ошибки, так как он описан в большом количестве статей (источники будут указаны в конце работы). Изучив принцип работы этого метода, необходимо разобраться каким образом передавать значение градиента ошибки от одного нейрона к другому, если они соединены сами с собой? Давайте выпишем выходное значение сети в какой-то момент времени t, используя ранее описанные формулы(1) – (4), но без матриц весов и смещений, чтобы не загромождать запись. Пусть $t = 3$ (для простоты).
$y_3 = h(x_3, s_2) = h(x_3, f(x_2, s_1)) = h(x_3, f(x_2, f(x_1, s_0)))$
s_0 - начальное состояние сети

Теперь стало понятно, каким образом считать градиент и передавать его значение дальше, а также почему рассматриваемая архитектура называется "рекуррентной". Так как матрицы весов W и U в каждом блоке одинаковы, а градиент умножается на них с каждым шагом и растет экспоненциально, может возникнуть две основных проблемы: «взрывающийся» и «затухающий» градиент.

Проблема «взрывающегося» градиента»

Если же матрица весов устроена таким образом, что при умножении на нее норма вектора-градиента возрастает, рано или поздно возникнет проблема «взрывающегося» градиента. Тогда с каждым следующим шагом в методе градиентного спуска норма градиента растет экспоненциально, и глобальный минимум функции ошибки не будет достигнут, как и в случае со слишком большим шагом. Решением в данном случае будет простое ограничение значения градиента извне, что просто не даст ему расти выше заданной границы.

Проблема «затухающего» градиента»

В простых нейронных сетях не учитывается взаимосвязь предыдущих элементов со следующими, поэтому факт того, что градиент затухает, не является проблемой. В рекуррентных особенно важно как можно дольше сохранить «контекст», то есть степень влияния предшествующих элементов ряда. Проблема заключается в том, что если матрица весов устроена таким образом, что при каждом умножении на нее норма вектора-градиента уменьшается экспоненциально, то в какой-то момент $t + i$ она и вовсе будет незначительной величиной. Здесь возникает проблема краткосрочной памяти. Таким образом, в какой-то момент времени t влияние вектора x_{t-i} будет аналогично незначительным или вовсе не будет учитываться при формиро-

вании предсказания. Вполне логично, что чем больше объем контекста, тем точнее будет предсказание последнего элемента, так что наша задача – сохранить влияние как можно большего числа элементов ряда.

Пока не вдаемся в подробности реализации, этому посвящена вся практическая часть работы. Решение проблемы «затухающего» градиента не такое простое. Существует несколько усовершенствованных архитектур (базовым принципом остаются рекуррентные сети), а именно модель долгой краткосрочной памяти LSTM, GRU, SCRN и другие. Рассмотрим каждую из них более подробно в следующем разделе.

3 Практическая часть. Пробы и ошибки.

3.1 Что будем делать?

Наша задача — реализовать обыкновенную рекуррентную нейронную сети на языке Python. Выделим основные этапы:

- выбор и установка библиотек
- выбор, подготовка и визуализация данных для обучения
- инициализация модели RNN
- обучение модели
- оценка качества работы модели
- настройка гиперпараметров

Опишем каждый шаг подробнее.

3.2 Выбор и установка библиотек

На сегодняшний день существует несколько удобных библиотек для машинного обучения. Все они направлены на упрощение и ускорение решений задач машинного обучения, предоставляя чаще всего используемые готовые функции и методы. Благодаря им можно уместить целый алгоритм в несколько строк кода, а это заметно уменьшает затрачиваемые время и силы.

Мы будем использовать TensorFlow, библиотеку, разработанную в Google. Основными принципом её работы является построение вычислительных графов, с их помощью представляются все операции. Не будем подробно описывать принцип работы, так как существует множество пособий, среди которых отличный курс от Google Developers.

Для всех вычислительных операций и работы с TensorFlow необходимы векторы, с этим нам поможет библиотека NumPy (Numerous Python) — мощный инструмент, используемый для решения огромного количества задач.

Для чтения файла с данными и построения таблицы с данными, а также для их первичного анализа используем Pandas.

Чтобы лучше понимать данные, с которыми мы будем работать и отслеживать изменения качества работы модели понадобится визуализация — пакет PyPlot из библиотеки Matplotlib.

Из библиотеки SciKitLearn импортируем функцию для нормализации MinMaxScaler, которая приравнивает максимальное значение по выборке к 1, а минимальное — к нулю. Будем использовать её для нормировки.

Считаем, что все библиотеки установлены (как это делается можно найти тут), а нам остается просто их подключить.

```
import tensorflow as tf
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.preprocessing import MinMaxScaler
```

3.3 Выбор, подготовка и визуализация данных

Так как мы хотим научить нашу сеть прогнозировать временные ряды, нам необходимо обучать её на таких временных рядах, которые бы демонстрировали разное поведение. Для этого хорошо подходит набор данных M4, будем использовать годовые данные по макроэкономическим показателям.

```
# Читаем файл.
data = pd.read_csv('Yearly-train.csv')
# Удаляем строки, в которых содержатся значения NaN.
data = data.dropna()
# Обрежем таблицу так, чтобы остались только нужному нам столбцы.
data = np.array(data.iloc[:,1:832])
# Приводим к нужному виду (1, n).
data.reshape(1, -1)
```

В результате этих действий мы получаем одну строку размера (1, 831). Изобразим её (реализации этого в работе приводить не будем, в приложении будет весь получившийся код целиком).

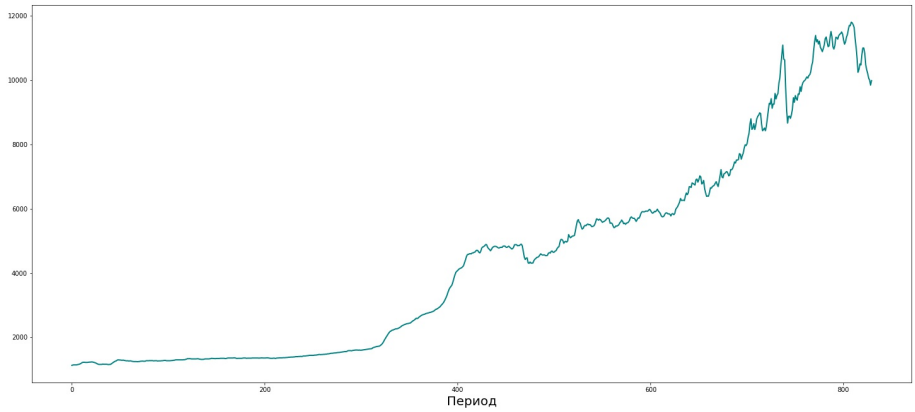

Рис. 7: Визуализация полученных данных

Как видно на графике, данные отлично подходят для обучения модели, так как выделяются четыре периода с разным характером поведения (от 0 до 200, от 200 до 400, от 400 до 600 и до конца).

Для обучения нам потребуется выборка с тренировочными и тестовыми данными. Разделим исходную выборку так, что в тренировочной будет 781 элементов, а остальные 50 — в тестовой.

```
# Для дальнейшей нормировки сразу приводим к виду (781, 1) и (50, 1).
train_data = data[:,:781].reshape(-1, 1)
test_data = data[:,781:].reshape(-1, 1)
```

И тестовую, и тренировочную выборки необходимо пронормировать по выбранному периоду, так как в значениях наблюдается слишком большой разброс: минимальное значение около 1000, а максимальное — 11807. Для этого будем использовать описанную выше функцию MinMaxScaler: обработаем ей разбитую на части по 200 элементов выборку для тренировки (не забудем часть, не вошедшую в разбиение) и для тестирования целиком.

```
scaler = MinMaxScaler()
# Определяем размер периода.
size = 200
for i in range(0, 781, _size):
        scaler.fit(train_data[el:i+size,:])
    train_data[i:i+_size,:] = scaler.transform(train_data[i:i+size,:])
# Обрабатываем оставшуюся часть выборки.
train_data[i:i+size] = scaler.fit_transform(train_data[i:i+size])
# Каждый раз настраиваем функцию по новой выборке.
scaler.fit(test_data)
test_data = scaler.transform(test_data)
# Приводим полученный результат к нужному виду (1, 50, 1).
# Далее будет понятно, почему.
test_data = test_data.reshape(-1, 50, 1)
```

Теперь, если мы визуализируем полученный результат, получится следующая картина:

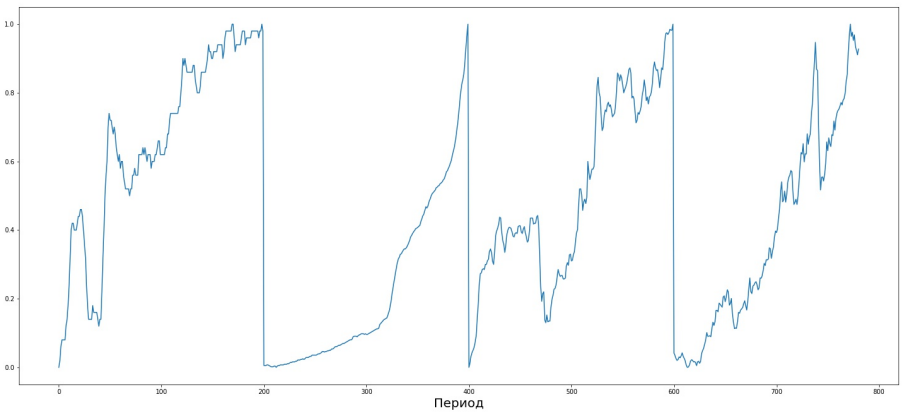

Рис. 8: Визуализация нормированных данных

Для эффективного обучения модели разделим тренировочные данные на меньшие части, так как если настраивать модель на слишком большом объеме, как такового обучения не произойдет — мы предскажем только одно значение. Оценивать качество по такому результату невозможно. Поэтому разделим всю выборку на подвыборки по 50 элементов. Заметим, что столько же значений в тестовой выборке.

Для удобства в дальнейшем использовании настраиваем форму (None, 50, 1). Значение «None» указывает на то, что мы не знаем, сколько у нас получится строк.

```
# Количество элементов в одной подвыборке.
num_periods = 50
# Сколько значений "вперед" мы предсказываем.
step = 1
# Приводим всю выборку к одномерному массиву.
train_data = train_data.reshape(-1)
# Подвыборки для входных векторов.
x_data = train_data[:(len(train_data)-(len(train_data) % num_periods))]
x_batches = x_data.reshape(-1, 50, 1)
# Подвыборки для оценки качества (целевые значения).
y_data = train_data[1:(len(train_data)-(len(train_data) % num_periods))+step]
y_batches = y_data.reshape(-1, 50, 1)
```

3.4 Инициализация модели RNN

Определим необходимые параметры и переменные.

```
# Количество векторов на вход.
inputs = 1
# Количество клеток в каждом нейроне.
hidden_layers = 100
# Количесвто векторов на выход.
output = 1
x = tf.placeholder(tf.float32, [None, num_periods, inputs])
y = tf.placeholder(tf.float32, [None, num_periods, output])
```

Ключевой момент в модели — её инициализация. Будем использовать готовые функции TensorFlow, с их документацией можно ознакомиться по ссылкам. Отметим, что по умолчанию BasicRNNCell использует в качестве функции активации гиперболический тангенс *tanh*. Специально определим её, так как в работе с гиперпараметрами стоит рассмотреть зависимость качества от выбранной функции. BasicRNNCell создает одну клетку нейрона, dynamic rnn формирует нейросеть целиком.

```
basic_cell = tf.nn.rnn_cell.BasicRNNCell(num_units = hidden_layers,
activation = tf.nn.tanh)
rnn_output, previous_state = tf.nn.dynamic_rnn(basic_cell, x,
dtype=tf.float32)
```

Важным моментом является формирование и настройка формата выходного вектора.

```
stacked_rnn_output = tf.reshape(rnn_output, [-1, hidden_layers])
stacked_outputs = tf.layers.dense(stacked_rnn_output, output)
# Финальный результат нужной формы (None, 50, 1).
outputs = tf.reshape(stacked_outputs, [-1, num_periods, output])
```

3.5 Обучение модели

Определим шаг градиентного спуска. В разделе с настройкой гиперпараметров рассмотрим, как его значение влияет на качество предсказаний.

```
learning_rate = 0.0001
```

Для настройки будем пользоваться готовым оптимизатором AdamOptimizer, который работает по принципу обратного распространения ошибки через градиентный спуск. Подробнее о нем можно почитать тут.

Функцию ошибки определим как среднеквадратичную (MSE, Mean Squared Error) $\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$, где n - количество предсказаний, y_i - истинные значения, а $\hat{y}_i$ - предсказанные значения. Задача оптимизатора — её минимизация.

```
# Функция ошибки.
# Усредняем квадратичную разность предсказаний и реальных.
# Делим на второй элемент размерности выходного вектора (количество столбцов).
loss = tf.reduce_sum(tf.square(outputs - y))/int(outputs.shape[1])
# Инициализация оптимизатора и его задачи.
optimizer = tf.train.AdamOptimizer(learning_rate = learning_rate)
train_op = optimizer.minimize(loss)
```

3.6 Оценка качества работы модели

В первую очередь объявим все требующиеся переменные и запустим вычислительный граф.

```
init = tf.global_variables_initializer()
# Количество итераций.
epochs = 1000
with tf.Session() as sess:
    init.run()
    for ep in range(epochs):
        sess.run(train_op, feed_dict={x: x_batches, y: y_batches})
        if ep % 100 == 0:
            mse = loss.eval(feed_dict={x: x_batches, y: y_batches})
            print (ep, "\tMSE:", mse)
    y_pred = sess.run(outputs, feed_dict={x: test_data})
```

Значения ошибок для одного из запусков модели:

```
0           MSE: 1.7926495
100         MSE: 0.1196886
200         MSE: 0.075741634
300         MSE: 0.053305633
400         MSE: 0.044808585
500         MSE: 0.048330028
600         MSE: 0.043623384
700         MSE: 0.03213477
800         MSE: 0.029414654
900         MSE: 0.031827863
```

Визуализация полученных предсказаний (красным цветом изобразим полученные предсказания, синим — реальные значения тестовой выборки)

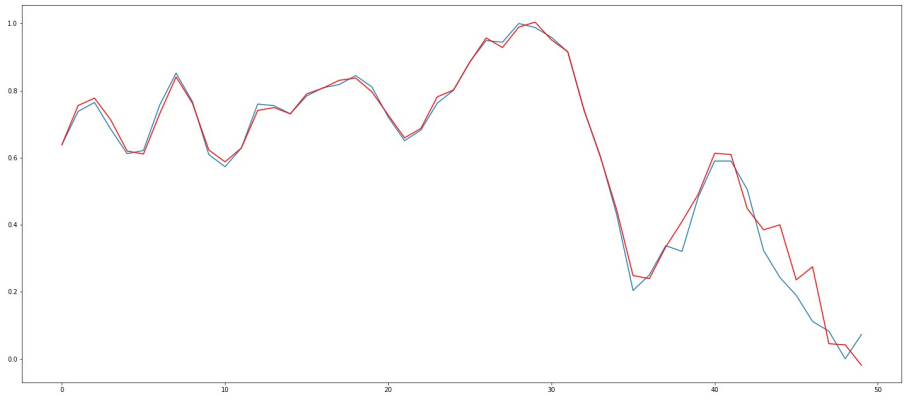

Рис. 9: Визуализация полученных предсказаний, $learningrate = 0.001, f = tanh$

В целом, модель работает достаточно качественно, не считая ошибок в последнем периоде. Следует иметь в виду то, что объем нашей выборки сравнительно небольшой, а построенная модель несколько грубая, так что такие отклонения можно считать приемлемыми. Тем не менее, постараемся улучшить картину дальнейшими манипуляциями.

3.7 Настройка гиперпарметров

В первую очередь уточним, что такое гиперпараметр. Так называются значения, от варьирования которых изменяется качество работы модели. Подбирается методом проб и ошибок. В нашем случае это:

- функция активации (activation)
- шаг в методе градиентного спуска (learning rate)
- количество клеток в каждом нейроне (hidden layers)

Рассмотрим, что будет с качеством при изменении каждого из них.
Заменим функцию активации на $ReLU$.
Посмотрим на значения MSE:

```
0           MSE: 4.592088
100         MSE: 0.08316497
200         MSE: 0.068788625
300         MSE: 0.06168739
400         MSE: 0.057213753
500         MSE: 0.053298
600         MSE: 0.06333121
700         MSE: 0.058019985
800         MSE: 0.037902087
900         MSE: 0.02434863
```

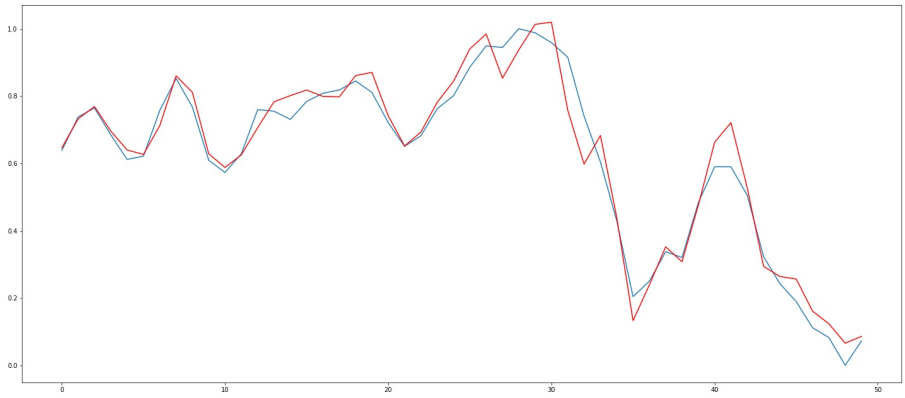

Рис. 10: Визуализация полученных предсказаний, learning rate = 0.001, $f = ReLU$

Полученные предсказания получились лучше, прогнозируемое поведение сильнее совпадает с реальным, ошибка меньше. Далее попробуем изменить шаг градиентного спуска, используя сначала $ReLU$, а потом $tanh$.

Увеличим, а затем уменьшим шаг градиентного спуска. Обратим внимание на значение MSE — если оно будет в какой-то момент колебаться, мы столкнемся с проблемой, когда минимум «перепрыгивается», если значение MSE «застывает» — градиент «застрял» в локальном минимуме:

Значения среднеквадратичной ошибки для learning rate = 0.00001 $f = ReLU$

```
0           MSE: 3.3546538
100         MSE: 2.9543178
200         MSE: 2.5492237
300         MSE: 2.118637
400         MSE: 1.5398437
500         MSE: 0.6524048
600         MSE: 0.34827211
700         MSE: 0.31749552
800         MSE: 0.29518446
900         MSE: 0.27794966
```

Значения ошибки выросли практически в 10 раз, а график (Рис. 11) демонстрирует, что полученная модель достаточно грубая.

Проверим, настолько ли сильно влияет уменьшение шага при использовании другой функции активации. Рассмотрим значения среднеквадратичной ошибки для learning rate = 0.00001, $f = tanh$:

```
0           MSE: 2.263204
100         MSE: 0.44168743
200         MSE: 0.22856939
300         MSE: 0.19207193
400         MSE: 0.1704937
500         MSE: 0.15704815
600         MSE: 0.14776282
700         MSE: 0.14076014
800         MSE: 0.1351383
```

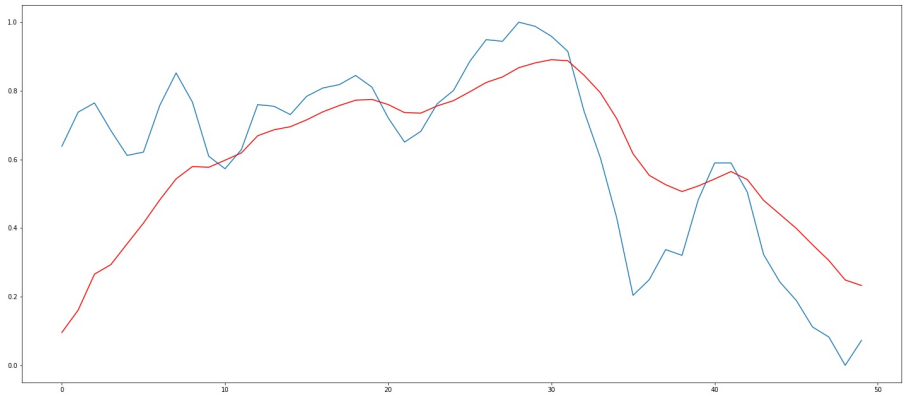

Рис. 11: Визуализация полученных предсказаний, learning rate = 0.00001, $f = ReLU$

```
900         MSE: 0.13038883
```

Аналогично с предыдущим результатом значения ошибки сильно выросли. Стоит отметить, что обучаться наша модель стала медленнее (об этом говорит изменение ошибки на каждом шаге). Вполне возможно, мы «застряли» в локальном минимуме. График подтверждает нашу гипотезу о неэффективности уменьшения шага (Рис. 12).

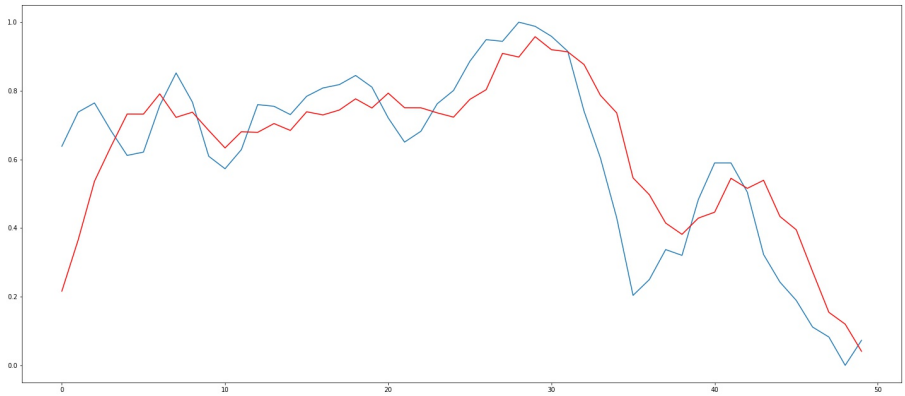

Рис. 12: Визуализация полученных предсказаний, learning rate = 0.00001, $f = tanh$

Пойдем по другому пути и увеличим значение learning rate до 0.01. Проверим эффективность при использовании функции активации ReLU:

Значения среднеквадратичной ошибки для Learning rate = 0.01 и $f = ReLU$

```
0           MSE: 36.208885
100         MSE: 0.083905205
200         MSE: 0.06306617
300         MSE: 0.054982495
400         MSE: 0.060820855
500         MSE: 0.06665602
600         MSE: 0.050509673
```

```
700          MSE: 0.0509433
800          MSE: 0.044805717
900          MSE: 0.044100452
```

Ошибка стала меньше, чем в прошлом эксперименте, однако немного выше, чем в самом первого варианта с learning rate = 0.001. Однако на графике (Рис.13) явно заметно улучшение: предсказанные значения чаще совпадают с реальными.

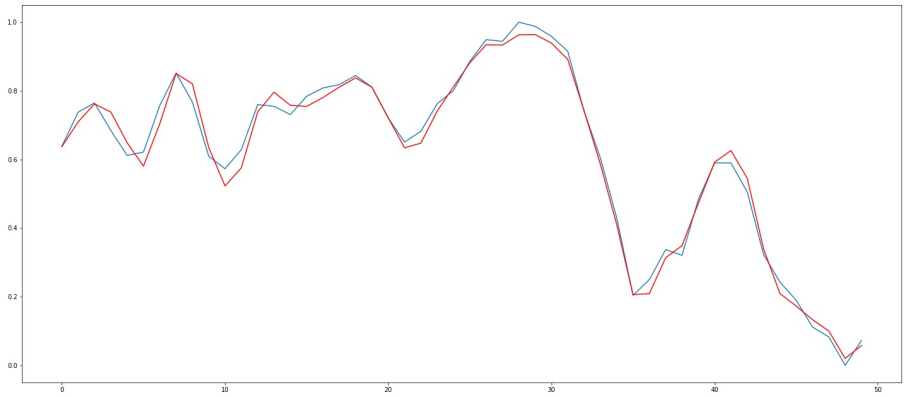

Рис. 13: Визуализация полученных предсказаний, learning rate = 0.01, $f = ReLU$

Для полноты проводимого опыта рассмотрим результат при использовании функции активации tanh:

Значения среднеквадратичной ошибки для Learning rate = 0.01 и $f = tanh$

```
0            MSE: 0.27224845
100          MSE: 0.07114297
200          MSE: 0.06425515
300          MSE: 0.061823256
400          MSE: 0.05978895
500          MSE: 0.064860746
600          MSE: 0.057521738
700          MSE: 0.060832363
800          MSE: 0.05663346
900          MSE: 0.058352303
```

Ошибка не дает явного ответа на вопрос об улучшении качества, но совсем немного хуже предыдущей. А график (Рис. 14) демонстрирует немного большую неточность.

Остановим наш выбор на функции активации $ReLU$ и шаге градиентного спуска 0.01. Это не означает, что такие параметры стоит выбирать всегда — все зависит от данных и модели.

4 Вывод

Как итог проделанной работы еще раз выделим основные результаты. Мы ознакомились с теоретическими понятиями, связанными с рекурреннтынми нейронными

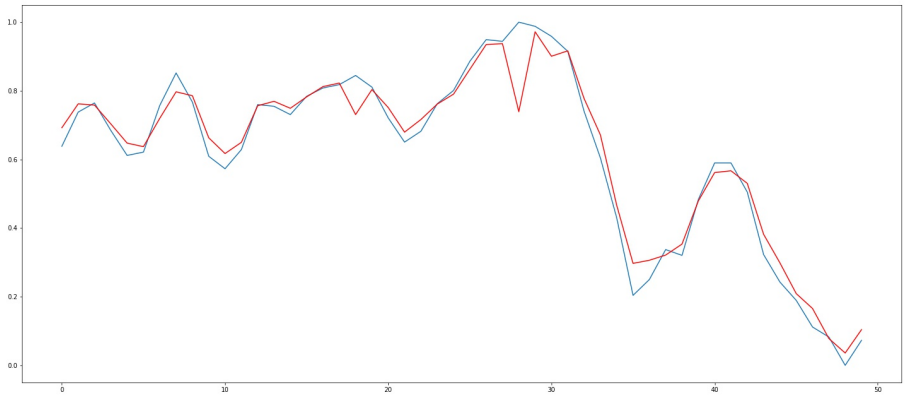

Рис. 14: Визуализация полученных предсказаний, learning rate = 0.01, $f = tanh$

сетями, разобрались с основой каждого нейрона (функции активации), а также построили собственную простую модель, предсказывающую временной ряд. Дальнейшей деятельностью по этим направлениям может быть разнообразное улучшение полученной модели. Применение LSTM, GRU, анализ текстовых последовательностей и многое другое может стать предметом изучения и позволит улучшить полученный результат.

Весь код написанной сети и файл с данными можно найти тут.

Список литературы

[1] Николенко С., Кадурин А., Архангельская Е. Глубокое обучение. СПб.: Питер, 2018. 480 с.

[2] urlhttp://www.grandars.ru/college/medicina/neyrony-mozga.html.

[3] Hope T., Resheff Y. S., Lieder. I. Learning TensorFlow. USA: O'Reilly, 2017. 221 p.