ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

Национальный исследовательский университет «Высшая школа экономики»

Факультет экономических наук
Образовательная программа «Экономика»

КУРСОВАЯ РАБОТА

«Рекуррентная нейронная сеть LSTM. Общее устройство и применение для анализа макроэкономических рядов»

Студент группы БЭК162
Зехов Матвей Сергеевич

Научный руководитель:
Демешев Борис Борисович

Москва 2018

Содержание

1 Введение

Сети долгой краткосрочной памяти(Long Short-Term Memory) представляют собой специальный подкласс архитектур рекуррентных нейронных сетей, предназначенных для обработки различных форм последовательностей. Они были представлены Зеппом Хохрайтером и Юргеном Шмидхубером в 1997 году. Наиболее яркими примерами применения таких сетей являются проекты по обработке текстов и временных рядов. Последнему аспекту и посвящена данная работа. Предполагаемые ключевые этапы:

1. Теоретический принцип работы модели
2. Реализация модели через библиотеку Tensorflow для языка Python, используя методы расширяющегося и сдвигающегося окна
3. Подбор оптимальных гиперпараметров модели
4. Сравнение методов по ключевым характеристикам
5. Описание ключевых тонкостей реализации в главе "Грабли"
6. Дополнительная реализация модели через библиотеку Keras для языка R

В качестве рабочих данных будет использован датасет M4, содержащий в себе различные временные ряды. Скачать его можно здесь.

Под вышеупомянутом методом расширяющегося окна подразумевается следующее. Пусть ряд разбит на две части, train и test:

$$train \rightarrow \boxed{x_1, x_2, x_3, x_4,} \boxed{x_5, x_6, x_7} \leftarrow test$$

Получив на основе части train предсказание относительно элемента x_5, сравним его с маркером x_5. После этого перенесём элемент x_5 из test в train:

$$train \rightarrow \boxed{x_1, x_2, x_3, x_4, x_5,} \boxed{x_6, x_7} \leftarrow test$$

Далее процесс продолжается до окончания ряда.

Метод сдвигающегося окна подразумевает более распространённый принцип реализации. Допустим, сначала мы на основе первых трёх элементов ряда предсказываем четвёртый. Иными словами при ширине окна 4 предсказываем его последний элемент:

$$\boxed{x_1, x_2, x_3, x_4,}\boxed{x_5, x_6, x_7}$$

Далее окно просто сдвигается на одно значение, и итерация повторяется:

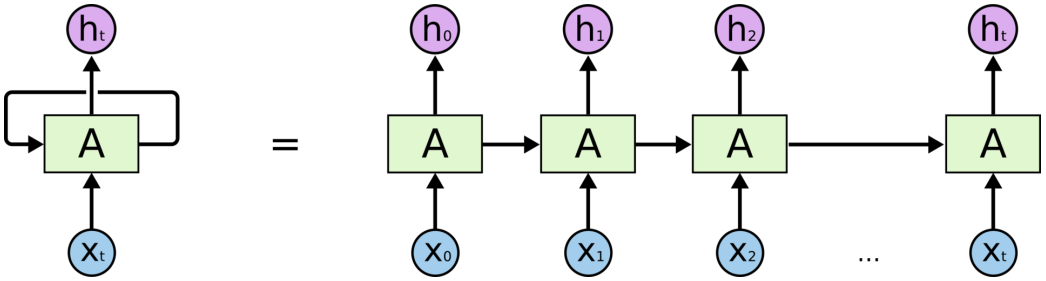
$$\boxed{x_1,}\boxed{x_2, x_3, x_4, x_5,}\boxed{x_6, x_7}$$

Таким образом, создадим таблицу из окон, где последний столбец будет означать маркеры, а все остальные - тренировочные данные. На указанных принципах и будут построены впоследствии модели.

2 Теоретический принцип работы модели

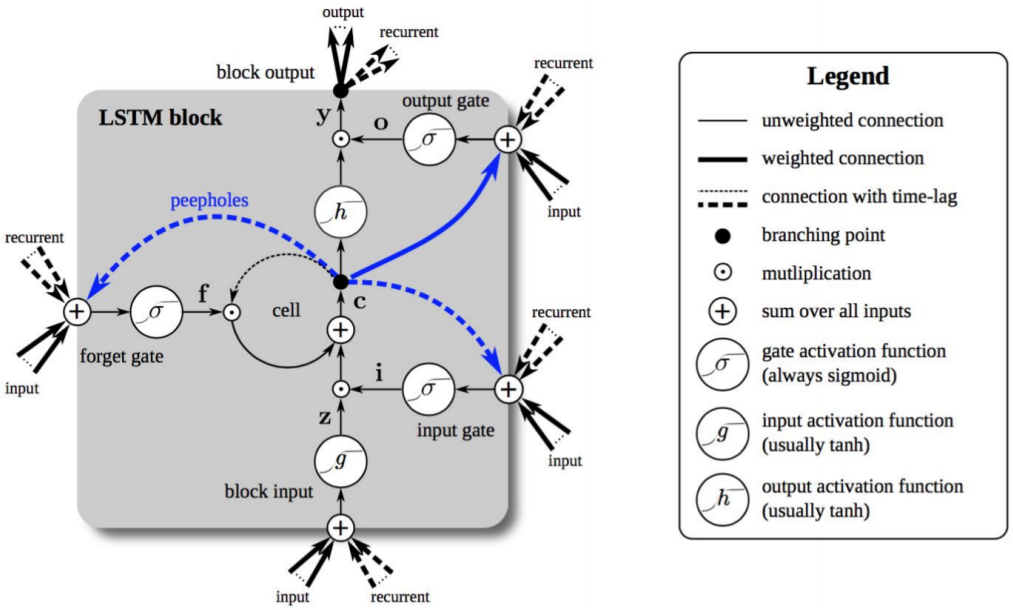
Не секрет, что для обработки долговременных зависимостей классические RNN модели слабо подходят, так как их архитектура не подразумевает какого-либо вида памяти. Структура LSTM решает эту проблему.

Для начала вспомним общий принцип работы рекуррентных сетей. Как известно, получая на вход значение, они строят прогноз, учитывая также своё предыдущее выходное значение. Иллюстрацию этого процесса можно увидеть на рисунке ниже.



В том случае, когда расстояние между актуальной информацией и местом, где она пригодилась, невелико, RNN вполне адекватно справляются с задачей, однако при увеличении расстояния эта связь теряется.

Разберёмся, что же из себя представляет стандартный блок LSTM. Ключевым понятием для построения блока является состояние ячейки. Этот вектор проходит через все итерации внутри клетки и несёт в себе информацию о всей последовательности в целом. Именно этот элемент обеспечивает восстановление протяжённых зависимостей. Разберём пошагово схему работы блока LSTM, и предназначение этого вектора станет очевидным.



Так как данные вектора состояния клетки проходят её насквозь, очевидно, что внутри неё он проходит некоторую обработку. Для этого существую четыре так называемых фильтра.

Первый фильтр называется фильтром забывания. На схеме он обозначен как forget gate. Данные, которые он производит на выходе, считаются по следующей формуле:

$$f^t = \sigma(W_f x^t + R_f y^{t-1} + b_f) \tag{1}$$

То есть, на основе входных данных и ответа с предыдущей итерации фильтр выдаёт значения от нуля до единицы и перемножает с вектором состояния поточечно. Идейно это означает, что фильтр указывает, какую часть вектора-состояния следует забыть, а какую оставить. То есть единица означает, что конкретный элемент следует полностью сохранить, а ноль - полностью забыть.

Следующие два фильтра решают, какую информацию необходимо добавить в состояние ячейки. Сначала сигмоидальный слой, он же "слой входного фильтра"(на рисунке - input gate), решает, какие значения следует обновить. По аналогии с фильтром забывания, он возвращает значения от нуля до единицы:

$$i^t = \sigma(W_i x^t + R_i y^{t-1} + b_i) \tag{2}$$

Одновременно с этим tanh-слой(на рисунке - block input) генерирует новые значения-кандидаты:

$$z^t = \sigma(W_z x^t + R_z y^{t-1} + b_z) \tag{3}$$

Соответственно, новое состояне клетки C^t будет определяться следующей формулой:

$$C^t = f^t \odot C^{t-1} \oplus i^t \odot z^t \tag{4}$$

Далее одна копия вектора состояния уходит для последующей рекурсии, а вторая участвует в генерации выходных данных клетки.

Чтобы понять, какую информацию выводить, входные данные и выход предыдущей итерации проходят через четвёртый сигмоидальный фильтр(на рисунке - output gate). Он решает, какую информацию необходимо выводить:

$$o^t = \sigma(W_o x^t + R_o y^{t-1} + b_o) \tag{5}$$

Параллельно с этим вектор состояния проходит через tanh-слой и перемножается с выходом сигмоидального слоя, что позволяет выводить только требуемую информацию:

$$y^t = o^t \odot tanh(C^t) \tag{6}$$

3 Построение модели на Python, библиотека Tensorflow

3.1 Библиотеки и гиперпараметры

Первым этапом построения модели является импортирование всех необходимых библиотек. Для работы необходимы следующие пакеты:

```
import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

С помощью библиотеки Tensorflow будут заданы основные переменные модели, клетка нейрона и некоторые прочие детали. Библиотека NumPy понадобится для создания некоторых полезных функций, преобразующих данные. Библиотека Pandas будет применяться для высокоскоростной работы с данными большой размерности. Библиотека Matplotlib понадобится в случае необходимости визуализировать какие-либо данные (например, изменение ошибки на обучении). Подробнее об установке Tensorflow и сопуствтвующих проблемах можно узнать в главе Грабли 8.1.

Следующим этапом является задание параметров модели.

```
num_classes = 1
hidden_layer_size = 30
batch_size = 1
sample_size = 1
sequence_length = 1
num_LSTM_layers = 30
learning_rate = 0.01
```

В данном случае $num\_classes$ определяет размерность выходного слоя модели. В нашем примере решается задача регрессии, поэтому выход будет одномерным. В случае, если решалась бы задача классификации, как, например, на известном датасете MNIST, то количество классов выходного слоя было бы 10.

$hidden\_layer\_size$ обозначает количество нейронов в каждом из скрытых слоёв. Является гиперпараметром, подбор которого будет описан в главе "Подбор оптимальных параметров"на стр. 19

sample_size является параметром, который регулироет объём нашей выборки, то есть то количество первых рядов(строк), которые отсекутся от исходного датесета.

batch_size служит для определения количества рандомно выбранных элементов нашей выборки, на которых будет происходить обучение модели на одной итерации. И sample_size и batch_size для простоты в данном примере равны единице.

num_LSTM_layers предначен для определения количества скрытых слоёв нейронной сети. Этот гиперпараметр также будет подбираться в главе "Подбор оптимальных параметров"на стр. 19

learning_rate - параметр обучения, шаг градиентного спуска оптимизатора на каждой итерации.

3.2 Специальные функции

В данном разделе представлены некоторые специальные функции для предварительной обработки входных данных, а также преобразования выходных данных модели. На этом этапе построения модели не обязательно вникать в принципы работы данных функций, так как все пояснения со ссылками будут даны по мере применения непосредственно в коде. Суть расположения данного раздела в последовательности представления полного кода модели.

```
def norm_array(array):
        mean = np.mean(array)
        stddev = np.sqrt(np.sum(np.square(array - mean)) / len(array - 1))
        normalized_array = ((array - mean) / stddev)
        return normalized_array,
                mean.reshape(batch_size),
                stddev.reshape(batch_size)

def test_decoder(array, mean, stddev):
        return (array * stddev) + mean

def get_linear_layer(vector):
        return(tf.matmul(vector, Wl) + bl)

def train_test_split(X, y, train_size):
        X_train = X[:int(len(X) // (1 / train_size))]
        y_train = y[:int(len(y) // (1 / train_size))]
        X_test = X[int(len(X) // (1 / train_size)):]
        y_test = y[int(len(y) // (1 / train_size)):]
        return X_train, X_test, y_train, y_test
```

```
\label{fun1}
def data_labels(data, sample_size, batch_size):
        labels = data.iloc[:sample_size, 2:]
        data = data.iloc[:sample_size, 1:]
        indeces = np.random.choice(len(data), size=batch_size, replace=False)
        batch_x = np.array(data.loc[indeces[0]] \
                                        [data.loc[indeces[0]][:] > 0][:-1])
        batch_y = np.array(labels.loc[indeces[0]] \
                                        [labels.loc[indeces[0]][:] > 0])
        return (batch_x, batch_y)

def cell_list(num_LSTM_layers):
        cell_list = []
        for i in range(num_LSTM_layers):
                cell_list.append(tf.contrib.rnn.BasicLSTMCell(
                                hidden_layer_size, \ forget_bias=1.0))
        return cell_list
```

3.3 Загрузка данных

Рассматриваемый датасет изначально был представлен в формате excel-таблицы, поэтому для работы в Pandas он был переформатирован в csv-файл. Это можно сделать, например, здесь. Далее прочитаем и преобразуем данные следующими командами:

```
data = pd.read_csv(r"C:\M4datasets\Yearly-train.csv")
data_x, data_y = data_labels(data,
                                sample_size = sample_size,
                                batch_size = batch_size)
```

Вторая строка использует одну из функций, представленных ранее:

```
def data_labels(data, sample_size, batch_size):
        labels = data.iloc[:sample_size, 2:]
        data = data.iloc[:sample_size, 1:]
        indeces = np.random.choice(len(data), size=batch_size, replace=False)
        batch_x = np.array(data.loc[indeces[0]] \
                        [data.loc[indeces[0]][:] > 0][:-1])
        batch_y = np.array(labels.loc[indeces[0]] \
                        [labels.loc[indeces[0]][:] > 0])
        return (batch_x, batch_y)
```

Данная функция преобразует данные в форму, которая подойдёт для дальнейшего применения в клетке LSTM. Первым этапом она обрезает датасет по строкам вплоть до размера $sample\_size$, при этом для входных рядов обрезает первые элементы, а для маркеров - первые два элемента. Это связано с тем, что в датасете M4 первым столбцом идёт список годов в буквенном выражении, и он для дальнейшего анализа не требуется. Второй элемент ряда маркеров также удаляется. Пояснить, зачем это необходимо, проще всего на примере. Допустим, мы подаём ряд на вычислительный алгоритм. Для каждого из элементов определяется прогноз, то есть, предположительный последующий элемент, который необходимо сравнить с маркером и определить ошибку. Тогда для прогноза на основе первого элемента маркером служит второй элемент, для второго элемента - третий и так далее.

Далее необходимо, чтобы из нашей выборки извлекалось заданное количество случайно взятых элементов(в нашем случае один элемент из выборки размерности 1). Для данного случая в этом мало смысла, но для масштабируемости и быстрого преобразования модели под другую схему обучения это критически важно. Соответственно, генерируются случайные индексы в количестве $batch\_size$. Аргумент $replace = False$ означает, что выбор происходит без возвращения. Далее по этим индексам извлекаются ряды.

Особенностью массива Pandas является то, что таблица значений содержит пустые значения типа Nan, эквивалентные нулям. Они компенсируют различия в длине разных рядов, и их необходимо убрать. Соответственно, у случайно выбранных рядов удаляются все нули путём сравнения всех элементов с нулём. В рамках конкретного датасета M4 это допустимо, так как он не содержит нулевых или отрицательных значений, однако в случае других наборов данных такая функция может потребовать внесения изменений. Помимо этого от входного ряда отрезается последний ненулевой элемент, так как для него не найдётся маркера. Таким образом, на выходе получаем два ряда одинаковой размерности. Для наглядности всё вышесказанное можно проиллюстрировать небольшим примером:

Исходный ряд:	12, 43, 53, 83, 23, Nan, Nan
Ряд входных данных:	12, 43, 53, 83
Ряд маркеров:	43, 53, 83, 23

3.4 Создание вычислительного графа

На данном этапе требуется сформировать основную вычислительную структуру библиотеки Tensorflow – вычислительный граф. Если вы не знаете, что это за структура, то можно подробно прочесть об этом в книге [1, Глава 3, стр. 23]

tf.reset_default_graph()

Данная строка обнулит вычислительный граф, созданный на предыдущем запуске. Подробнее узнать о том, что произойдёт, если ей пренебречь, можно прочесть в соответствующем разделе главы "Грабли". 8.2.

Далее добавим непосредственно элементы графа. Вопрос о том, прописывать ли scope, стоит не так остро, поэтому здесь он добавлен скорее для визуальной структуризации кода, нежели для практической пользы. Однако, ссылки на имя объекта всё же позволяют более эффективно читать логи ошибок при запуске кода.

```
with tf.name_scope("Training_data"):
        inputs = tf.placeholder(tf.float32,
                                shape=[batch_size, None,
                                sequence_length])
        y = tf.placeholder(tf.float32, shape=[batch_size])
```

В данных строках задаются ключевые тензоры, тензор входных данных и тензор маркеров. На них при запуске модели будут подаваться обработанные данные из датасета. Этот процесс будет подробно описан при рассмотрении запуска итерационного цикла. На данном этапе очень важно правильно соблюсти размерность тензоров. Касательно тензора *input*, если бы ряды, подаваемые на вход, были бы одинаковой длины, то вместо значения None в параметре shape должно было быть прописано значение *num_steps*, которое эту длину бы определяло. Однако для унификации модели этот параметр остаётся незаполненным, что даёт модели возможность подстраиваться под любую длину входных данных, если определены другие два параметра. Безусловно, при учёте того, что ряды подаются по одному.

Так как в данном примере мы рассмотрим модель расширяющегося окна, то размерность тензора маркеров определяется просто количеством примеров, подаваемых на вход. В нашем случае, как уже упоминалось, она всегда будет равна единице.

После этого зададим непосредственно LSTM-алгоритм, который будет эти данные обрабатывать. Для этого введём следующие команды:

```
with tf.variable_scope("LSTM_cell"):
        cell = tf.contrib.rnn.MultiRNNCell(cells=cell_list(num_LSTM_layers),
                                           state_is_tuple=True)
        outputs, states = tf.nn.dynamic_rnn(cell, inputs, dtype=tf.float32)
```

Первая команда создаёт непосредственно многослойную LSTM-модель. В качестве первого аргумента этой функции необходимо подать список клеток, то есть слоёв модели. О том, почему это сделано именно представленным способом, можно прочесть на стр. 38.

```
def cell_list(num_LSTM_layers):
        cell_list = []
        for i in range(num_LSTM_layers):
                cell_list.append(tf.contrib.rnn.BasicLSTMCell(
                hidden_layer_size, forget_bias=1.0))
        return cell_list
```

Функция $cell\_list$ сначала создаёт пустой список, а потом последовательно, исходя из количества слоёв модели, добавляет в него новые клетки. Параметр $hidden\_layer\_size$ как раз и будет определять на этом этапе количество нейронов в каждом скрытом слое.

Далее этот список клеток будет подан на функцию, инициализирующую модель. На выходе получаются два массива, множество прогнозов на каждый элемент ряда, а также вектора состояния, которые в нашем случае не имеют значения и не будут нигде использованы далее.

Следующим шагом необходимо задать тензоры весов и порогов для линейного слоя модели. Так как клетка LSTM на выход подаёт тензор размерности $[batch\_size, sequence\_length, hidden\_layer\_size]$, из которого нам понадобится только последний элемент размерности $[batch\_size, hidden\_layer\_size$ то веса выходного слоя будут, соответственно, $[hidden\_layer\_size, num\_classes]$, а пороги - $[num\_classes]$. Это делается следующими командами:

```
with tf.name_scope("Linear_weights_and_biases"):
        Wl = tf.Variable(tf.truncated_normal([hidden_layer_size, num_classes],
                                             mean = 0,
                                             stddev = 0.01))
        bl = tf.Variable(tf.truncated_normal([num_classes],
                                             mean = 0,
                                             stddev = 0.1))
```

Параметры mean и stddev особой роли не играют и могут быть заданы произвольно. Помимо этого внимательный читатель может заметить,

что входные данные и маркеры имели тип Placeholder, тогда как веса и пороги - Variable. Подробнее о том, в чём здесь разница, можно прочесть в книге [1, Глава 3, стр. 38]. Главное отличие в том, что на эти тензоры не надо подавать данные, они будут инициализироваться самостоятельно и изменяться в процессе обучения модели.

Последними двумя тензорами, которые мы зададим, будут means и stddevs. Подробнее о том, зачем они нужны, будет сказано ниже.

```
with tf.name_scope("Means_and_standard_deviations"):
        mean = tf.placeholder(tf.float32, shape = [batch_size])
        stddev = tf.placeholder(tf.float32, shape = [batch_size])
```

Финальным шагом на данном этапе будет определение взаимодействия элементов графа. В нашем случае необходимы следующие команды:

```
linear_output = get_linear_layer(outputs[0])
final_output = test_decoder(linear_output, mean, stddev)[-1]
mse = tf.reduce_mean(tf.squared_difference(final_output, y))
train_step = tf.train.AdamOptimizer(learning_rate = learning_rate).minimize(mse)
```

Первая строка, что очевидно, преобразует выход LSTM-модели в линейный вид. Для этого используется следующая функция:

```
def get_linear_layer(vector):
        return(tf.matmul(vector, Wl) + bl)
```

На выходе получается вектор предсказаний модели. Так как входные данные должны нормироваться, о чём будет подробно рассказано позже, то выход линейного слоя должен быть отнормирован в обратную сторону, причём с использованием тех значений среднего и стандартного отклонения, с помощью которых нормировались исходные данные. Для этого и были созданы тензоры *mean* и *stddev*. Их наполнение будет определено далее в процессе обучения и использовано в следующей функции второй строки:

```
def test_decoder(array, mean, stddev):
        return (array * stddev) + mean
```

Так как нам необходимо только предсказание по последнему элементу, устанавливается индекс [-1]. После приведения выходных данных в необходимый вид можно посчитать ошибку. Соответственно, в третьей строке кода и вычисляется среднеквадратичная ошибка

В четвёртой строчке мы просто даём оптимизатору, который можно выбрать произвольно, указание минимизировать эту ошибку с заданным шагом оптимизации.

3.5 Инициализация графа

Перед запуском алгоритма обучения необходимо запустить процесс инициализации графа и всех его переменных. Это необходимо, так как переменные в библиотеке Tensorflow активны только в рамках сессии:

```
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
```

Также можно создать пустой список для сохранения истории ошибок на итерациях или же другие опциональные переменные. На процесс обучения они никак не повлияют.

```
mse_list = []
```

3.6 Создание обучающего цикла

Естественно, что при подаче временного ряда алгоритм не будет сходиться за одну итерацию, поэтому необходимо создать цикл. Он имеет следующий вид:

```
for i in range(1000):
        batch_x, batch_y = data_labels(data,
                                       sample_size = sample_size,
                                       batch_size = batch_size)
        data_train, data_test,
        labels_train, labels_test = train_test_split(batch_x,
                                                     batch_y,
                                                     train_size = 0.8)

        normed_data_train, mean_batch, stddev_batch = norm_array(data_train)

        normed_data_train = normed_data_train.reshape(batch_size,
                                                     len(data_train),
```

```
                                                    sequence_length)

        label = labels_train[-1].reshape(batch_size)
        sess.run(train_step, feed_dict = {inputs : normed_data_train,
                                          y : label,
                                          mean : mean_batch,
                                          stddev : stddev_batch})
        mse_batch = sess.run(mse, feed_dict = {inputs : normed_data_train,
                                          y : label,
                                          mean : mean_batch,
                                          stddev : stddev_batch})
```

Данный отрывок кода задаёт первый шаг окна. Входные данные уже были извлечены ранее, поэтому просто переопределяются имена переменных. Этого можно и не делать, если изначально назвать их правильно. О вреде альтернативных способов чтения и подгрузки в цикл данных можно узнать на стр. 38

Далее ряд данных и ряд маркеров будет разбит в определённой пропорции, которая является одним из гиперпараметров модели. Это можно сделать с помощью следующей функции:

```
def train_test_split(X, y, train_size):
        X_train = X[:int(len(X) // (1 / train_size))]
        y_train = y[:int(len(y) // (1 / train_size))]
        X_test = X[int(len(X) // (1 / train_size)):]
        y_test = y[int(len(y) // (1 / train_size)):]
        return X_train, X_test, y_train, y_test
```

Следует обратить внимание на то, что эта функция весьма похожа по названию на функцию *train_test_split* из библиотеки Scikit-learn, однако она не перемешивает в случайном порядке элементы ряда, а просто "обрезает"его в пропорции 0.8 : 0.2. Далее с помощью функции *norm_array* нормируется та часть ряда, которая подаётся на вход:

```
def norm_array(array):
        mean = np.mean(array)
        stddev = np.sqrt(np.sum(np.square(array - mean)) / len(array - 1))
        normalized_array = ((array - mean) / stddev)
        return normalized_array,
                mean.reshape(batch_size),
                stddev.reshape(batch_size)
```

Кроме нормированного ряда на выход функции подаются значения среднего и стандартного отклонения. Впоследствии они будут использованы в тензорах mean и stddev для обработки выходных данных. Эти значения возвращаются сразу в форме, подходящей для подачи в тензоры.

После этого необходимо преобразовать сам нормированный ряд для подачи в сооветствующий тензор. Для наглядности это действие выделено отдельной строкой кода. Как уже упоминалось, эту размерность важно соблюсти, так как в случае ошибки и несоответствия размерности данные не будут поданы в тензор и возникнет ошибка.

Так как изначально при выборе метода мы подразумеваем, что оцениваем только предсказания следующего элемента ряда, поэтому необходимым маркером в нашем случае является последний член ряда маркеров для входных данных. Запишем его под именем label. Наконец, кульминацией этого этапа является запуск тренировочного шага и подача в него всех необходимых данных: входных, маркера, среднего и стандартного отклонения. Все значения, которые необходимо подать, записываются в аргумент $feed\_dict$. Как и в случае с инициализацией переменных, запустить процесс необходимо с помощью функции sess.run(). Полностью аналогично определяется и среднеквадратичная ошибка. Она записывается в отдельную переменную.

В качестве следующего этапа можно рассматривать непосредственно реализацию расширяющегося окна модели. Этот механизм будет представлек в виде цикла, вложенного в предыдущий цикл. Таким образом, каждый пример, поданный на вход, на одной итерации будет производить $len(data\_test) + 1$ шагов оптимизации.

```
for j in range(len(data_test)):
        data_train =  np.append(data_train, data_test[j])
        labels_train = np.append(labels_train, labels_test[j])
        normed_data_train, mean_batch, stddev_batch = norm_array(data_train)
        normed_data_train = normed_data_train.reshape(batch_size,
                                                      len(data_train),
                                                      sequence_length)

        label = labels_train[-1].reshape(batch_size)
        sess.run(train_step, feed_dict = {inputs : normed_data_train,
                                          y : label,
                                          mean : mean_batch,
                                          stddev : stddev_batch})
        mse_batch += sess.run(mse, feed_dict = {inputs : normed_data_train,
```

```
                                        y : label,
                                        mean : mean_batch,
                                        stddev : stddev_batch})
```

Различия вложенного цикла от основного минимальны. Единственное, что изменяется - ширина окна по входным данным и маркерам. Дополнительно суммируется среднеквадратичная ошибка по всем итерациям одного ряда. Это сделано для того, чтобы по завершении вложенного цикла в теле основного подсчитать среднюю ошибку по всему ряду. Это делается следующей командой, расположенной уже в теле основного цикла:

```
mse_batch = mse_batch / len(data_test)
```

На этом шаге можно считать построение модели полностью законченным. Следующий шаг является опциональным. Можно добавить в конце основного цикла следующие три команды:

```
if  i % 100 == 0:
        print("Iter", i, mse_batch, mse_naive(data_train, labels_train))
        mse_list.append(mse_batch)

if mse_batch < 1:
        print("Iter", i, mse_batch, mse_naive(data_train, labels_train))
        break
```

А вне цикла - последнюю команду кода:

```
x = range(1, len(mse_list))
plt.plot(x, mse_list[1:])
plt.show()
```

Первая команда будет сохранять в ранее созданный список значение ошибки на каждой сотой итерации. Вторая - прервёт цикл при достижении критерия останова. Параметры этого алгоритма получены из эмпирических наблюдений и не являются ориентировочными. Последняя команда будет визуализировать тенденции в изменении ошибки. Первое наблюдение не включается так как это ошибка на нулевой итерации. Она всегда чрезмерно высокая и делает график абсолютно неинформативным.

Безусловно, может быть несколько сложно читать отдельные куски кода, поэтому полную версию кода можно найти здесь.

4 Подбор оптимальных параметров

В представленной в предыдущей главе модели некоторые параметры были зафиксированны. Собственно, это были все параметры кроме $hidden\_layer\_size$ и num_LSTM_layers. В данной главе будет представлена попытка подобрать эти параметры.

Для того, чтобы подобрать параметры модели, необходимо несколько видоизменить основной код. Также возникает естественный вопрос касательно используемых данных. Датасет M4 имеет шесть подразделов: почасовые, дневные, недельные, месячные, квартальные и годовые временные ряды. Представленный ниже алгоритм обработает по одному ряду из каждого раздела, а потом результаты будут сопоставлены между собой. Для простоты будут взяты первые ряды каждого раздела. Принцип работы алгоритма будет представлен на на примере ряда почасовых данных.

Структура кода остаётся примерно той же. Сначала подгружаются необходимые библиотеки и функции. Эта часть кода остаётся неизменной. Новшества возникают на этапе загрузки и подготовки входных данных:

```
data_daily = pd.read_csv(r"C:\M4datasets\Monthly-train.csv")
data_day, labels_day = data_labels(data_daily,
                                   sample_size = 1,
                                   batch_size = 1)
data_day = data_day[:50]
labels_day = labels_day[:50]
```

У читателя может возникнуть вполне резонный вопрос, зачем обрезать данные до пятидесяти значений. Дело в том, что почасовые или, например, дневные ряды очень длинные(вплоть до двух тысяч значений), и при попытке произвести вычисления в пропорции $train : test = 0.8 : 0.2$ требуются высокие вычислительные мощности. Можно, конечно, обозначить за тестовые данные просто последние 15-20 значений ряда. При желании читатель может пройти этот альтернативный путь самостоятельно или же, при наличии высоких мощностей, вычислить заявленным методом без обрезания рядов.

Далее зафиксируем длины рядов, на которых производился подбор параметров:

Year: 30 | Quarter: 24 | Month: 50 | Week: 70 | Day: 90 | Hour: 100

Помимо этого необходимо обеспечить различные длины рядов для сравнительного анализа и в данном случае это вполне удобно сделать вручную.

Далее необходимо создать список параметров. Подойдёт следующая команда:

```
num_layers = [1, 2, 3, 4, 5]
num_hidden = [1, 2, 5, 10, 15, 20, 30, 50, 75, 100, 125, 150, 200]
param_list = [(x, y) for x in num_layers for y in num_hidden]
```

Переменная $param\_list$ представляет собой список кортежей из всех возможных пар параметров, в котором первый элемент всегда является количеством слоёв LSTM-модели, а второй - количество нейронов в каждом скрытом слое. Числа взяты произвольно.

Изначально параметры будут подобраны грубо, поэтому промежутки между значениями количества нейронов такие большие.

После этого создаётся пустой список под названием $stop\_list$. Позже будет объяснено, для чего он предназначен.

```
stop_list = []
```

Следующим шагом создадим цикл, который будет последовательно перебирать все пары гиперпараметров из ранее созданного списка:

```
for param in param_list:
        num_LSTM_layers = param[0]
        hidden_layer_size = param[1]
```

На каждой итерации этого цикла будет инициализироваться своя пара параметров. Нетрудно догадаться, что далее в тело цикла необходимо поместить весь код от строки $tf.reset\_default\_graph()$ и до конца вложенного цикла. Тот факт, что инициализация тензоров помещена внутрь цикла связан с тем, что их параметры определяются в соответствии с подбираемыми гиперпараметрами модели, и, следовательно, они должны заново инициализироваться на каждой итерации внешнего цикла. Также перед началом вложенного цикла следует вставить строку:

```
mse_batch_old = 1e-10
```

Её необходимость раскрыта ниже. После окончания внутреннего цикла как и в предыдущий раз необходимо подсчитать среднюю ошибку и ввести критерии останова:

```
if mse_batch_new <= 1:
        stop_list.append(i)
        mse_batch_old = mse_batch_new
        break
if mse_diff(mse_batch_old, mse_batch_new) < 0.001 and mse_batch_new > 1 :
        stop_list.append(2000)
        mse_batch_old = mse_batch_new
        break
mse_batch_old = mse_batch_new
if i == 1999:
        stop_list.append(2000)
        break
```

Первое условие определяет момент, когда алгоритм сошёлся. Также фиксируется значение последней ошибки, так как далее алгоритм выходит из цикла. Это сделано для корректного отображения ошибки на печати при выходе из внутреннего цикла и завершении итерации на внешнем цикле.

Второе условие задаёт ещё один критерий останова. Было замечено, что при большом количестве слоёв и размерности скрытого слоя оптимизация останавливается и ошибка фактически стоит в одной точке, притом на очень больших значениях. Для того, чтобы не производить излишние вычисления, был добавлен критерий, прерывающий цикл, если ошибка изменялась слишком незначительно и была при этом больше критерия сходимости. Функция mse_diff в данном случае просто возвращает абсолютное значение разницы ошибок:

```
def mse_diff(mse_batch_old, mse_batch_new):
        return ((mse_batch_old - mse_batch_new)**2)**(1/2)
```

Третье условие говорит, что если алгоритм не сошёлся к последней итерации(в нашем случае их было 2000), то следует записать просто номер последней итерации.

После выхода из внутреннего цикла в конце внешнего можно вывести напечать, например, следующие показатели для отслеживания процесса перебора:

```
print(param, len(stop_list), mse_batch_old)
```

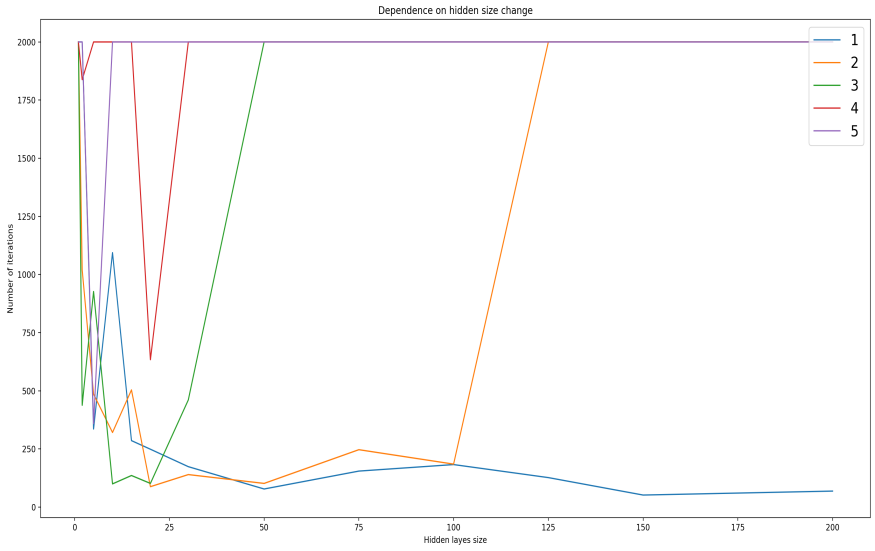
После выполнения данных шагов алгоритм можно считать завершённым. Полную версию можно найти здесь. После его исполнения должен остаться список итераций, на которых алгоритм сошёлся, если он вообще сошёлся. Полученные списки имеют следующий вид:

```
stop_list_year = [2000, 2000, 2000, 727, 554, ... 2000, 2000]
stop_list_quarter = [896, 827, 483, 251, 488, ... 2000, 2000]
stop_list_month = [2000, 2000, 336, 1094, 286, ... 2000, 2000]
stop_list_week = [2000, 2000, 2000, 2000, 1909, ... 2000, 2000]
stop_list_day = [2000, 2000, 455, 251, 1063, ... 2000, 2000]
stop_list_hour = [2000, 2000, 2000, 2000, 2000, ... 2000, 2000]
```

Длина каждого из списков должна равняться количеству комбинаций гиперпараметров, то есть длине списка $param\_list$. В нашем случае она равна 65. Полные версии списков можно посмотреть здесь.

Следует обратить внимание на одно важное замечание. Для ряда дневных данных были использованы несколько другие ограничения на условия сходимости и уменьшение длины шага. Это связано с тем, что вы этом ряду были весьма незначительные различия между соседними членами ряда и поэтому в среднем ошибка была меньше на порядок. Из-за этого сходимость при старых условиях была слишком быстрой и периодически случалась уже на нулевой итерации. Для решения этой проблемы условие сходимости было изменено до $mse\_batch < 0.01$, а условие ограничения шага до $mse\_batch < 5$. Данный шаг позволил получить сопоставимые результаты.

С помощью нескольких несложных функций можно легко визуализировать зависимости, скрытые внутри этих списков. Ниже можно увидеть график сходимости алгоритма, где каждой кривой соответствует определённое количество слоёв модели, а по горизонтали отложены различные размерности скрытого слоя



Из этого графика очевидно, что наилучшим образом при при различной размерности скрытого слоя ведёт себя именно однослойная модель. Схожие тенденции наблюдаются и на других рядах, хотя на дневных и почасовых данных тренды плохо читаемы.

Проверим, действительно ли это так. Для этого просто найдём в каждом стоп-листе минимальное значение и соответствующий ему набор гиперпараметров. Это можно сделать с помощью следующей несложной функции:

```
def optimal_params(param_list, stop_list):
        dict_stop = dict((key, value) for (key, value)
                                        in zip(stop_list, param_list))
        dict_keys = list(dict_stop.keys())
        dict_keys.sort()
        dict_keys = dict_keys[:1]
        for key in dict_keys:
                print(dict_stop[key],  "-" , key)
```

Она формирует словарь из всех значений гиперпараметров и соответствующих итераций, на которых алгоритм остановился. Последние записываются в качестве ключей. Далее словарь сортируется по ключам и на выход подаются гиперпараметры, на которых алгоритм сошёлся быстрее всего. Получим следующие результаты:

```
optimal_params(param_list, stop_list_year) = (1, 30) - 229
optimal_params(param_list, stop_list_quarter) = (1, 200) - 77
optimal_params(param_list, stop_list_month) = (1, 150) - 52
optimal_params(param_list, stop_list_week) = (5, 15) - 390
optimal_params(param_list, stop_list_day) = (1, 7) - 251
optimal_params(param_list, stop_list_hour) = (1, 31) - 575
```

Действительно, в подавляющем большинстве случаев наиболее быстрым алгоритмом оказался однослойный.

После того, как выбор количества слоёв стал очевидным, есть возможность подобрать боле точно размерность скрытого слоя. Для этого несколько модифицируем наш алгоритм. Для начала исправим список пар гиперпараметров.

```
num_layers = [1]
num_hidden = np.arange(1, 200, 2)
param_list = [(x, y) for x in num_layers for y in num_hidden]
```

Так как для дневных и почасовых рядов было сложно понять, выигрывает однослойный алгоритм или двухслойный, добавим для них ещё и двухслойные комбинации и немного увеличим шаг для снижения вычислительной сложности. На качестве это не должно сказаться серьёзно.

```
num_layers = [1, 2]
num_hidden = np.arange(1, 200, 3)
param_list_2 = [(x, y) for x in num_layers for y in num_hidden]
```

Помимо этого, попросим алгоритм фиксировать итерацию остановки. Если она меньше лучшей, которая была до неё, то значение лучшей обновляется, алгоритм прерывает внутренний цикл, печатает значение лучшей и соответствующую пару, и переходит к следующей паре параметров. Если определённая итерация превышает значение лучшей, то внутренний цикл также прерывается, и алгоритм переходит к следующей паре параметров, так как данная нам уже неинтересна. Эти несложные модификации основного алгоритма при надобности читателю предлагается реализовать самостоятельно. Но самые ленивые могут найти модифицированную версию кода здесь. Таким образом, возросшее количество пар параметров компенсируется увеличивающейся скоростью работы алгоритма, который более не производит бессмысленные вычисления. В итоге для теж же рядов получим:

Соответственно, финальные оптимальные параметры выглядят вот так:

Year	Quarter	Month	Week	Day	Hour
(1, 92)	(1, 91)	(1, 130)	(1, 101)	(1, 121)	(1, 143)

Втпомним, что длины рядов были 30, 24, 50, 70, 90 и 100 соответственно. Невооружённым глазом прослеживается некоторая зависимость между размерностью скрытого слоя и длиной ряда. Для большей уверенности вычислим корреляцию мнежду этими показателями:

```
from scipy.stats.stats import pearsonr
a = np.array([30, 24, 50, 70, 90, 100])
b = np.array([21, 33, 49, 55, 61, 57])
print(pearsonr(a, b)[0])
```

Полученное значение равно 0.763, что уже можно считать довольно высоким показателем. Для надёжности повторим эксперимент несколько раз при других параметрах. Для этого повторим эксперимент с финальным подбором, но для каждого ряда повторим подбор по пять раз, для

надёжности. Ряды возьмём те же, а длины сократим для уменьшения итеративной нагрузки:

Year: 30 | Quarter: 24 | Month: 50 | Week: 60 | Day: 60 | Hour: 75

Получим следующие параметры:

Year	Quarter	Month	Week	Day	Hour
(1, 50)	(1, 66)	(1, 56)	(1, 97)	(1, 70)	(1, 98)

Более подробные результаты можно найти здесь.
Вычислим корреляцию:

```
a = np.array([30, 24, 50, 60, 60, 75])
b = np.array([50, 66, 56, 96, 70, 98])
print(pearsonr(a, b)[0])
```

Полученное значение 0.746 почти ничем не отличается от предыдущего значения. Соответственно, некоторая фиксированная погрешность присутствует. Однако даже несмотря на неё приведённый анализ позволяет существенно сузить список параметров для подбора и сократить временные затраты.

5 Построение модели сдвигающегося окна

Теперь построим ещё одну модель окна для последующего сравнения. Для этого придётся внести существенные изменения в код. Они будут описаны кратко, так как большая часть информации уже очевидна из предыдущих глав.

Сначала добавим несколько параметров:

```
learning_rate = 0.0005
win = 50
```

Ширина окна взята произвольно. Не стоит брать слишком маленькое окно, так как это негативно скажется на обучении. Слишком большое окно уменьшит размер датасета. Золотая середина - идеальный вариант.

Также изменится список функций. Функция *data_labels* изменится следующим образом:

```
def data_labels_shift(data, sample_size, batch_size):
        labels = data.iloc[:sample_size, 2:]
        data = data.iloc[:sample_size, 1:]
        indeces = np.random.choice(len(data), size=batch_size, replace=False)

        row = np.array(data.loc[indeces[0]][data.loc[indeces[0]][:] > 0][:-1])

        new_frame = np.empty([len(row) - win + 1, win])
        for i in range(len(row) - win + 1):
                new_frame[i, :] = row[i:(i + win)]

        mean = np.mean(new_frame, axis = 1).reshape(len(new_frame), 1)
        stddev = np.std(new_frame, axis = 1).reshape(len(new_frame), 1)

        batch_x = (new_frame[:, :win-1] - mean) / stddev
        batch_y = new_frame[:, -1]

        return(batch_x, batch_y, mean, stddev)
```

Теперь эта функция генерирует из ряда значений таблицу, которая описывалась ещё во введении. сначала создаётся пустая матрица, которая потом по циклу заполняется значениями. Небольшое упражнение: подумайте, почему в цикле создаётся табличка именно такой размерности. Далее по каждому окну вычисляется среднее и стандартное отклонение, которые сохраняются в отдельные переменные. Затем данные нормируются, и возвращаются все ключевые значения.

Теперь преобразуем данные:

```
data = pd.read_csv(r"C:\M4datasets\Daily-train.csv")

batch_x, batch_y, mean_batch, stddev_batch, =
        data_labels_shift(data, sample_size = sample_size, batch_size = batch_s

data_train, data_test, labels_train, labels_test =
        train_test_split(batch_x, batch_y, train_size = 0.8)

mean_train, mean_test, stddev_train, stddev_test =
        train_test_split(mean_batch, stddev_batch, train_size = 0.8)
```

Доопределим переменные, которые понадобятся нам впоследствии.

```
minibatch_size = 1
test_size = len(data_test)
```

Параметр *minibatch_size* в нашем случае будет равен либо размерности тренировочных данных, либо единице. Позже станет ясно почему. *test_size* понадобится для создания некоторых тензоров.

Новые тензоры:

```
inputs_test = tf.placeholder(tf.float32,
                                shape=[test_size, win-1 , sequence_length])

y_test = tf.placeholder(tf.float32, shape=[test_size])

outputs_test, states_test = tf.nn.dynamic_rnn(cell,
                                inputs_test, dtype=tf.float32)

mean_2 = tf.placeholder(tf.float32, shape = [test_size])
stddev_2 = tf.placeholder(tf.float32, shape = [test_size])
```

Названи последних двух тензоров неэстетичны, но это вынужденное название, так как имена *mean_test* и *stddev_test* уже заняты, и впоследствии это вызовет ошибку, так как алгоритм не поймёт, использовать ему тензор или переменную. Все эти тензоры будут нужны для построения прогноза на тестовых данных. Также для этой цели нужно вставить дополнительные функции нейросети.

```
outputs_test = tf.reshape(outputs_test, [-1, hidden_layer_size])
linear_output_test = tf.reshape(get_linear_layer(outputs_test),
                                [test_size, win-1])

final_output_test = test_decoder(linear_output_test[:, -1], mean_2, stddev_2)
mse_test = tf.reduce_mean(tf.squared_difference(final_output_test, y_test))
```

Изменятся и основные функции. Так как размерность выход изменилась, для поэлементного матричного произведения необходимо производить трансформации размерности тензоров:

```
outputs = tf.reshape(outputs, [-1, hidden_layer_size])
linear_output = tf.reshape(get_linear_layer(outputs), [minibatch_size, win-1])

final_output = test_decoder(linear_output[:, -1], mean, stddev)
mse = tf.reduce_mean(tf.squared_difference(final_output, y))
train_step = tf.train.AdamOptimizer(
                               learning_rate = learning_rate).minimize(mse)
```

На этом установка параметров окончена. Теперь необходимо написать обучающий цикл. Он будет иметь более простое и эргономичное построение.

```
for i in range(20000):
        indeces = np.random.choice(len(data_train),
                                        size=minibatch_size, replace=False)
        minibatch_x = data_train[indeces].reshape(
                                        minibatch_size, win-1, sequence_length)
        minibatch_y = labels_train[indeces].reshape(minibatch_size)
        mean_train_shuffle = mean_train[indeces].reshape(minibatch_size)
        stddev_train_shuffle = stddev_train[indeces].reshape(minibatch_size)


        data_test = data_test.reshape(test_size, win-1 , sequence_length)
        labels_test = labels_test.reshape(test_size)

        mean_test = mean_test.reshape(test_size)
        stddev_test = stddev_test.reshape(test_size)

        sess.run(train_step, feed_dict = {inputs : minibatch_x,
                                          y : minibatch_y,
```

```
                                                mean : mean_train_shuffle,
                                                stddev : stddev_train_shuffle})

        mse_batch = sess.run(mse, feed_dict = {inputs : minibatch_x,
                                                y : minibatch_y,
                                                mean : mean_train_shuffle,
                                                stddev : stddev_train_shuffle})

        mse_prob = sess.run(mse_test, feed_dict = {inputs_test : data_test,
                                                y_test : labels_test,
                                                mean_2 : mean_test,
                                                stddev_2 : stddev_test})
```

В общем, отличия от предыдущей модели не радикальные. На каждой итерации из тренировочных данных выбирается количество примеров в размере $minibatch\_size$. Также необходимо выбрать соответствующие им значения среднего и стандартного отклонения. При этом все значения приводятся к форме соответствующих им тензоров. Далее они подаются на соответствующие функции, которые делают шаг обучения, подсчитывают ошибку на тренировочных данных и на тестовых. Полную версию кода можно найти здесь

6 Сравнение различных методов

Теперь сравним различные методы окон. Для этого измерим минимальную ошибку, которую смог достичь каждый вариант алгоритма, а также время его исполнения с обучением и без.

Для измерения времени можно использовать следующую нехитрую конструкцию:

```
from datetime import datetime
startTime= datetime.now()

#некоторый код

timeElapsed=datetime.now()-startTime
print("Time elpased (hh:mm:ss.ms) {}".format(timeElapsed))
```

Измерив несколько алгоритмов, получим следующую таблицу:

Тип модели	Сдвигающееся окно		Расширяющееся окно	
	Time	MSE	Time	MSE
Один пример	0.0348 (0.3291)	60.4925 (0.5553)	0.0857 (0.3695)	$\geq 10^6$ (≤ 0.01)
50 примеров	0.1132 (0.4313)	43.3558 (0.2564)	— —	— —
Весь сет	0.1226 (0.3590)	69.6659 (16.2584)	— —	— —

Время совершения итерации указано как для алгоритма без совершения шага обучения, так и с ним (значение в скобках). Наименьшая достигнутая среднеквадратичная ошибка также дана в двух вариациях: на тестовых данных и на тренировочных(в скобках)

В глаза бросаются несколько моментов. Во-первых, алгоритм расширяющегося окна явно переобучается. На тренировочных данных ошибка близка к нулю, а на тестовых крайне велика. Очевидность этого можно увидеть из четвёртого графика в Приложении. Скорее всего это связано с недостаточной длиной ряда, которую может принять алгоритм из-за своей внутренней трудоёмкости. Это означает, что он в данных условиях неэффективен.

Во-вторых, метод сдвигающегося окна при полном градиентном спуске был довольно эффективен, но метрика качества не опустилась ниже наивного прогноза. Следует напомнить, что MSE наивного прогноза равнялась 65.62. Кроме того, время на прогнозе было самым большим. Время на прогнозе с обучением относительно сопоставмо с другими вариантами. График находится на третьем месте в Приложениии.

В-третьих, первым алгоритмом, который победил наивный прогноз, был вариант с использованием только одного примера, то есть стохастический спуск. Скорость этого алгоритма была самой высокой. Однако по сравнению с наивным прогнозом он выигрывал несильно. Любопытно взглянуть на график предсказания. Он на первом месте в Приложении. Легко заметить, что прогноз почти полностью повторяет рисунок целевого ряда, однако со сдвигом. Очевидно, что алгоритм почти полностью сымитировал наивный прогноз, хотя и с некоторой погрешностью.

В-четвёртых, наиболее эффективным стал алгоритм, который использовал 50 примеров за итерацию. Минимальная достигнутая ошибка на нем существенно ниже, чем у наивного прогноза. Из второго графика Приложения очевидно, что теперь алгоритм уже не похож на наивный прогноз с его сдвигами. Теперь он подстраивается уже под истинную последовательность. Вопрос о том, как подобрать оптимальное количество примеров остаётся открытым, хотя вполне можно попробовать подобрать

перебором, исходя из минимально достижимой ошибки.

7 Построение модели на R, библиотека Keras

Первым этапом необходимо установить ключевые библиотеки. Если вы работаете в Rstudio, то это можнор сделать через Tools → Install packages, или же просто написать в консоли:

```
install.packages("keras")
install.packages("rlist")
```

Дополнительно будут установлены все библиотеки, необходимые для работы устанавливаемых. Например, для Keras будет установлена библиотека Tensorflow. Далее необходимо эти библиотеки подключить:

```
library(keras)
library(rlist)
```

Следующим шагом импотрируем данные и удалим из них нижний столбец. Как мы помним, он содержит данные о происхождении ряда и в принципе нам не нужен.

```
data <- read.csv(file = "C:/M4datasets/Hourly-train.csv",
                         header = TRUE, sep = ",")
data <- data[, -1]
```

Введём необходимые переменные:

```
num_classes <- 1
batch_size <- 1
sample_size <- 1
sequence_length <-1
num_LSTM_layers <- 1
hidden_layer_size <- 700
win <- 5
epochs <- 50
train_size <- 0.7
```

Теперь необходимо обернуть входные данные в необходимую форму.

```
unscaled_row <- get_row(data)
scaled_row <- row_scale(unscaled_row)
mean <- mean(unscaled_row)
stddev<- sd(unscaled_row)
scaled_row <- gen_row(scaled_row)
df <- gen_data(win, scaled_row)
```

Функция get_row извлекает первый ряд, так как $sample\_size = 1$, и отсекает неизвестные значения типа NA. Если бы рядов было больше одного, то в операции обрезания ряда потребовался цикл, но в данном случае в этом нет необходимости.

```
get_row <- function(data) {
        row <- data[c(1:sample_size),]
        row <- row[!is.na(row)]
        return(row)
}
```

Далее указанный ряд нормируется с помощью функции row_scale, а значения среднего и отандартного отклонения сохраняются в отдельные переменные.

```
row_scale <- function(row) {
        return((row - mean(row))/sd(row))
}
```

Последним этапом является преобразование полученного ряда в датафрейм непосредственно для реализации метода сдвигающегося окна. Для этого используется следующая функция:

```
gen_data <- function(win, row) {
        df <- data.frame(matrix(, ncol = win), stringsAsFactors=FALSE)
        colnames(df)<-  c(paste0("X", 1:(win -1)) , "y")
        for (i in seq(1:as.integer(length(row)- win + 1))) {
                df[i,] <- row[(i:(i + win - 1))]
        }
        return(df)
}
```

Сначала она создаёт пустой датафрейм с количеством колонок равным ширине окна. Далее колонкам присваиваются имена, последней - y, а всем остальным X + порядковый номер. После этого строкам последовательно присваиваются значения через цикл.

Следующим этапом разделим нашу выборку на train и test:

```
X_train <- data.frame(train_test_split(win, df)[1])
X_test <- data.frame(train_test_split(win, df)[2])
y_train <- data.frame(train_test_split(win, df)[3])
y_test <- data.frame(train_test_split(win, df)[4])
```

Функция *train_test_split* в данном случае, как и для варианта с Tensorflow, просто делит выборку, не перемешивая её. Следует также помнить, что при обращении к датафрейму функция *length*() возвращает количество столбцов, а не строк, как в Python. Дополнительная обёртка в датафрейм понадобится далее, так как иначе код выдаст ошибку.

```
train_test_split <- function(win, df1) {
        X_train <- df1[1: (nrow(df1) * train_size),-length(df1)]
        y_train <- df1[1: (nrow(df1) * train_size),length(df1), drop = F]
        X_test <- df1[(nrow(df1) * 0.7):nrow(df1),-length(df1)]
        y_test <- df1[(nrow(df1) * 0.7):nrow(df1),length(df1), drop = F]
        return(list(X_train, X_test, y_train, y_test))
}
```

Следующим шагом необходимо преобразовать полученные двумерные таблицы в трёхмерные тензоры для их обработки алгоритмом. Столбцы маркеров можно оставить двумерными.

```
X_train <- array(as.matrix(X_train), dim = c(nrow(X_train), ncol(X_train), 1))
y_train <- array(as.matrix(y_train), dim = c(nrow(y_train), ncol(y_train)))
X_test <- array(as.matrix(X_test), dim = c(nrow(X_test), ncol(X_test), 1))
y_test <- array(as.matrix(y_test), dim = c(nrow(y_test), ncol(y_test)))
```

Очень важно для начала обратить датафрейм в матрицу, а только после в трёхмерный тензор, так как иначе на выходе получится бессмысленный набор чисел. Аргумент *dim* указывает финальные размерности тензора.

Далее необходимо создать непосредственно модель. В Keras она выглядит несколько проще, чем в Tensorflow.

```
model <- keras_model_sequential(layers = layer_list(num_LSTM_layers,
                                                    hidden_layer_size))

model %>%
        compile(loss = "mse", optimizer = "adam")
```

Первая строка создаёт непосредственно модель, прописывая столько слоёв, сколько указывает функция $layer\_list$. Она создаёт список слоёв. Из-за крайней массивности и большого количества отступов её проще продемонстрировать в полной версии кода.

Очень важно внутри этой функции соблюсти размерность входного слоя, а также установить параметры всех LSTM-слоёв $return\_sequences = FALSE$, а в последнем $return\_sequences = TRUE$. Помимо этого в конце необходимо добавить линейный выходной dense-слой и соблюсти размерности.

Также модель необходимо скомпилировать, указав метрику качества и оптимизатор.

Следующим этапом обучим нашу модель. Здесь всё интуитивно понятно:

```
for (i in 1:epochs) {
        model %>% fit(  x          = X_train,
                        y          = y_train,
                        batch_size = batch_size,
                        epochs     = 20,
                        verbose    = 2,
                        validation_split = 0.3,
                        shuffle    = FALSE)
        model %>% reset_states()
}
```

Указываются данные для обучения, количество итераций, доля данных для кросс-валидации и параметр, отвечающий за перемешивание выборок. Параметр *verbose* отвечает за то, в какой форме будет выводиться информация на печати. При запуске этого элемента в консоли будет отображаться примерно следующее:

```
Train on 340 samples, validate on 147 samples
Epoch 1/20
- 4s - loss: 0.3421 - val_loss: 0.0932
Epoch 2/20
- 2s - loss: 0.1244 - val_loss: 0.0234
...
Epoch 19/20
- 1s - loss: 0.0092 - val_loss: 0.0084
Epoch 20/20
- 1s - loss: 0.0091 - val_loss: 0.0086
```

Как можно видеть, ошибка при обучении снижается. Теперь на обученной модели построим предсказания.

```
train_prediction <- model %>%
        predict(X_train, batch_size = batch_size)

test_prediction <- model %>%
        predict(X_test, batch_size = batch_size)
```

Также отнормируем в обратную сторону все переменные для адекватного подсчёта ошибки. Для этого используем ранее сохранённые значения среднего и стандартного отклонения, а также простую функцию $row\_decoder$.

```
row_decoder <- function(row, mean, stddev) {
        return((row*stddev) + mean)
}

train_prediction <- row_decoder(train_prediction, mean, stddev)
test_prediction <- row_decoder(test_prediction, mean, stddev)
y_train <- row_decoder(y_train, mean, stddev)
y_test <- row_decoder(y_test, mean, stddev)
```

Наконец, в качестве финального шага протестируем нашу модель, вычислив MSE на тренировочных и на тестовых данных.

```
mse <- function(row1, row2){
        return(sum((row1 - row2)^2) / length(row1))
}

print(mse(train_prediction, y_train))
print(mse(test_prediction, y_test))
```

На выходе получим:

```
[1] 234.5212
[1] 252.7503
```

Как можно видеть, ошибка на тестовых данных хоть и выше, чем на тренировочных, но не на много. При правиильном подборе параметров наверняка удастся уменьшить ошибку на тестовых данных, но повторять этот процесс в данной работе ещё раз уже не имеет особого смысла.

На данном этапе можно считать построение модели завершённым. Следует отметить, что при попытке запустить код в Rstudio среда выдаёт критическую ошибку на этапе вызова функции $layer\_list$. Логичного решения этой проблемы найдено, к сожалению, не было. Однако, так как наиболее оптимальным алгоритмом, как показывает практика, будет однослойная модель, можно задать слои и вручную.

8 Грабли

8.1 Сложности с установкой библиотек

Даннная работа подразумевает установку библиотеки Tensorflow в среду Anaconda как наиболее подходящую и удобную для написания кода и его редактирования. Эта библиотека не входит в список предустановленных для Anaconda, поэтому её необходимо установить отдельно. Подробную инструкцию по установке можно найти здесь, на официальном сайте Tensorflow. Для установки необходимо создать отдельный подраздел. Для этого нужно открыть консоль Anaconda и ввести команду conda create -n tensorflow pip python=3.6. Соответственно, на место числа 3.6 следует вписать вашу версию python. Далее нужно активировать подраздел командой activate tensorflow. После этого есть два варианта установки, то есть две возможные команды:

1. Установка через pip: 'pip install tensorflow'
2. Установка через conda: 'conda install tensorflow'

Официальный сайт Tensorflow рекомендует второй метод на момент написания работы, однако практика показывает, что это ошибка. Если установить через conda, то автоматически установится довольно старая версия Tensorflow, в районе 1.2, в то время как на момент написания работы вышла версия 1.7. При этом при попытке обновить пакет новая версия оказывалась не выше 1.5. При установке и обновлении через pip подобных недостатков не обнаруживалось.

8.2 Обнуление графа

В некоторых примерах кода, которые были обнаружены в сети, строки, обнуляющей граф, не было обнаружено. Например, в книге [1, Глава 5, стр. 89-93]. Возможно, это связано с тем, что выверенные примеры из учебников идеальны настолько, что не подразумевают перезапуска. Однако на практике в процессе поиска ошибок это жизненно необходимо. Без соответствующей команды интерпретатор будет выдавать ошибку. Можно, конечно, каждый раз перезапускать ядро Anaconda, но это весьма неудобно. Поэтому в начале построения графа следует вписать эту строку:

```
tf.reset_default_graph()
```

8.3 Многослойная LSTM

В некоторых учебниках [1, Глава 5, стр 93], а также на большинстве интернет-ресурсов(например, здесь и здесь) указано, что для того, чтобы создать многослойную LSTM, необходимо выполнить такую команду:

```
lstm_cell = tf.contrib.rnn.BasicLSTMCell(hidden_layer_size,
                                         forget_bias=1.0)
cell = tf.contrib.rnn.MultiRNNCell(cells=[lstm_cell]*num_LSTM_layers,
                                   state_is_tuple=True)
```

Однако при попытке её выполнить возникает ошибка. До конца неясно, с чем это может быть связано. Возможно, один из багов последней версии, который запрещает функции ссылаться на одну и ту же клетку несколько раз. При этом при параметре num_LSTM_layers равном единице код работает в штатном режиме.

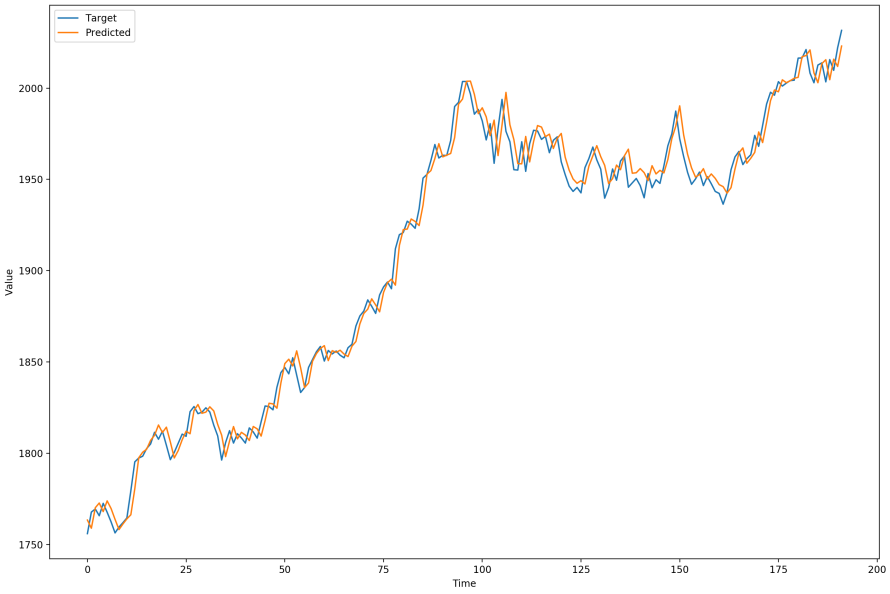
8.4 Чтение данных в цикле

Изначально может показаться хорошей идеей использовать функцию $data\_labels$ напрямую в самом начале цикла, чтобы не делать этого в начале кода. Создаётся впечатление, что это повысит читабельность кода и оптимизирует его. Однако при попытке запуска кода с рядом большой размерности и "тяжёлым"датасетом, постоянное обращение к данным будет сводить вычислительную мощность буквально к нулю. Проверить это несложно. Достаточно просто добавить функцию в цикл и попросить алгоритм обработать первый ряд в сете "Hourly"датасета M4. Таким образом гораздо проще обрезать данные до необходимого небольшого размера сразу после чтения и в дальнейшем использовать только их.

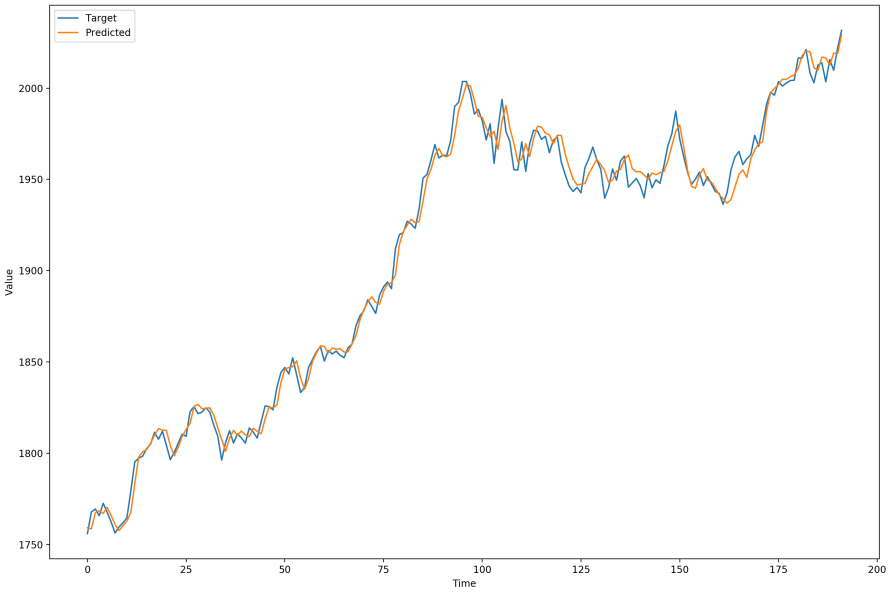
9 Заключение

Несмотря на продемонстрированную неэффективность метода расширяющегося окна, представленный принцип перебора гиперпараметров унифицирован и может быть использован на моделях с другим устройством. Более того, благодаря своей простоте он легко может быть переведён на другой язык, например, на R. Метод сдвигающегося окна зарекомендовал себя высокой эффективностью даже без должного подбора параметра $minibatch\_size$.

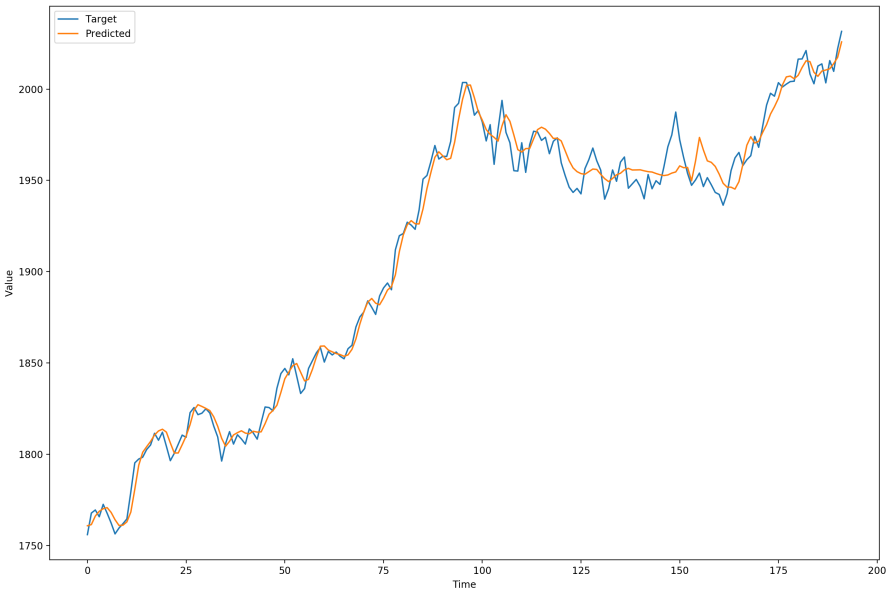
10 Приложение



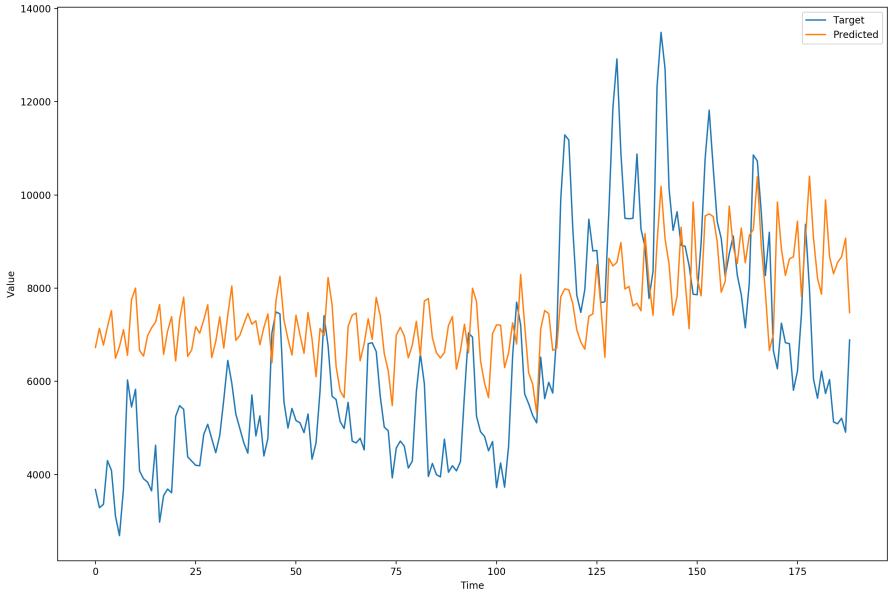
1 пример



50 примеров



Весь сет



Сдвигающееся окно

Список литературы

[1] Learning TensorFlow: A Guide to Building Deep Learning Systems by Tom Hope English | 2017 | ISBN: 1491978511 | 242 Pages |

[2] Tensorflow For Machine Intelligence | Sam Abrahams, Danijar Hafner, Erik Erwitt, Ariel Scarpinelli | Bleeding Edge Press | 2016 | ISBN : 978-1-939902-35-1 | 305 pages

[3] Jason Brownlee | Deep Learning With Python Develop Deep Learning Models On Theano And TensorFlow Using Keras | 2016