Stock Price Prediction Using Deep Learning

Submitted By: Khawaja Ammad ul Islam (P15-6037) and Saif Ur Rehman (P15-6005)

Submitted To: Dr Muhammad Nauman

National University of Computer and Emerging Sciences

Contents

Abstract

The data consisted of index as well as stock prices of the S&P's 500 constituents. So we developed deep learning model for predicting the S&P 500 index based on the 500 constituents prices one minute ago.

Data Processing

Importing and Preparing the Data:

Our team exported the scraped stock data from our scraping server as a csv file. The dataset contains n = 41266 minutes of data ranging from April to August 2017 on 500 stocks as well as the total S&P 500 index price. Index and stocks are arranged in wide format.

```
# Import data
data = pd.read_csv('data_stocks.csv')

# Drop date variable
data = data.drop(['DATE'], 1)

# Dimensions of dataset
n = data.shape[0]
p = data.shape[1]

# Make data a numpy array
data = data.values
```

The data was already cleaned and prepared, meaning missing stock and index prices were LOCF'ed (last observation carried forward), so that the file did not contain any missing values.

Preparing Training and Test Data:

The dataset was split into training and test data. The training data contained 80% of the total dataset. The data was not shuffled but sequentially sliced. The training data ranges from April to approx. end of July 2017, the test data ends end of August 2017.

```
# Training and test data
train_start = 0
train_end = int(np.floor(0.8*n))
test_start = train_end
test_end = n
data_train = data[np.arange(train_start, train_end), :]
data_test = data[np.arange(test_start, test_end), :]
```

There are a lot of different approaches to time series cross validation, such as rolling forecasts with and without refitting or more elaborate concepts such as time series bootstrap resampling. The latter involves repeated samples from the remainder of the seasonal decomposition of the time series in order to simulate samples that follow the same seasonal pattern as the original time series but are not exact copies of its values.

Data scaling:

Most neural network architectures benefit from scaling the inputs (sometimes also the output). Why? Because most common activation functions of the network's neurons such as tanh or sigmoid are defined on the [-1, 1] or [0, 1] interval respectively. Nowadays, rectified linear unit (ReLU) activations are commonly used activations which are unbounded on the axis of possible activation values. However, we will scale both the inputs and targets anyway. Scaling can be easily accomplished in Python using sklearn's MinMaxScaler.

```
# Scale data
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(data_train)
data_train = scaler.transform(data_train)
data_test = scaler.transform(data_test)

# Build X and y
X_train = data_train[:, 1:]
y_train = data_train[:, 0]
X_test = data_test[:, 1:]
y_test = data_test[:, 0]
```

Designing the Network Architecture

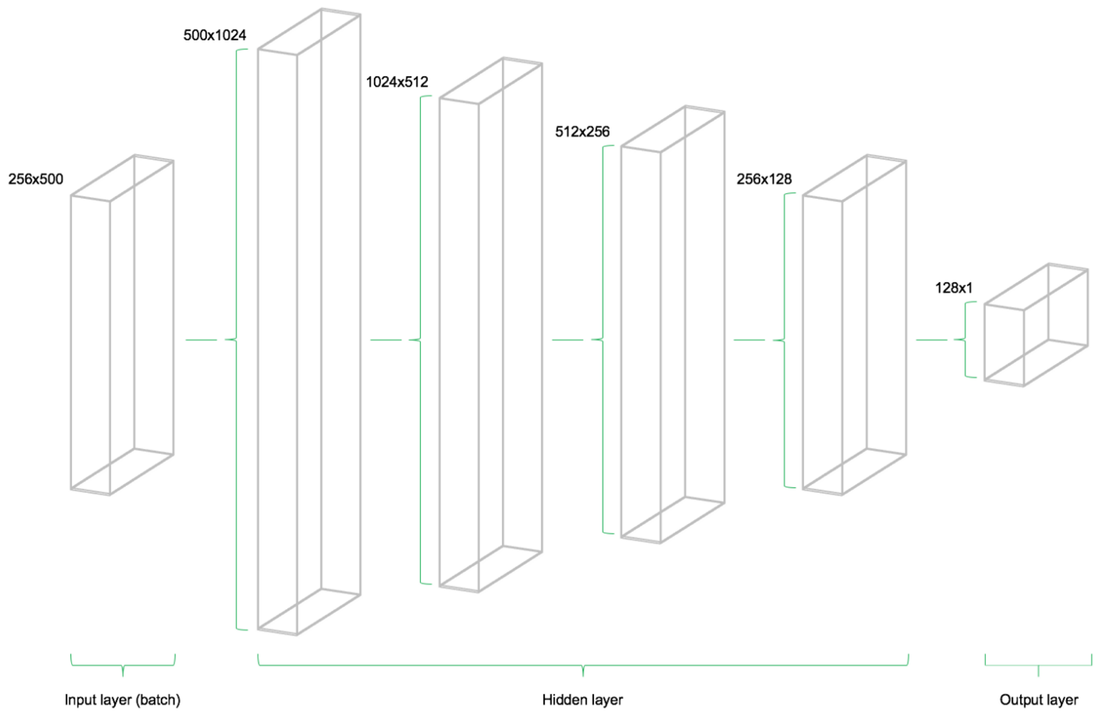
After definition of the required weight and bias variables, the network topology, the architecture of the network, needs to be specified. Hereby, placeholders (data) and variables (weighs and biases) need to be combined into a system of sequential matrix multiplications.

Furthermore, the hidden layers of the network are transformed by activation functions. Activation functions are important elements of the network architecture since they introduce non-linearity to the system. There are dozens of possible activation functions out there, one of the most common is the rectified linear unit (ReLU) which will also be used in this model.

```
# Hidden layer
hidden_1 = tf.nn.relu(tf.add(tf.matmul(X, W_hidden_1), bias_hidden_1))
hidden_2 = tf.nn.relu(tf.add(tf.matmul(hidden_1, W_hidden_2), bias_hidden_2))
hidden_3 = tf.nn.relu(tf.add(tf.matmul(hidden_2, W_hidden_3), bias_hidden_3))
hidden_4 = tf.nn.relu(tf.add(tf.matmul(hidden_3, W_hidden_4), bias_hidden_4))

# Output layer (must be transposed)
out = tf.transpose(tf.add(tf.matmul(hidden_4, W_out), bias_out))
```

The image below illustrates the network architecture. The model consists of three major building blocks. The input layer, the hidden layers and the output layer. This architecture is called a feedforward network. Feedforward indicates that the batch of data solely flows from left to right. Other network architectures, such as recurrent neural networks, also allow data flowing "backwards" in the network.



Cost Function

The cost function of the network is used to generate a measure of deviation between the network's predictions and the actual observed training targets. For regression problems, the mean squared error (MSE) function is commonly used. MSE computes the average squared deviation between predictions and targets.

Basically, any differentiable function can be implemented in order to compute a deviation measure between predictions and targets.

```
# Cost function
mse = tf.reduce_mean(tf.squared_difference(out, Y))
```

However, the MSE exhibits certain properties that are advantageous for the general optimization problem to be solved.

Optimizer

The optimizer takes care of the necessary computations that are used to adapt the network's weight and bias variables during training. Those computations invoke the calculation of so called gradients that indicate the direction in which the weights and biases have to be changed during training in order to minimize the network's cost function. The development of stable and speedy optimizers is a major field in neural network a deep learning research.

```
# Optimizer
opt = tf.train.AdamOptimizer().minimize(mse)
```

Here the Adam Optimizer is used, which is one of the current default optimizers in deep learning development. Adam stands for "**Ada**ptive **M**oment Estimation" and can be considered as a combination between two other popular optimizers AdaGrad and RMSProp.

Fitting the Neural Network and Results

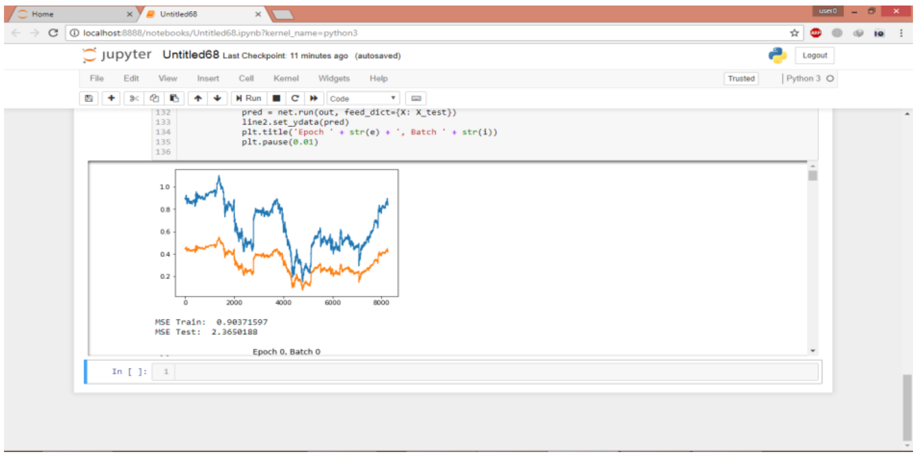
After having defined the placeholders, variables, initializers, cost functions and optimizers of the network, the model needs to be trained. Usually, this is done by minibatch training. During minibatch training random data samples of n = batch_size are drawn from the training data and fed into the network. The training dataset gets divided into n / batch_size batches that are sequentially fed into the network. At this point the placeholders X and Y come into play. They store the input and target data and present them to the network as inputs and targets.

A sampled data batch of X flows through the network until it reaches the output layer. There, TensorFlow compares the models predictions against the actual observed targets Y in the current batch. Afterwards, TensorFlow conducts an optimization step and updates the networks parameters, corresponding to the selected learning scheme. After having updated the weights and biases, the next batch is sampled and the process repeats itself. The procedure continues until all batches have been presented to the network. One full sweep over all batches is called an epoch.

The training of the network stops once the maximum number of epochs is reached or another stopping criterion defined by the user applies.

During the training, we evaluate the networks predictions on the test set—the data which is not learned, but set aside—for every 5th batch and visualize it.

A sample screenshot for the results:


Home
Untitled68
localhost:8888/notebooks/Untitled68.ipynb?kernel_name=python3
jupyter Untitled68 Last Checkpoint: 11 minutes ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code
1.0 0.8 0.6 0.4 0.2
0 2000 4000 6000 8000
Epoch 0, Batch 0
In [ ]:

```
            pred = net.run(out, feed_dict={X: X_test})
            line2.set_ydata(pred)
            plt.title('Epoch ' + str(e) + ', Batch ' + str(i))
            plt.pause(0.01)
```

```
MSE Train:  0.90371597
MSE Test:  2.3650188
```

References

- https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/
- https://www.tensorflow.org/api_docs/python/
- https://www.researchgate.net/publication/269935079_Adam_A_Method_for_Stochastic_Optimization

GitHub link to the Project: https://github.com/amd23/machineLearningsSemesterProject