

CSC 158

PROJECT 2 – Training a Neural Network to Predict Stock Prices



Youssef Elmougy

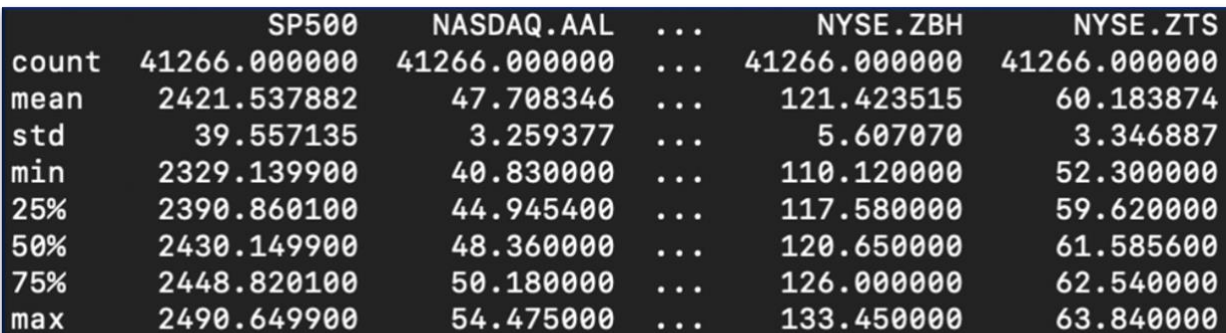
PROBLEM DESCRIPTION

There exists several API's, such as the Google Finance API, providing real-time stock charts and stock prices. The idea of accurately extracting the minutely stock data from these API's and developing a deep learning model aimed at training and testing the data and hence withhold the ability to predict real-time stock prices, of course to a certain percentage of accuracy, is inevitable.

This project exploits a huge dataset containing stock prices for the S&P 500 Index and its constituents from April 2017 to August 2017 to precisely predict the next minute stock price of the S&P 500 Index. As the neural network is training and testing, a graph of the actual stock price and the predicted stock price will be plotted, moreover the error and accuracy will be calculated and displayed.

DATA DESCRIPTION

The dataset is stored in the file '`data_stocks.csv`'. This dataset contains 41,266 minutes of data ranging from April 2017 to August 2017 on the prices of the 500 stock constituents along with the total S&P 500 index price. The following shows the description of the data:



	SP500	NASDAQ.AAL	...	NYSE.ZBH	NYSE.ZTS
count	41266.000000	41266.000000	...	41266.000000	41266.000000
mean	2421.537882	47.708346	...	121.423515	60.183874
std	39.557135	3.259377	...	5.607070	3.346887
min	2329.139900	40.830000	...	110.120000	52.300000
25%	2390.860100	44.945400	...	117.580000	59.620000
50%	2430.149900	48.360000	...	120.650000	61.585600
75%	2448.820100	50.180000	...	126.000000	62.540000
max	2490.649900	54.475000	...	133.450000	63.840000

Fig. 1: `print(dataset.describe())`

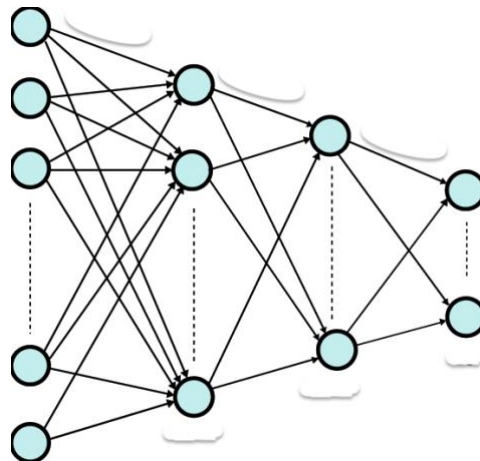
The date/time column in the dataset has been dropped as it serves no firm purpose because the implementation instead is greatly fixated on the numerical values of the stocks.

This project focuses on predicting the next minute stock price for the S&P 500, so each row of the dataset contains the constituent 500 stock's prices at $T = t$ and the stock price of S&P 500 at $T = t + 1$.

METHOD DESCRIPTION

This deep learning model is built with TensorFlow. This provides much more flexibility and the ability to use a wider range of building blocks and concepts. The dataset is split in the following manner: 80% as the training data, and 20% as the testing data. Both the training and testing data is scaled using sklearn's ***MinMaxScaler()*** and bounded within the feature range [-1,1].

The TensorFlow model consists of four layers. The number of neurons in each layer is experimentally changed to test for the configuration with the best output. Although it follows a common sequence, the subsequent layer's number of neurons is always half the number of neurons of the previous layer. This idea of decreasing number of neurons is because each subsequent layer compresses the information that is provided to it and therefore produces a more accurate result.



The model's abstract representation of neural network is through placeholders and variables. They are explained below:

- *placeholders* – two placeholders are utilized, ***ins***: which comprises the NN's inputs which is the stock prices of the 500 constituents, and ***outs***: which comprises the NN's outputs which is the stock price of the S&P 500.



- *variables* – each of the four layers as well as the output layer has its unique weight and bias variable. The weight variables are implemented in a way that allows for each layer

to pass its outputs as the input of the next layer. The bias variables are implemented as the number of neurons in the layer.

The placeholders and variables are then combined to design the architecture of the neural network. An activation function is then implemented, this activation function is experimentally changed to test for the finest accuracy.

Finally, the neural network is fitted and trained. The network is trained using batches which change in number for each epoch run on the data. The error is calculated using the Mean Squared Error (MSE) approach, and an optimizer is used to minimize the MSE. The predictions of the neural network are plotted on a graph along with the actual data line.

IMPLEMENTATION

The following shows the code of the algorithm and a description for its implementation:

Importing the data

```
#IMPORT THE DATA FILE, REMOVE THE 'DATE' COLUMN FROM DATASET
dataset = pd.read_csv('data_stocks.csv')
dataset = dataset.drop(['DATE'], 1)
```

The dataset is read from the file and the date/time column is dropped.

Splitting the dataset into training data and testing data

```
#SPLIT DATASET INTO 80% FOR TRAINING DATA AND 20% FOR TESTING DATA
#TRAINING DATA, 80%
traindata = dataset[np.arange(0, int(np.floor(0.8*num_data))), :]
#TESTING DATA, 20%
testdata = dataset[np.arange(int(np.floor(0.8*num_data))+1, num_data), :]
```

As mentioned earlier, the dataset is split as 80% for the training data and stored in the variable ***traindata*** and the rest is set for the testing data stored in the variable ***testdata***.

Scaling the dataset

```
#SCALE DATASET USING MinMaxScaler WITH VALUES BEING IN THE RANGE OF (-1,1)
scaler = MinMaxScaler(feature_range=(-1, 1))
scaler.fit(traindata)

#SCALE BOTH THE TRAINING AND THE TESTING DATASET
traindata = scaler.transform(traindata)
testdata = scaler.transform(testdata)
```

The **MinMaxScaler()** is implemented with the **feature_range=(-1,1)** which bounds the data within those values. The data is then fitted, and the training and testing variables are transformed to the scaled values.

Defining the placeholders

```
# placeholders
ins = tf.placeholder(dtype=tf.float32, shape=[None, num_stocks])
outs = tf.placeholder(dtype=tf.float32, shape=[None])
```

As mentioned before, the placeholder's **ins** and **outs** store the data that is inputted and outputted in the network. The input is in the form of a 2-dimensional matrix, and the output is a 1-dimensional vector. 'None' is used to give freedom to change the variable later on in the program.

Defining the variables

```
# layer_i_neurons, -----TRY OUT DIFFERENT NUMBER OF NEURONS-----
layer1_neurons = 1000 # double input size
layer2_neurons = 500  # 50% of previous layer
layer3_neurons = 250  # 50% of previous layer
layer4_neurons = 125  # 50% of previous layer
```

```
# layer_i_weight, layer_i_bias
layer1_weight = tf.Variable(weight_initializer([num_stocks, layer1_neurons]))
layer1_bias = tf.Variable(bias_initializer([layer1_neurons]))
layer2_weight = tf.Variable(weight_initializer([layer1_neurons, layer2_neurons]))
layer2_bias = tf.Variable(bias_initializer([layer2_neurons]))
layer3_weight = tf.Variable(weight_initializer([layer2_neurons, layer3_neurons]))
layer3_bias = tf.Variable(bias_initializer([layer3_neurons]))
layer4_weight = tf.Variable(weight_initializer([layer3_neurons, layer4_neurons]))
layer4_bias = tf.Variable(bias_initializer([layer4_neurons]))
output_weight = tf.Variable(weight_initializer([layer4_neurons, 1]))
output_bias = tf.Variable(bias_initializer([1]))
```

The number of neurons in each layer is defined as variables to provide easy access to changes to find the best architecture combinations. Furthermore, the weight and bias for each layer is defined. The weight for each layer is defined as a two-dimensional matrix and the bias for each layer is defined as a one-dimensional vector.

Defining the activation function

```
layer1 = tf.nn.relu(tf.add(tf.matmul(ins, layer1_weight), layer1_bias))
layer2 = tf.nn.relu(tf.add(tf.matmul(layer1, layer2_weight), layer2_bias))
layer3 = tf.nn.relu(tf.add(tf.matmul(layer2, layer3_weight), layer3_bias))
layer4 = tf.nn.relu(tf.add(tf.matmul(layer3, layer4_weight), layer4_bias))
layer_output = tf.transpose(tf.add(tf.matmul(layer4, output_weight), output_bias))
```

This is where the activation function is defined. In this case, the layers of the network are transformed by the **ReLU** function (Rectified Linear Unit).

Defining error analysis function and optimizer function

```
#ERROR ANALYSIS FUNCTION, Measure of deviation of predictions and actual using Mean Squared Error
MSE = tf.reduce_mean(tf.squared_difference(layer_output, outs))
trainMSE = []
testMSE = []
#OPTIMISER RATE TO DECREASE THE MSE, using Adaptive Moment Estimation Optimizer (default for deep learning dev)
MSE_dec = tf.train.AdamOptimizer().minimize(MSE)
```

The error analysis function is used to measure the deviation between the real values and the predicted values. **MSE** is commonly used. The optimizer is used to adapt the network's weight and bias variables during training. The optimizer being used is Adaptive Moment Estimation is the default.

Fitting the network and training

```
#TRAINING WITH DIFFERENT SIZED BATCHES FOR EACH EPOCH
for epoch in range(10):
    #GENERATE SHUFFLED TRAINING DATA
    size = len(y_train)
    batch_range = size //256
    random = np.random.permutation(np.arange(size))
    X_train = X_train[random]
    y_train = y_train[random]
    for x in range(0, batch_range):
        #TRAIN AND RUN THE BATCH AND MINIMIZE MSE
        X_batch = X_train[(256*x):(256*x)+256]
        Y_batch = y_train[(256*x):(256*x)+256]
        session.run(MSE_dec, feed_dict={ins:X_batch, outs:Y_batch})

    #DISPLAY PLOT EVERY 50th BATCH
    if(np.mod(x, 50) == 0):
        #RUN A PREDICTION ON THE DATA
        prediction = session.run(layer_output, feed_dict={ins: X_test})
        pred_line.set_ydata(prediction)
        plt.pause(0.01)
```

During this training, random test samples are created and used for the **X_train** and **y_train**. These batches are fed into the network through the **ins** placeholder. This batch flows through to the output layer where the predictions are compared with the values in the **outs** placeholder. The data is then optimized, and the weights and biases are then updated. This cycle is repeated for the next batches until all batches are processed. When all batches are processed, that is one epoch. For every 5th batch the network is visualized.

RESULTS

The following displays results for the algorithm run with different activation functions:

layer1_neurons = 1000 # double input size
layer2_neurons = 500 # 50% of previous layer
layer3_neurons = 250 # 50% of previous layer
layer4_neurons = 125 # 50% of previous layer

ACTIVATION FUNCTION: ReLU

```
-----  
MSE for test data: 0.004365  
Accuracy on test data: 0.995635000575185
```

layer1_neurons = 500 # double input size
layer2_neurons = 250 # 50% of previous layer
layer3_neurons = 125 # 50% of previous layer
layer4_neurons = 100

ACTIVATION FUNCTION: ReLU

```
-----  
MSE for test data: 0.004756349  
Accuracy on test data: 0.9952436508610845
```

layer1_neurons = 2000 # double input size
layer2_neurons = 1000 # 50% of previous layer
layer3_neurons = 500 # 50% of previous layer
layer4_neurons = 250 # 50% of previous layer

ACTIVATION FUNCTION: ReLU

```
-----  
MSE for test data: 0.0020091033  
Accuracy on test data: 0.9979908966924995
```

layer1_neurons = 2000 # double input size
layer2_neurons = 1000 # 50% of previous layer
layer3_neurons = 500 # 50% of previous layer
layer4_neurons = 250 # 50% of previous layer

ACTIVATION FUNCTION: sigmoid

```
-----  
MSE for test data: 0.007297084  
Accuracy on test data: 0.9927029157988727
```

layer1_neurons = 500 # double input size
layer2_neurons = 250 # 50% of previous layer
layer3_neurons = 125 # 50% of previous layer
layer4_neurons = 100

ACTIVATION FUNCTION: sigmoid

```
-----  
MSE for test data: 0.0070824847  
Accuracy on test data: 0.9929175153374672
```

layer1_neurons = 1000 # double input size
layer2_neurons = 500 # 50% of previous layer
layer3_neurons = 250 # 50% of previous layer
layer4_neurons = 125 # 50% of previous layer

ACTIVATION FUNCTION: sigmoid

```
-----  
MSE for test data: 0.005050706  
Accuracy on test data: 0.9949492937885225
```

layer1_neurons = 2000 # double input size
layer2_neurons = 1000 # 50% of previous layer
layer3_neurons = 500 # 50% of previous layer
layer4_neurons = 250 # 50% of previous layer

ACTIVATION FUNCTION: tanh

```
MSE for test data: 0.022667188
Accuracy on test data: 0.9773328118026257
```

layer1_neurons = 1000 # double input size
layer2_neurons = 500 # 50% of previous layer
layer3_neurons = 250 # 50% of previous layer
layer4_neurons = 125 # 50% of previous layer

ACTIVATION FUNCTION: tanh

```
MSE for test data: 0.004171451
Accuracy on test data: 0.9958285489119589
```

layer1_neurons = 500 # double input size
layer2_neurons = 250 # 50% of previous layer
layer3_neurons = 125 # 50% of previous layer
layer4_neurons = 100

ACTIVATION FUNCTION: tanh

```
MSE for test data: 0.0037012252
Accuracy on test data: 0.9962987748440355
```

CONCLUSION

Looking at the results above, decreasing the number of neurons in each layer does not necessarily increase the accuracy on test data. It is clear that the least effective activation function was the *sigmoid* function. *tanh* was an average activation function but *ReLU* produced the best accuracy results.

The combination of an activation function and number of neurons for each layer that yields the best accuracy results is the following:

Activation Function: ReLU

layer1_neurons = 2000, layer2_neurons = 1000, layer3_neurons = 500, layer4_neurons = 250

Accuracy on Test Data: 0.9979908966924995

(the progress of the plot of actual and predicted lines as the NN trains and tests is shown in the PowerPoint presentation)

The NN quickly adapts to the line of the actual stock prices and continues to find and learn finer patterns of the data. The optimizer works to reduce the learning rate as the model trains so that it can accurately reach the maximum accuracy without a chance of overshooting. Following the 10 epochs, the data is pretty much close to a perfect fit. The final MSE is **0.0020091033** which is extremely low.

FULL CODE

```
1. #Stock_Prediction
2.
3. #IMPORT
4. import numpy as np
5. import pandas as pd
6. import tensorflow as tf
7. import matplotlib.pyplot as plt
8. from sklearn.preprocessing import MinMaxScaler
9.
10.
11. #IMPORT THE DATA FILE, REMOVE THE 'DATE' COLUMN FROM DATASET
12. dataset = pd.read_csv('data_stocks.csv')
13. dataset = dataset.drop(['DATE'], 1)
14.
15. #DATASET VARIABLES, 'num_data' = NUMBER OF DATA POINTS, 'num_const' = NUMBER OF STOCKS
16. num_data = dataset.shape[0]
17. num_const = dataset.shape[1]
18.
19. #MAKE DATASET np.array
20. dataset = dataset.values
21.
22. #SPLIT DATASET INTO 80% FOR TRAINING DATA AND 20% FOR TESTING DATA
23. #TRAINING DATA, 80%
24. traindata = dataset[np.arange(0, int(np.floor(0.8*num_data))), :]
25. #TESTING DATA, 20%
26. testdata = dataset[np.arange(int(np.floor(0.8*num_data))+1, num_data), :]
27.
28. #SCALE DATASET USING MinMaxScaler WITH VALUES BEING IN THE RANGE OF (-1,1)
29. scaler = MinMaxScaler(feature_range=(-1, 1))
30. scaler.fit(traindata)
31.
32. #SCALE BOTH THE TRAINING AND THE TESTING DATASET
33. traindata = scaler.transform(traindata)
34. testdata = scaler.transform(testdata)
35.
36. #SPLIT DATASET INTO X AND y TRAIN AND TEST
37. # X_train, X_test, etc = are arrays
38. X_train = traindata[:, 1:]
39. X_test = testdata[:, 1:]
40. y_train = traindata[:, 0]
41. y_test = testdata[:, 0]
42.
43. #RETRIEVING NUMBER OF STOCKS IN SPLIT TRAINING DATASET
44. num_stocks = X_train.shape[1]
45.
46. #SETTING UP TensorFlow MODEL, ABSTRACT REPRESENTATION OF NN THROUGH placeholders AND variables
47. # placeholders, 'ins' = inputs (stock prices of all S&P 500 stocks), 'outs' = outputs (stock price of the S&P 500)
48. # variables, layeri_neurons = number of neurons on layer i, layeri_weight = weight for layer i, layeri_bias = bias for layer i
49.
50. # layeri_neurons, -----TRY OUT DIFFERENT NUMBER OF NEURONS-----
51. layer1_neurons = 2000 # double input size
52. layer2_neurons = 1000 # 50% of previous layer
```

```

53. layer3_neurons = 500 # 50% of previous layer
54. layer4_neurons = 250 # 50% of previous layer
55.
56. session = tf.InteractiveSession()
57.
58. # placeholders
59. ins = tf.placeholder(dtype=tf.float32, shape=[None, num_stocks])
60. outs = tf.placeholder(dtype=tf.float32, shape=[None])
61. ### WEIGHT AND BIAS INITIALIZERS using default initialization strategy ###
62. weight_initializer = tf.variance_scaling_initializer(mode="fan_avg", distribution="uniform", scale=1)
63. bias_initializer = tf.zeros_initializer()
64. #####
65. # layer1_weight, layer1_bias
66. layer1_weight = tf.Variable(weight_initializer([num_stocks, layer1_neurons]))
67. layer1_bias = tf.Variable(bias_initializer([layer1_neurons]))
68. layer2_weight = tf.Variable(weight_initializer([layer1_neurons, layer2_neurons]))
69. layer2_bias = tf.Variable(bias_initializer([layer2_neurons]))
70. layer3_weight = tf.Variable(weight_initializer([layer2_neurons, layer3_neurons]))
71. layer3_bias = tf.Variable(bias_initializer([layer3_neurons]))
72. layer4_weight = tf.Variable(weight_initializer([layer3_neurons, layer4_neurons]))
73. layer4_bias = tf.Variable(bias_initializer([layer4_neurons]))
74. output_weight = tf.Variable(weight_initializer([layer4_neurons, 1]))
75. output_bias = tf.Variable(bias_initializer([1]))
76. #NN ARCHITECTURE AND ACTIVATION FUNCTION (ReLU) -----
    TRY OUT DIFFERENT ACTIVATION FUNCTIONS-----
77. layer1 = tf.nn.relu(tf.add(tf.matmul(ins, layer1_weight), layer1_bias))
78. layer2 = tf.nn.relu(tf.add(tf.matmul(layer1, layer2_weight), layer2_bias))
79. layer3 = tf.nn.relu(tf.add(tf.matmul(layer2, layer3_weight), layer3_bias))
80. layer4 = tf.nn.relu(tf.add(tf.matmul(layer3, layer4_weight), layer4_bias))
81. layer_output = tf.transpose(tf.add(tf.matmul(layer4, output_weight), output_bias))
82.
83. #ERROR ANALYSIS FUNCTION, Measure of deviation of predictions and actual using Mean Squared Error
84. MSE = tf.reduce_mean(tf.squared_difference(layer_output, outs))
85. #OPTIMISER RATE TO DECREASE THE MSE, using Adaptive Moment Estimation Optimizer (default for deep learning dev)
86. MSE_dec = tf.train.AdamOptimizer().minimize(MSE)
87.
88. #SETTING UP NN SESSION AND PLOT
89. session.run(tf.initializers.global_variables()) #initialise global variables in plot
90. plt.ion() #turning on interactive mode
91. graph = plt.figure() #create new plot
92. grid_param = graph.add_subplot(111) #subplot grid parameter
93. real_line, = grid_param.plot(y_test)
94. pred_line, = grid_param.plot(y_test * 0.5)
95. plt.show()
96.
97.
98. #num_in_batch = 256
99.
100.
101. #TRAINING WITH DIFFERENT SIZED BATCHES FOR EACH EPOCH
102. for epoch in range(10):
103.     #GENERATE SHUFFLED TRAINING DATA
104.     size = len(y_train)
105.     batch_range = size //256
106.     random = np.random.permutation(np.arange(size))
107.     X_train = X_train[random]
108.     y_train = y_train[random]
109.     for x in range(0, batch_range):

```

```
110.         #TRAIN AND RUN THE BATCH AND MINIMIZE MSE
111.         X_batch = X_train[(256*x):((256*x)+256)]
112.         Y_batch = y_train[(256*x):((256*x)+256)]
113.         session.run(MSE_dec, feed_dict={ins:X_batch, outs:Y_batch})
114.
115.         #DISPLAY PLOT EVERY 50th BATCH
116.         if(np.mod(x, 50) == 0):
117.             #RUN A PREDICTION ON THE DATA
118.             prediction = session.run(layer_output, feed_dict={ins: X_test})
119.             pred_line.set_ydata(prediction)
120.             plt.pause(0.01)
121.
122.
123.         #DISPLAY MSE AND TEST SCORE ACCURACY FOR TEST DATA
124.         MSE_test = session.run(MSE, feed_dict={ins:X_test, outs: y_test})
125.         print("-----")
126.         print("\tMSE for test data: ", MSE_test)
127.         print("\tAccuracy on test data: ", 1-MSE_test)
```