CSC 158
PROJECT 2

Training a Neural Network to Predict Stock Prices

BY: YOUSSEF ELMOUGY

1

PROBLEM DESCRIPTION

Utilize a huge dataset containing stock prices for the S&P 500 Index and its constituents to precisely predict the next minute stock price of the S&P 500 Index

2 DATA DESCRIPTION

◇ Dataset contains 41,266 minutes of data ranging from April 2017 to August 2017 on prices of the 500 stock constituents along with the total S&P 500 index price

	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())`

◇ Each row of the dataset contains the constituent 500 stock's prices at time ***T* = t** and the stock price of the S&P 500 at ***T* = t + *1***

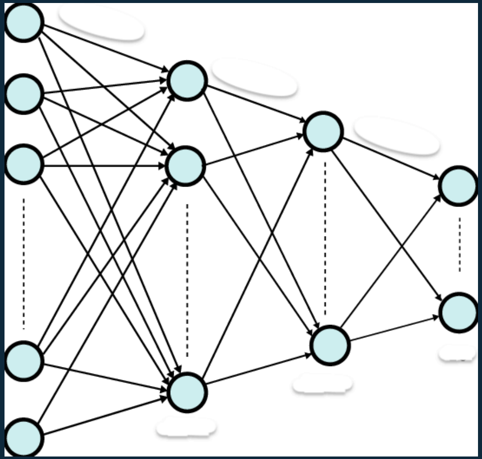
3

METHOD DESCRIPTION

This deep learning model is built with TensorFlow

The dataset is split into 80% used as the training data, and 20% used as the testing data. Both the training and testing data are scaled using sklearn's ***MinMaxScaler()*** and bounded within the range [-1, 1]

The TensorFlow model consists of four layers. The number of neurons in each layer is adjusted to find the architecture that produces the best accuracy.



Each subsequent layer's number of neurons is always half the number of neurons of the previous layer

Information is compressed as it flows between layers hence creating a more reliable accuracy

The model is represented through placeholders and variables:

- *placeholders*
  - ***ins*** : contains the NN’s inputs (the stock prices of the 500 constituents)
  - ***outs*** : contains the NN’s outputs (the stock price of the S&P 500)

- *variables – each layer, including the output layer, has a unique set of variables:*
  - *Weight variable : each layer passes its output as the input of the next layer*
  - Bias variable : the number of neurons in the layer

The placeholders and variables are then combined to design the architecture of the neural network

The NN is then fitted and trained using adjustable sized batches

For each batch,

- The error is calculated using the Mean Squared Error (MSE) approach
- An optimizer is applied to minimize the MSE
- The predictions of the NN are plotted against the actual stock prices

4

Implementation

Details on how the algorithm was implemented

- Importing the data
- Splitting the dataset into training data and testing data
- Scaling the dataset
- Defining the placeholders
- Defining the variables
- Defining the activation function
- Defining error analysis function and optimizer function
- Fitting the network and training

Importing the data → Splitting the dataset into training data and testing data → Scaling the dataset → Defining the placeholders → Defining the variables → Defining the activation function → Defining error analysis function and optimizer function → Fitting the network and training

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

```
#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), :]
```

```
#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)
```

Importing the data

Splitting the dataset into training data and testing data

Scaling the dataset

Defining the placeholders

Defining the variables

Defining the activation function

Defining error analysis function and optimizer function

Fitting the network and training

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

- Importing the data
- Splitting the dataset into training data and testing data
- Scaling the dataset
- Defining the placeholders
- **Defining the variables**
- Defining the activation function
- Defining error analysis function and optimizer function
- Fitting the network and training

```
# layeri_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
```

```
# layeri_weight, layeri_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]))
```

Importing the data → Splitting the dataset into training data and testing data → Scaling the dataset → Defining the placeholders → Defining the variables → **Defining the activation function** → Defining error analysis function and optimizer function → Fitting the network and training

```
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))
```

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

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

1. Importing the data
2. Splitting the dataset into training data and testing data
3. Scaling the dataset
4. Defining the placeholders
5. Defining the variables
6. Defining the activation function
7. **Defining error analysis function and optimizer function**
8. Fitting the network and training

```
#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)
```

Importing the data → Splitting the dataset into training data and testing data → Scaling the dataset → Defining the placeholders → Defining the variables → Defining the activation function → Defining error analysis function and optimizer function → 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)
```

5

RESULTS

Layer1 = 1000, layer2 = 500, layer3 = 250, layer4 = 125

```
MSE for test data:  0.004365
Accuracy on test data:  0.9956350000575185
```

Layer1 = 500, layer2 = 250, layer3 = 125, layer4= 100

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

Layer1 = 2000, layer2 = 1000, layer3 = 500, layer4= 250

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

Activation Function: ReLU

Layer1 = 2000, layer2 = 1000, layer3 = 500, layer4 = 250

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

Layer1 = 500, layer2 = 250, layer3 = 125, layer4= 100

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

Layer1 = 1000, layer2 = 500, layer3 = 250, layer4= 125

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

Activation Function: sigmoid

Layer1 = 2000, layer2 = 1000, layer3 = 500, layer4 = 250

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

Layer1 = 1000, layer2 = 500, layer3 = 250, layer4= 125

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

Layer1 = 500, layer2 = 250, layer3 = 125, layer4= 100

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

Activation Function: tanh

6 CONCLUSION

- Decreasing the number of neurons in each layer does not necessarily increase the accuracy on test data
- Effectiveness (greatest accuracy on test data):

sigmoid < **tanh** < **ReLU**

The architecture combination that yields to the best accuracy results is the following:

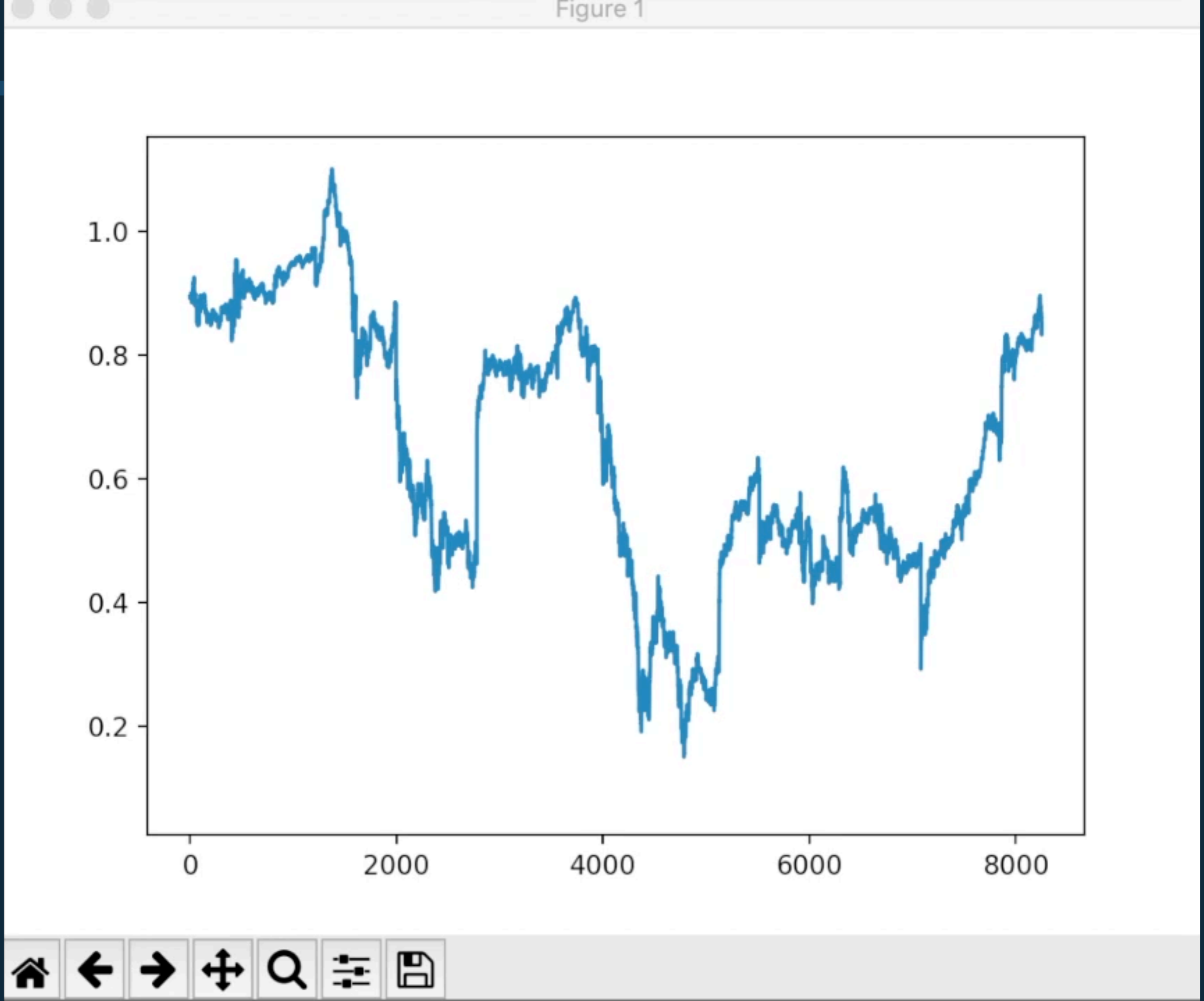
Activation Function: ReLU

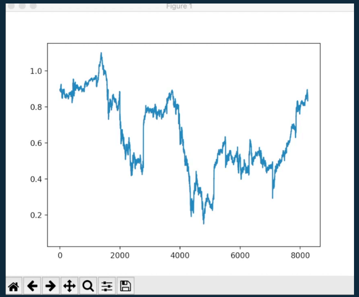
layer1 = 2000, layer2 = 1000, layer3 = 500, layer4 = 250

Accuracy on test data: 0.9979908966924995

Actual Stock Prices

Predicted Stock Prices





- The NN quickly adapts and continues to find and learn finer patterns of the data
- The optimizer works to reduce the learning rate as the model trains
  - Reduces the chance of overshooting maximum accuracy
- After 10 epochs, the data was pretty much close to a perfect fit
  - **Final MSE = 0.0020091033**

Thanks!