ECE62900-Fall 2018

NEURAL NETWORK

FINAL PROJECT

Time series Forecasting Using LSTM and Testing on Financial data

By- Varun Vallabhan
Project mentor- Prof. Okan k. Ersoy

Abstract:

Time Series model is purely dependent on the idea that past behavior and price patterns can be used to predict future price behavior. In this project we have discussed the drawbacks of different kinds of Networks and how LSTM gives better results for the same. First part of this paper talks about different models and LSTM. The second part discusses an example from MATLAB deep learning toolbox, Finally, an implementation for the NASDAQ data, using Opening, High, Low, Volumes as features to predict the closing.

Time Series and Different Networks.

What is Time series data?

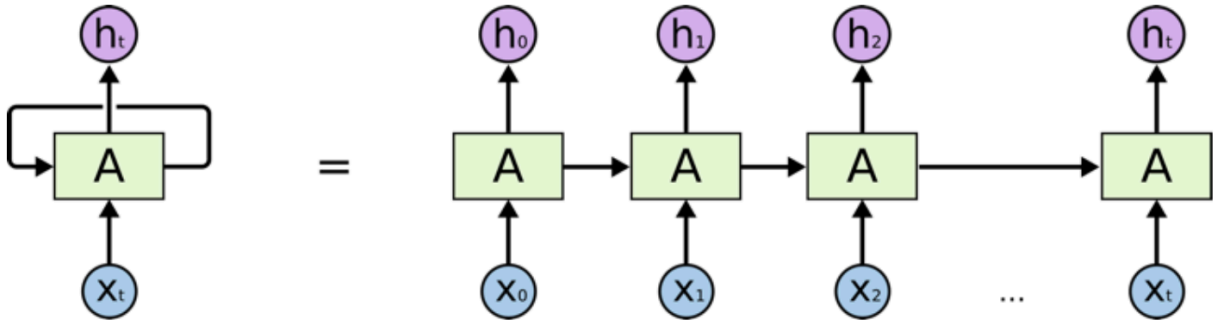
Time series is the data that, collectively represents how a system/process/behavior changes over time. Which means that data is continuously monitored and appended to the set. This can be climate data, stock market data, flight usage stats, water consumption/ electricity consumption throughout the year etc. While time series in general is very difficult to deal with, Our use case, the Stock market data and its forecasting is a different ball game altogether. Stock market data is known to be highly volatile and erratic. Which makes it more difficult to predict or find a pattern.

General RNN VS LSTM.

The decision a recurrent net reached at time step t-1 affects the decision it will reach one moment later at time step t. So recurrent networks have two sources of input, the present and the recent past, which combine to determine how they respond to new data, much as we do in life.
Recurrent networks are distinguished from feedforward networks by that feedback loop connected to their past decisions, ingesting their own outputs moment after moment as input. It is often said that recurrent networks have memory. Sequential information is preserved in the recurrent network's hidden state, which manages to span many time steps as it cascades forward to affect the processing of each new example.

Why it fails-

Even though the RNN have memory and work with the output to predict the next data, they iterate on the recent past and that is their main drawback as they can't predict something that happened long time ago. And if we try to incorporate larger memory by adding more layers the entire network collapses due to the **Vanishing and exploding gradie**nts. During the training of RNN, as the information goes in loop again and again which results in very large updates to neural network model weights. This is due to the accumulation of error gradients during an update and hence, results in an unstable network. At an extreme, the values of weights can become so large as to overflow and result in NaN values.The explosion occurs through exponential growth by repeatedly multiplying gradients through the network layers that have values larger than 1 or vanishing occurs if the values are less than 1.
Exploding gradients become saturated on the high end; i.e. they are presumed to be too powerful. But exploding gradients can be solved relatively easily, because they can be truncated or squashed. Vanishing gradients can become too small for computers to work with or for networks to learn – a harder problem to solve.

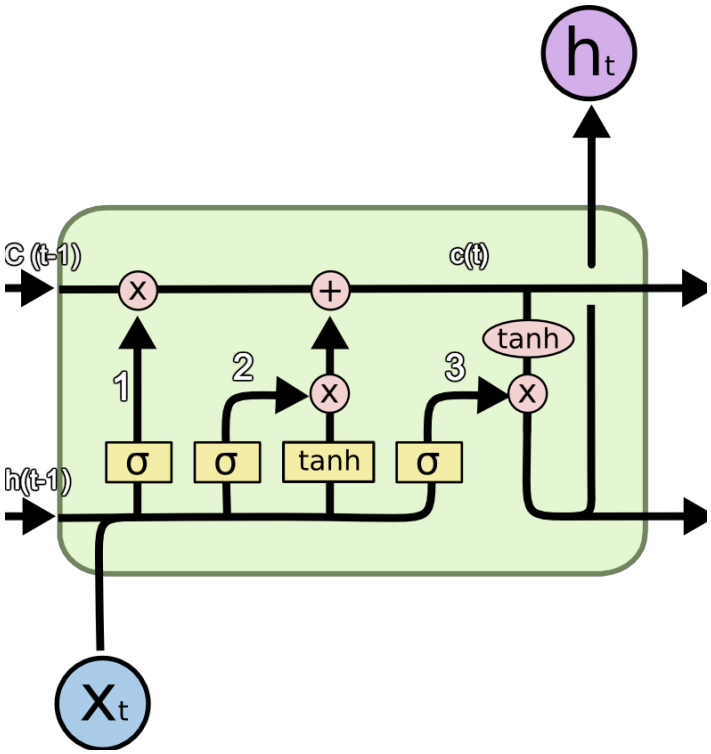


This an example of a Crude RNN to show how layers are linked, this adds more complexity as we include more and more states, and how model h(t) depends on the previous states.

SOLUTION?

The solution to this problem is to remember essential information and forget the rest after some time similar to human memory. Like, we remember life events clearly and in detail, but we do forget the cab number that we saw while booking, the previous day. This is exactly what is implemented **LSTM(Long Short term Memory).** LSTM uses gates to control the memorizing process unlike conventional RNN.

The figure below shows the architecture of a LSTM network.



The symbols used here have following meaning:

a) X : Scaling of information
b) + : Adding information
c) σ : Sigmoid layer
d) tanh: tanh layer
e) h(t-1) : Output of last LSTM unit
f) c(t-1) : Memory from last LSTM unit
g) X(t) : Current input
h) c(t) : New updated memory
i) h(t) : Current output
j). 1,2,3 are the steps that are followed in LSTM cell

LSTMs contain information outside the normal flow of the recurrent network in a gated cell. Information can be stored in, written to, or read from a cell, much like data in a computer's memory. The cell makes decisions about what to store, and when to allow reads, writes and erasures, via gates that open and close. Unlike the digital storage on computers, however, these gates are analog, implemented with element-wise multiplication by sigmoids, which are all in the range of 0-1. Analog has the advantage over digital of being differentiable, and therefore suitable for backpropagation.
Those gates act on the signals they receive, and similar to the neural network's nodes, they block or pass on information based on its strength and import, which they filter with their own sets of weights. Those weights, like the weights that modulate input and hidden states, are adjusted via the recurrent networks learning process. That is, the cells learn when to allow data to enter, leave or be deleted through the iterative process of making guesses, backpropagating error, and adjusting weights via gradient descent.

Steps involved in LSTM

The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at ht−1 and xt, and outputs a number between 0 and 1 for each number in the cell.

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, C̃ t, that could be added to the state. In the next step, we'll combine these two to create an update to the state.

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

It's now time to update the old cell state, Ct−1, into the new cell state Ct. The previous steps already decided what to do, we just need to actually do it. We multiply the old state by *ft*, forgetting the things we decided to forget earlier. Then we add $it * \tilde{C}\, t$. This is the new candidate values, scaled by how much we decided to update each state value.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Finally, we need to decide what we're going to output. This output will be based on our cell state but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

$$o_t = \sigma\left(W_o \; [\, h_{t-1}, x_t] \; + \; b_o\right)$$

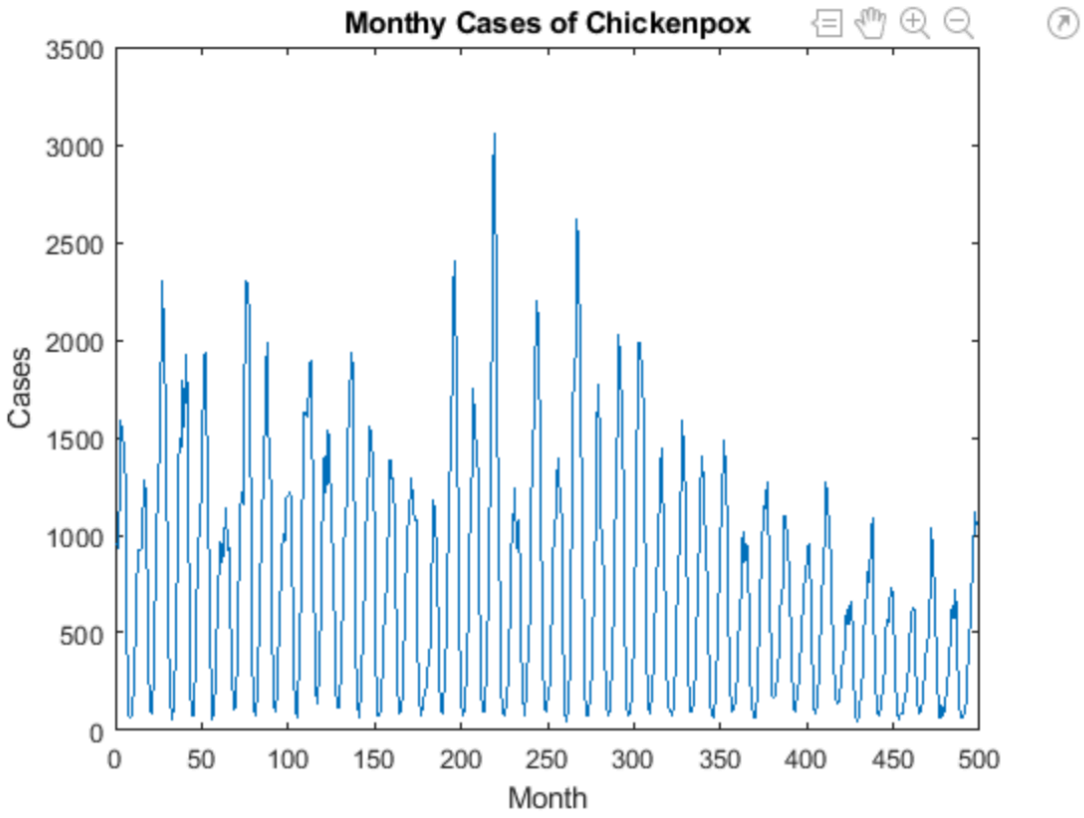
$$h_t = o_t * \tanh\left(C_t\right)$$

Studying LSTM using MATLAB deep learning toolbox.

We study the Lstm example from the deep learning toolbox. Here the dataset used is the chickenpox dataset. Here the sequence to sequence regression LSTM network training is done, where the responses are the training sequences with values shifted by one-time step. That is, at each time step of the input sequence, the LSTM network learns to predict the value of the next time step. Training is done by using the **predictandupdate** routine which predict time steps one at a time and update the network state at each prediction.

Following are the results and outputs observed by running the example.

```
plot(data)
xlabel("Month")
ylabel("Cases")
title("Monthy Cases of Chickenpox")
```



This is the plot of the data set and the dataset is then split into 9:1 ratio for train and test. And after splitting the data is then standardized by subtracting the mean and dividing by the standard deviation.

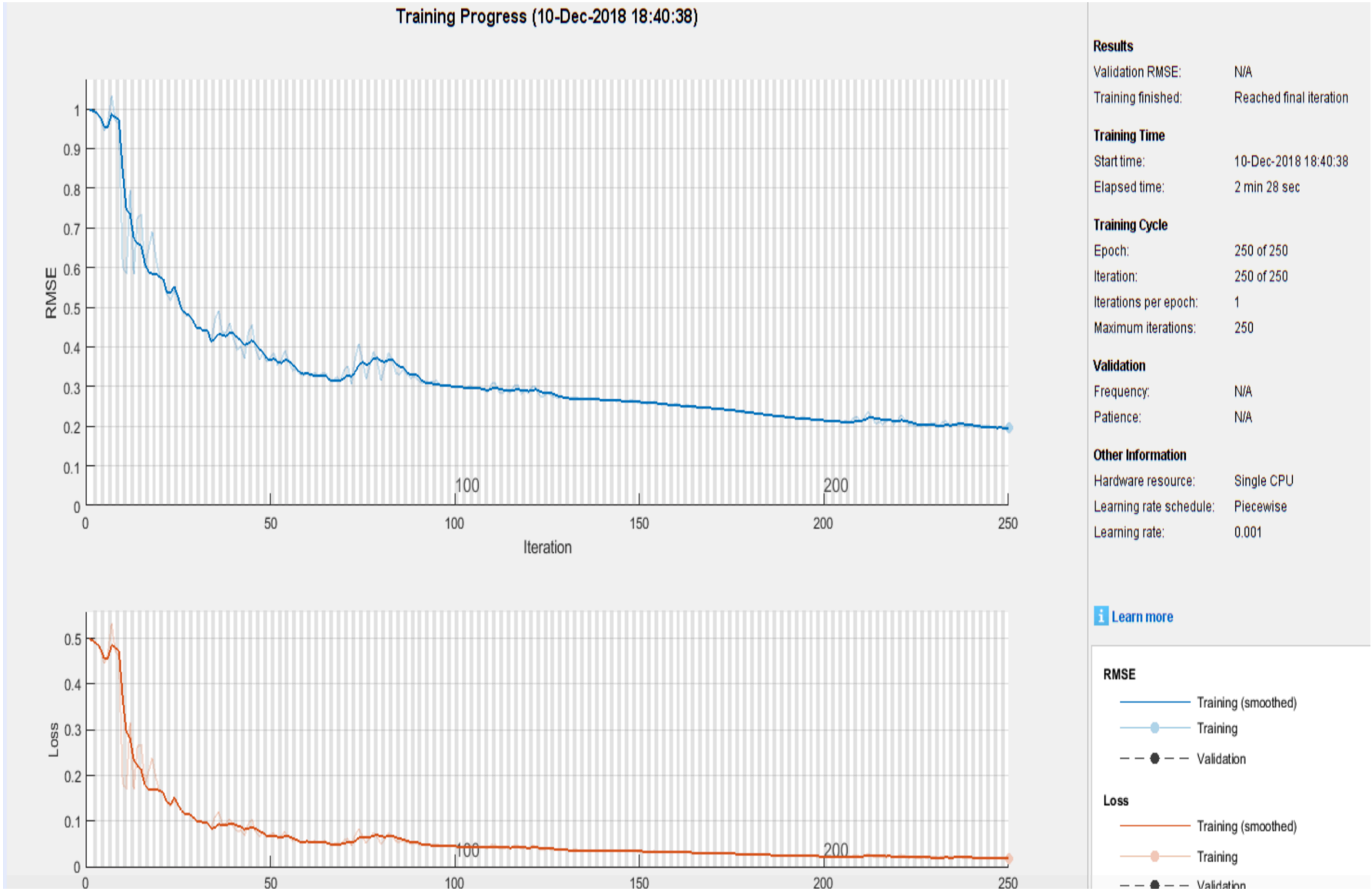
```
inputSize = 1;
numResponses = 1;
numHiddenUnits = 200;

layers = [ ...
    sequenceInputLayer(inputSize)
    lstmLayer(numHiddenUnits)
    fullyConnectedLayer(numResponses)
    regressionLayer];
```

These are the network parameters set for defining the LSTM network.

```
opts = trainingOptions('adam', ...
    'MaxEpochs',250, ...
    'GradientThreshold',1, ...
    'InitialLearnRate',0.005, ...
    'LearnRateSchedule','piecewise', ...
    'LearnRateDropPeriod',125, ...
    'LearnRateDropFactor',0.2, ...
    'Verbose',0, ...
    'Plots','training-progress');
```

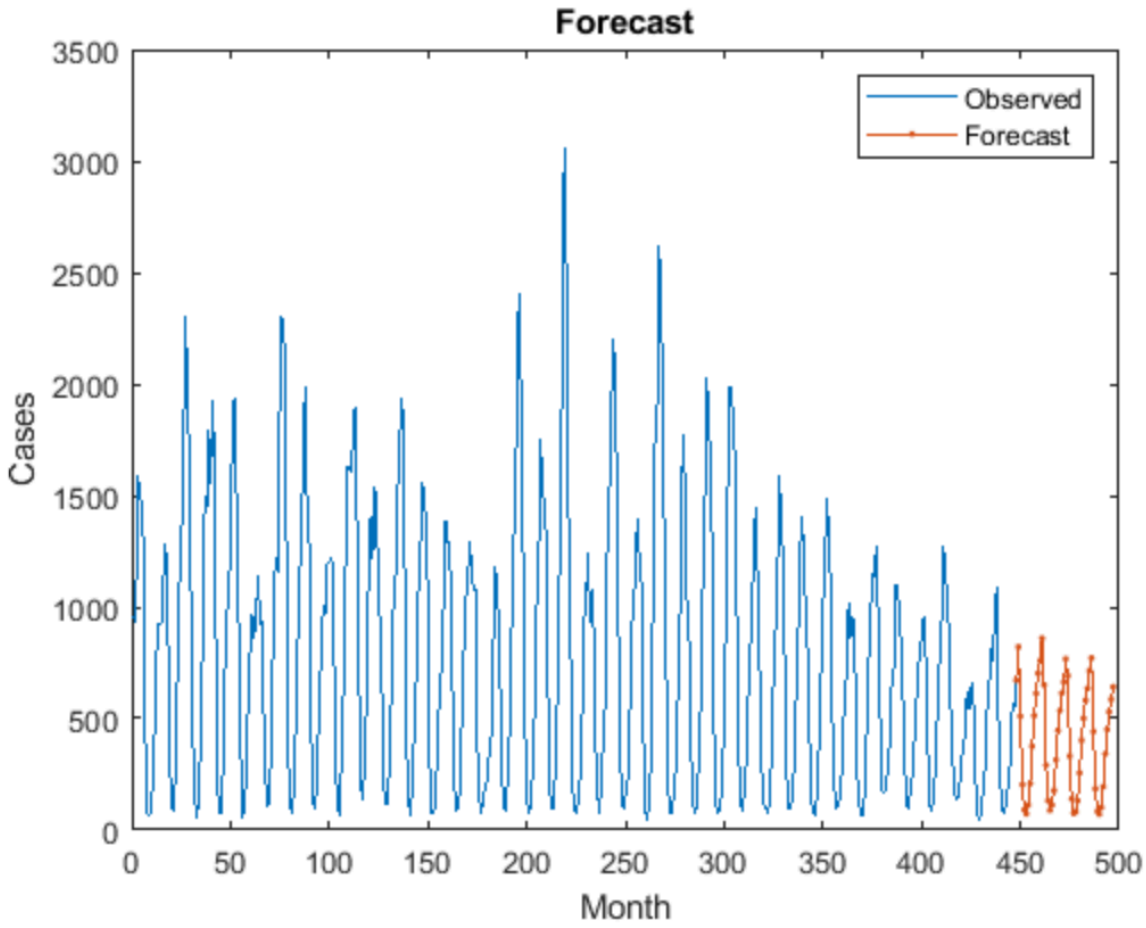
These are the training parameters used for training the above given network. These generally include the learning rate, number of epoch, etc.

The figure below shows the training RMSE and loss.


Training Progress (10-Dec-2018 18:40:38)
RMSE
Iteration
Loss
Results
Validation RMSE: N/A
Training finished: Reached final iteration
Training Time
Start time: 10-Dec-2018 18:40:38
Elapsed time: 2 min 28 sec
Training Cycle
Epoch: 250 of 250
Iteration: 250 of 250
Iterations per epoch: 1
Maximum iterations: 250
Validation
Frequency: N/A
Patience: N/A
Other Information
Hardware resource: Single CPU
Learning rate schedule: Piecewise
Learning rate: 0.001
Learn more
RMSE
Training (smoothed)
Training
Validation
Loss
Training (smoothed)
Training
Validation

```
net = predictAndUpdateState(net,XTrain);
[net,YPred] = predictAndUpdateState(net,YTrain(end));

numTimeStepsTest = numel(XTest);
for i = 2:numTimeStepsTest
    [net,YPred(1,i)] = predictAndUpdateState(net,YPred(i-1));
end
```

As mentioned earlier, we use predict and update to predict the next value given a current point. What this essentially does is that it first predicts the value for a given input and updates the network to be prepared for the next iteration.



This is the forecast provided by the model which is the output for the given model.
And has a RMSE of 194.34.

As we can see that the Lstm network actually follows the trends observed in the input data, where a traditional model or a regular regressor would have failed.

Python Implementation using Keras for NASDAQ data.

The following is the implementation of the LSTM model on NASDQ dataset. Which has the intrinsic features of the stock like opening, high, low, volume closing and not the regular data set which only has the closing values.

There are a few data preprocessing steps that need to be implemented. I have just normalized the data for now but, I will be mentioning a few data transformations which leads to better prediction for future scope. In case of such volatile data the preprocessing is in fact more important than the model parameters. Simply increasing the layers or number of neurons won't help and we need to dive deeper into financial data processing/management.

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import math
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense, Activation ,Dropout
from keras.layers import LSTM
from scipy.io import savemat
from sklearn.model_selection import train_test_split
```

These are the standard libraries that we might need for the code.
Including function like pandas for importing and processing data, Keras to create our LSTM, mean-square from scikitlearn to calculate the accuracy of the code by using mean-square to calculate root mean square. And a few additional libraries for saving and using a few math functions.

```
data_whole=pd.read_csv('drive/My Drive/vallabhan/Nasdaq.csv')
feature_cols = list(data_whole.columns[:-1])
target_col = data_whole.columns[-1]
X = data_whole[feature_cols].values
y = data_whole[target_col].values
#data=data_whole['ADBE']
#data.dropna(inplace=True)
plt.plot(y)
plt.show()

```

This segment loads the data from NASDAQ and splits into features and target.

Shown is the plot of the target values.
For the entire dataset.

```
np.random.seed(7)
scaler_x = MinMaxScaler(feature_range=(0,3))
X = scaler_x.fit_transform(X)
X = pd.DataFrame(X)
scaler_y=MinMaxScaler()
y=scaler_y.fit_transform(y.reshape(y.shape[0],1))
y=pd.DataFrame(y)
train_size = int(len(X) * 0.6)
test_size = len(X) - train_size
trainX, testX = X.values[0:train_size,:], X.values[train_size:len(X),:]
trainY, testY = y.values[0:train_size], y.values[train_size:len(X)]
trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
print(len(trainX), len(testX))
```

This segment has the data preprocessing we talked about. We first normalize the data and transform it for features and target. Then, we split the data into train and test in 60/40 ratio for both features and target.

```
model = Sequential()
model.add(LSTM(20, input_shape=(1, 4),return_sequences = False))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY,validation_split=0.1, epochs=100, batch_size=10, verbose=2)
```

This segment deals with modeling the layers we have a Lstm layer with 20 neurons as output and with 4 features as input for each guess. To converge this to a single output via dense layer we call the dense function and set its output dimension to be one as we need a single output guess for the given input. The model uses batch processing with batch size 10 and runs for 100 epochs. There is validation set which is 10% of the training set for better accuracy and to avoid overfitting. The loss parameter follows mean square and the Adam optimizer is used instead of the gradient descent as Adam optimizer is better than gradient descent and has better computation efficiency, Little memory requirements. Return sequences is set to be false in this scenario but it should be set to true if there is a cascade of LSTMs for passing the parameters to each layer from previous layer.

```
Train on 4426 samples, validate on 492 samples
Epoch 1/100
 - 4s - loss: 0.0013 - val_loss: 8.3185e-04
Epoch 2/100
 - 1s - loss: 2.5021e-04 - val_loss: 1.3253e-04
Epoch 3/100
 - 1s - loss: 2.3704e-04 - val_loss: 2.4229e-04
Epoch 4/100
 - 1s - loss: 2.3826e-04 - val_loss: 1.4288e-04
Epoch 100/100
 - 1s - loss: 2.3492e-04 - val_loss: 1.3576e-04
```

These are the run times observed in collab using settings which results in time taken between 4 seconds to 1 second per epoch.

```
Train on 4426 samples, validate on 492 samples
Epoch 1/100
 - 1s - loss: 0.0013 - val_loss: 8.0097e-04
Epoch 2/100
 - 0s - loss: 2.4928e-04 - val_loss: 1.3159e-04
Epoch 3/100
 - 0s - loss: 2.3734e-04 - val_loss: 2.4356e-04
Epoch 4/100
 - 0s - loss: 2.3834e-04 - val_loss: 1.4367e-04
Epoch 5/100
 - 0s - loss: 2.3350e-04 - val_loss: 1.2770e-04
Epoch 100/100
 - 0s - loss: 2.3491e-04 - val_loss: 1.3575e-04
```

These are the results result usinf the GPU settings in collab you can see that its faster than the CPU settings. For a complex data set this would result in a huge difference in the runtime of the training. Also, GPU presents slightly better accuracy which plays an important role as the dataset changes this yield better performance and better result in a complex problem and hence, we can conclude that GPU processing is more desirable as its faster, more accurate and reliable.

```
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
trainPredict = scaler_y.inverse_transform(trainPredict)
trainY = scaler_y.inverse_transform(trainY)
testPredict = scaler_y.inverse_transform(testPredict)
testY = scaler_y.inverse_transform(testY)
lst={}
lst['trainPredict']=trainPredict
lst['testPredict']=testPredict
lst['trainY']=trainY
lst['testY']=testY
savemat('lstm.mat',lst)
trainScore = math.sqrt(mean_squared_error(trainY, trainPredict[:,0]))
print('Train Score: %.2f RMSE' % (trainScore))
testScore = math.sqrt(mean_squared_error(testY, testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))
```

This segment deals with the prediction using the trained model. The predicted and the target values are used to calculate the test and train scores.

```
Train Score: 85.81 RMSE
Test Score: 138.98 RMSE
```

These are the values for CPU.

```
Train Score: 85.81 RMSE
Test Score: 139.00 RMSE
```

These are the values for GPU. As you can see that GPU processing yields better results as stated above.

```
trainPredictPlot = np.empty_like(y)
trainPredictPlot[:] = np.nan
trainPredictPlot[:len(trainPredict)] = trainPredict
# shift test predictions for plotting
testPredictPlot = np.empty_like(y)
testPredictPlot[:] = np.nan
testPredictPlot[len(trainPredict):len(y)]= testPredict
# plot baseline and predictions
plt.plot(scaler_y.inverse_transform(y))
plt.plot(trainPredict,'r')
plt.plot(testPredictPlot)
plt.show()
```

Finally to view and compare the result we scale the data back to its original format and plot them. BLUE is the original data, Red is the train predicted data, Green is the test predicted data.

For CPU.

For GPU. The plot looks nearly the same as the error difference is minute in this case.

Observations.

As we can see the predicted output follows the trends of the train and test data and the data fits the train curve as it should as we also use validation to make sure of it. The Test data deteriorates towards the and yet follows the trend pattern. Even though it loses its accuracy it still predits if it's a high or low which is the most desirable quality that we look for. Hence, we can use to predict High low values at any point given.

Future Scope.

As mentioned earlier in this case there are a few data preprocessing parameters which can be used to improve the prediction. As, this project is meant to understand LSTM and its ability to follow trend and its properties, the preprocessing part has been neglected. The few processing parameters are Moving Average, standard deviation, and momentum these help in smoothening out the curve to give better prediction. In this use case, the accuracy depends more on how we preprocess the data than on the complexity of the model. In fact increasing he complexity decreases the efficiency.
For example,

```
model = Sequential()
model.add(LSTM(50, input_shape=(1, 4),return_sequences = True))
model.add(Dropout(0.5))
model.add(Dense(150))
model.add(LSTM(100))
model.add(Dropout(0.5))
model.add(Activation("relu"))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(trainX, trainY,validation_split=0.2, epochs=100, batch_size=20, verbose=2)
```

This model would have given exemplar results, but in this case

```
Train Score: 109.51 RMSE
Test Score: 360.72 RMSE
```

The error increased, and the degradation was more, and it lost its ability to follow the trends in some cases.

References-

https://www.bioinf.jku.at/publications/older/2604.pdf
https://keras.io/layers/recurrent/
http://colah.github.io/posts/2015-08-Understanding-LSTMs/#fnref1
https://skymind.ai/wiki/lstm
https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/
https://www.datacamp.com/community/tutorials/lstm-python-stock-market