Risha Shah Follow

Apr 16, 2021 · 12 min read · Listen

Predicting Google's Stock Prices using LSTM Networks

Machines are able to predict future stock prices just like human investors. But more successfully.

Imagine living in a world where you know the exact prices of the stocks you invested in, but ahead of time. When you know exactly when they're heading up. When you know exactly when they're going down. You'll be on your path to becoming a millionaire.

This may sound superficial, because for so many years, investors have been doing a ton of

This could be done through **machine learning,** specifically, utilizing recurrent neural networks to predict the future opening prices of Google.

Understanding Recurrent Neural Networks (RNN)

RNNs are widely known for their ability to memorize previous inputs in memory, when a huge amount of sequential data is fed to them. These are different from the classic feedforward neural networks which have no memory of the input they received just a second ago.

For example, if you provide the feedforward neural network with a sequence of letters like "N-E-U-R-O-N", by the time it gets to 'R', it will most likely have forgotten that it just read "U". This is a big issue, and doesn't make these neural networks too useful.

However, RNNs continuously collect the inputs they just passed, and store them as well. For example, in "N-E-U-R-O-N", they will be able to remember "R" but also the "U" it just passed. This is very useful at the model will do a better job at predicted what is next in the sequence.

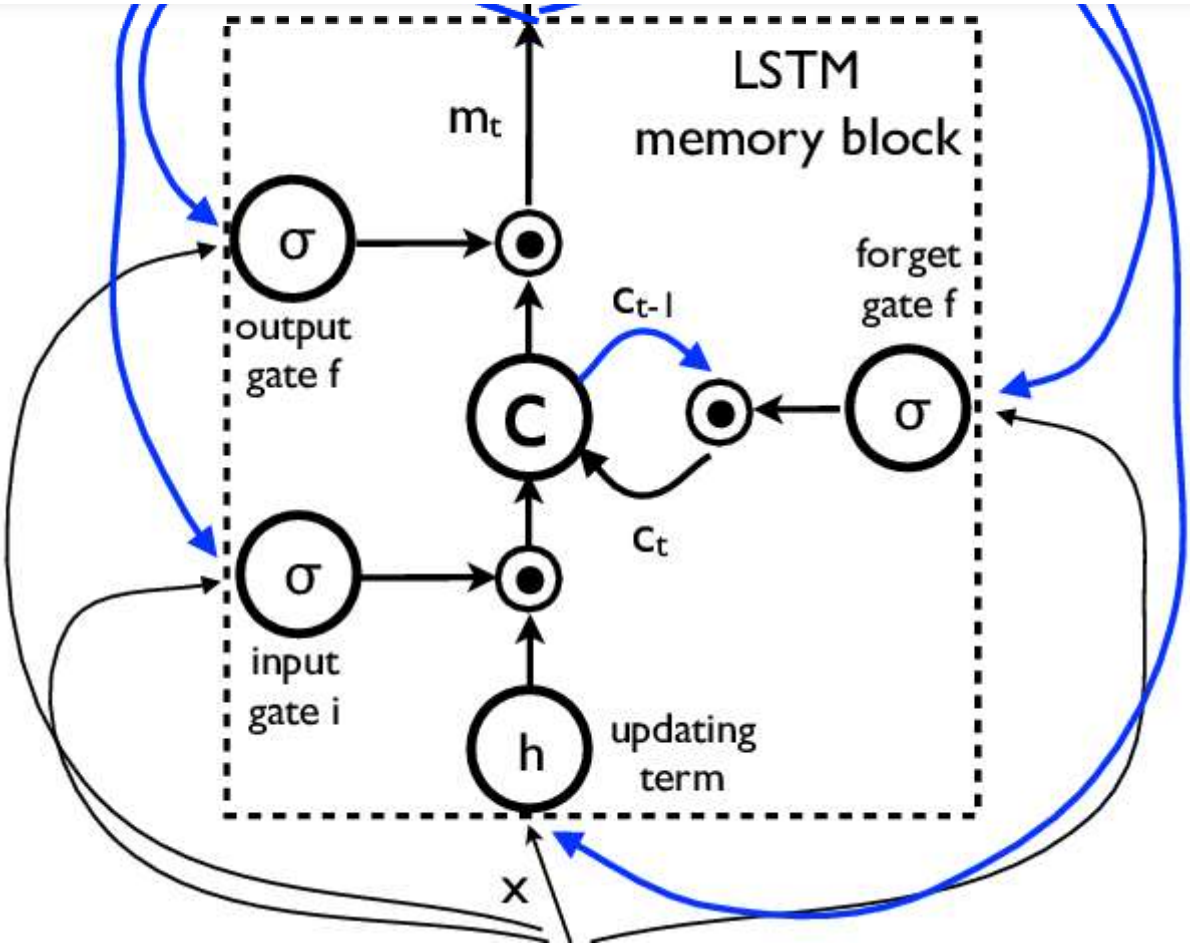
Long Short Term Memory (LSTM) Networks

LSTM networks have unique units along with their standard units, which have **memory cells**. It allows the network to maintain information in memory for an extensive period of time. The model includes multiple **gates** which control the time at which information enters the memory, when it is outputted, and when that information is disregarded.

To apply this new lens to stock predictions, a simple feed forward neural network would forget the earlier years stock prices as soon as the neural network is fed with more recent stock prices from this year.

However, with the LSTM network:

- The **memory gate** is able to collect potential outputs, and also stores the relevant ones.
- The **selection gate** is the one *selecting* the single output being sent out from the possible outputs that the memory gate had gathered.
- The **forget the ignore gate** is responsible for deciding which data memories are unnecessary and gets rid of them- as the name indicates.

Now that you've gained some background knowledge on RNNs, let's start building!

Importing Necessary Packages

Python libraries needed for this model include:

Numpy: used for performing basic array operations. It is widely used in training machine learning models as it supports multidimensional arrays compared to lists

Pyplot from matplotlib: required for visualizing the predicted values after testing it

Pandas: most commonly used to read the dataset

MinMaxScaler from sklearn: used to scale the data. It is used in the data normalization step and essentially what it does is, takes the large numbers, and transforms the values into the range between 0 to 1 range in order for the model to be trained.

```
import numpy as np
```

Dataset

This is the dataset I used to train the model. We're storing the content from the dataset into the variable called data. I also used the data_parser in order to convert the sequence of string columns into an array of datetime instances.

```
data = pd.read_csv('GOOG.csv', date_parser = True)
data.tail()
```

Splitting the Data

Training data is part of the dataset that we'll be training our model on. Think of it like teaching a child how to ride their bike. They fall down, they fail, but then they also learn, and that's how they train before riding their bike without training wheels. Once they're trained, they will be tested on the bike without training wheels. Similarly, I am splitting the dataset so that the data between 2004 and 2020 will be our training data. While the stock market prices during 2021, will be out testing data.

```
data_training = data[data['Date'] < '2021-01-01'].copy() data_training
```

Now, we will also drop the unnecessary columns in the training dataset, which don't have an impact on our predicted value. Since we are only focusing on the **opening price** of the stock, we can drop some columns. The date and adjusted closing price don't have an impact on the future opening price of this stock. This is why we will drop them.

It's also important to note that the function will search the y-axis by default, so that's why it is important to set **axis=1,** to make sure the function looks through the columns rather than the rows.

```
data_training = data_training.drop(['Date', 'Adj Close'], axis = 1)
```

the maximum value would be 1. Scaling it makes the run time shorter, while also reducing the error. This step is called the **normalization** of data.

```
scaler = MinMaxScaler()
data_training = scaler.fit_transform(data_training)
data_training
```

As mentioned above, we are going to train this model in data of 60 days at a time. Now, we will be dividing the data into chunks of 60 rows. First, let's make the variables that will hold the x and y training values.

I created a for loop so the data will began at 60 days, and perform the following code until the the data_training.shape[0]. The shape of the training data is basically the length of the dataset or our input.

Next, the 'i' will iterate from 60 to the length of the input (in this particular dataset, the length is 3617). To append, it will start from 'i -60' and go until i is actually equal to 60. Once again, we are splitting the entire dataset into chunks of 60. Similarly, we will do the same thing to the y training data. However, the parameters will be 'i, 0' in this case, which indicates the stock price on the 60th day.

```
x_train = []
Y_train = []
for i in range(60, data_training.shape[0]):
    x_train.append(data_training[i-60:i])
    y_train.append(data_training[i, 0])
```

Next, we are simply storing the x_train and y_train data into 2 different numpy arrays. Once again, this is beneficial as it supports multidimensional arrays.

```
x_train, y_train = np.array(x_train), np.array(y_train)
```

We can also check what our x_train data looks like. We see how it consists of 3557 chunks of data

```
X_train.shape
```

Output:

```
(3557, 60, 5)
```

Building the LSTM

I imported these necessary layers to build our recurrent neural network.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout
```

This next part may be difficult if you are not familiar with neural networks, but I will be explaining each block of code to make it easier to understand

We will be calling this model the regressor. This is because we will be continuously predicting the value.

First, we will be using the Sequential model as it is best to use this model when each layer will be provided with one input tensor, and will have one output tensor.

The Sequential model is *not* appropriate when:

- The model has multiple inputs or multiple outputs
- Any of the layers have multiple inputs or multiple outputs

Moving onto the layers:

The first layer of the LSTM layer which will have 60 units. The units are essentially memory units instead of each neuron. We will have 60 in our case.

The activation function that I used was the Rectified Linear Activation Function (ReLU). This is


function.

Next, the **reurn_sequences** is set to **True,** in order to return the full sequence in the output. In more technical terminology, the return_sequence will also output the hidden sequence as well. By default, the return_sequences is set to False in Keras RNN layers, so the RNN layer would simply only return the **last hidden state** output.

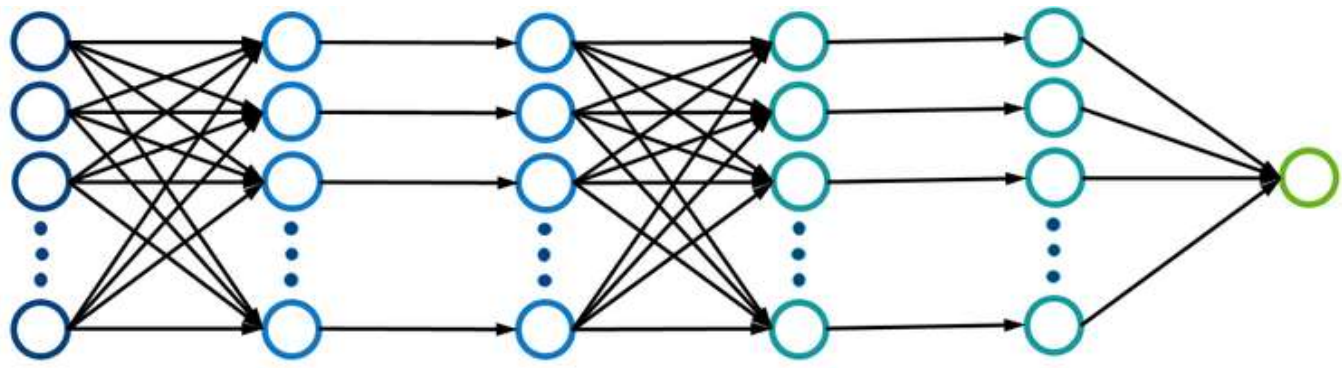
After that, the `input_shape` is set to `(X_train.shape[1],5)` which is [60,5]. This means for a single layer, the dimensions would be 60 by 5. Each row has a shape of 60 by 5 for the length of the training dataset.

The `Dropout layer` is used to prevent the model from overfitting, which essentially means that the model is limited only to the certain dataset, which makes the model not very flexible for other random inputs. To prevent this, the dropout layer (it is implemented for each layer as shown below), ignores units (neurons) during training randomly. It essentially randomly sets the outgoing edges of the hidden units to 0 in each layer. However, they are not completely ignored. Each value dropped specifies the probability at which outputs of the layer are dropped out at.

The last layer of this neural network is the `Dense layer`. by adding this dense layer, we are simply stating that the neurons from that layer are fully connected to the neurons in the previous layer.

Finally, since we are predicting a single value (the opening stock price), the number of units in the last/output layer should be set to 1.

This may be a bit hard to imagine, but referring to this diagram could help you fully understand the process of building this neural network:



```
regressor = Sequential()

regressor.add(LSTM(units = 60, activation = 'relu', return_sequences = True,
input_shape = (X_train.shape[1], 5)))
regressor.add(Dropout(0.2))

regressor.add(LSTM(units = 60, activation = 'relu', return_sequences = True))
regressor.add(Dropout(0.2))

regressor.add(LSTM(units = 80, activation = 'relu', return_sequences = True))
regressor.add(Dropout(0.2))

regressor.add(LSTM(units = 120, activation = 'relu'))
regressor.add(Dropout(0.2))

regressor.add(Dense(units = 1))
```

Run this code to see a summary of your model:

```
regressor.summary()
```

Output:

lstm_7 (LSTM)	(None, 60, 50)	11200
dropout_6 (Dropout)	(None, 60, 50)	0
lstm_8 (LSTM)	(None, 60, 60)	26640
dropout_7 (Dropout)	(None, 60, 60)	0
lstm_9 (LSTM)	(None, 60, 80)	45120
dropout_8 (Dropout)	(None, 60, 80)	0
lstm_10 (LSTM)	(None, 120)	96480
dropout_9 (Dropout)	(None, 120)	0
dense_1 (Dense)	(None, 1)	121

Total params: 179,561
Trainable params: 179,561
Non-trainable params: 0

Compiling the Model

Next, I compile the model. To do so, for the optimize, we use *adam*, which is the best optimizer for training deep learning models.

Also, the loss function will be the **mean squared error** comparing the observed values, to the predicted values.

The formula for the mean squared errors is:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

n indicated the number of data points.

Yi indicates the actual values, and the next one indicated the predicted value

Fit the Model

We will be using 50 epochs to train this model. Think of an epoch as the number of iterations/passes of the training data that the model has completed. Having 1 epoch for an entire dataset that is passed forward and backward would be too large, and would slow down your run time. Instead, we divide up into batch_size.

`batch_size` indicates the number of training samples used for each iteration through the neural network. In more technical terms, it is the number of samples per gradient update. That would mean the amount of times the weights will be updates after each training example. For smaller datasets, the batch size could be equivalent to the epoch, but that is most definitely not the case for this dataset. I chose the batch size in this model to be 32.

```
regressor.compile(optimizer='adam', loss = 'mean_squared_error')
regressor.fit(X_train, y_train, epochs=50, batch_size=32)
```

Compiling this code will take a while as each epoch is being iterated in our neural network

Output:

```
Train on 3557 samples

Epoch 45/50
3557/3557 [==============================] - 26s 7ms/sample - loss: 6.8088e-04
Epoch 46/50
3557/3557 [==============================] - 25s 7ms/sample - loss: 6.0968e-04
Epoch 47/50
3557/3557 [==============================] - 25s 7ms/sample - loss: 6.6604e-04
Epoch 48/50
3557/3557 [==============================] - 25s 7ms/sample - loss: 6.2150e-04
Epoch 49/50
3557/3557 [==============================] - 25s 7ms/sample - loss: 6.4292e-04
Epoch 50/50
3557/3557 [==============================] - 25s 7ms/sample - loss: 6.3066e-04
```

We are officially done training our model on the training data!!

Preparing the Test Dataset

Since we are predicting the stock price in terms of the previous 60 days, we will store the stock

```
past_60_days = data_training.tail(60)
```

Next, we will append through the test data and ignore the index. We will also drop the Date and Adj Close column as once again, it odes not affect the predicted value.

```
df = past_60_days.append(data_test, ignore_index = True)
df = df.drop(['Date', 'Adj Close'], axis = 1)
df.head()
```

Output:

	Open	High	Low	Close	Volume
0	1195.329956	1197.510010	1155.576050	1168.189941	2209500
1	1167.500000	1173.500000	1145.119995	1157.349976	1184300
2	1150.109985	1168.000000	1127.364014	1148.969971	1932400
3	1146.150024	1154.349976	1137.572021	1138.819946	1308700
4	1131.079956	1132.170044	1081.130005	1081.219971	2675700

Similar to what we did to the training data, we once again have to scale the test date to ensure it is between the range of 0 to 1.

```
inputs = scaler.transform(df)
inputs
```

Output:

As you can see, all the values are in the range of 0 to 1.


```
[0.90103881, 0.91345208, 0.89872289, 0.90204445, 0.02331778],
...,
[0.93940683, 0.93712442, 0.93529076, 0.9247443 , 0.01947328],
[0.92550693, 0.93064972, 0.92791493, 0.9339358 , 0.01954719],
[0.93524016, 0.94894575, 0.95017564, 0.95130949, 0.01227612]])
```

Once again, we have to prepare the test data by dividing it into chunks of 60 days, like the training data. To understand what's happening here, refer back to the explanation given when preparing the training data

```
X_test = []
y_test = []

for i in range(60, inputs.shape[0]):
    X_test.append(inputs[i-60:i])
    y_test.append(inputs[i, 0])

X_test, y_test = np.array(X_test), np.array(y_test)
X_test.shape, y_test.shape
```

Output:

The last 2 lines of the above code give us the shape of our x_test and y_test data.

```
((192, 60, 5), (192,))
```

Predicting the Opening Price

We will use `predict()` to predict the opening price for the x_test data.

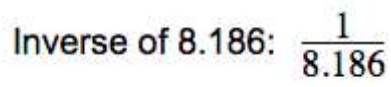
```
y_pred = regressor.predict(X_test)
```

If you run the above code, you may have noticed that the predicted value was way off as it was too small. If you recall, we had scaled all the values down to be in the range of 0 to 1. Now, we have to scale them back up to their original price.

```
array([8.18605127e-04, 8.17521128e-04, 8.32487534e-04, 8.20673293e-04,
       1.21162775e-08])
```

As you can see above, 8.186 is the first value in the list which corresponds to the scale factor applied to the first column (opening price). As our predicted value will be only from that column, we will disregard the rest of the scale factors.

This means we multiply `y_pred` and `y_test` , with the **inverse of the scale factor,** in our case, inverse of 8.186, to get all the values back to their original value.



The inverse is our scale factor

```
scale = 1/18605127e-04
scale
```

Output:

```
1221.5901990069017
```

We will store values into the y_test value (actual value), and y_pred value (predicted value), using the scale factor.

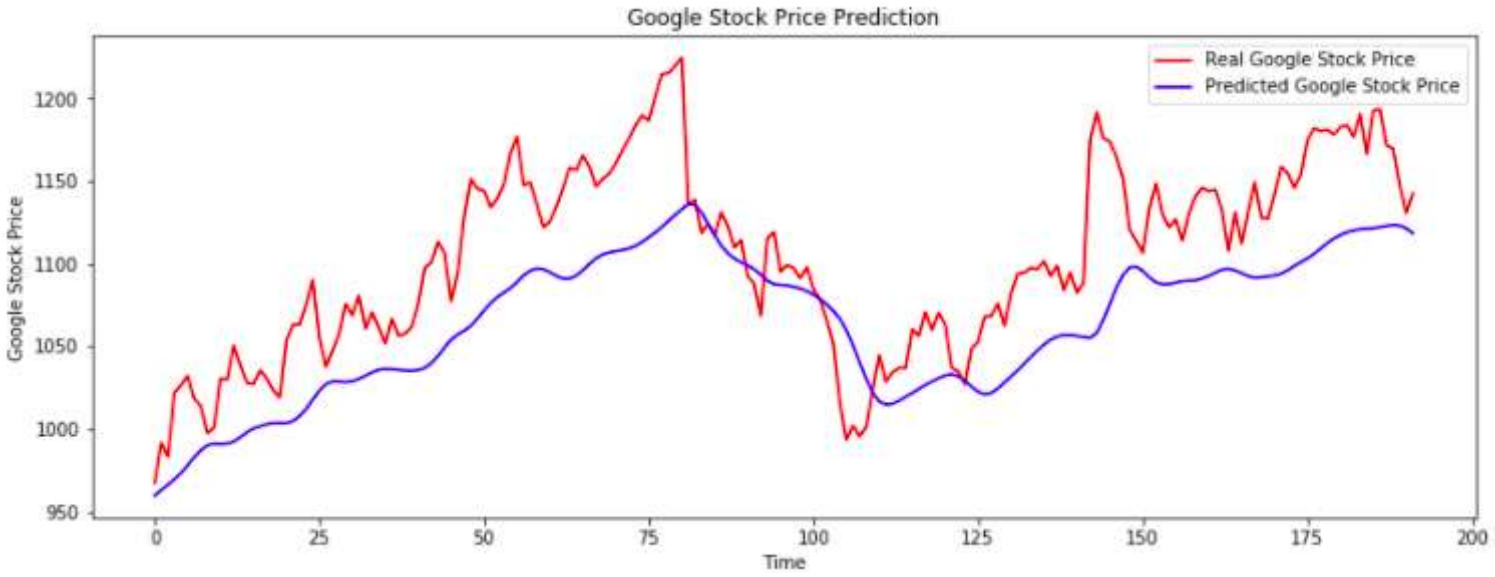
```
y_pred = y_pred*scale
y_test = y_test*scale
```

Data Visualization

Pyplot is great for visualizing the data. It will be much easier to compare the predicted values to the actual values using a graph, especially in the case of stock prices.

```
plt.plot(y_pred, color = 'blue', label = 'Predicted Google Stock Price')
plt.title('Google Stock Price Prediction')
plt.xlabel('Time')
plt.ylabel('Google Stock Price')
plt.legend()
plt.show()
```

Output:



As you can see from the graph, we have a decent accuracy for the opening prices! The difference between both values are not completely off. You can also see how between 75–125, the stock prices are very very similar!

Hopefully, after doing some tweaking to the model, which can be done by feeding the LSTM neural network with more data or feature prediction modelling to handle missing values of the dataset, we are able to make the accuracy even greater.

If the accuracy of the model is increased, this means you'll be able to uncover the behavior of the future market, helping you to know exactly **when** and **which** stocks to invest in, so that you're on your way to becoming a *millionaire*!

. . .

I hope you were able to gain more insight on how LSTM networks work, and how they can be

Connect with me:

🔗 Linkedin

💻 Github

🗓️ Subscribe to my monthly newsletter