

Deep Learning for Predicting Stock Prices



Aditya Gogoi

[Follow](#)

Feb 4 · 9 min read ★

LSTM 4 units, one layer.

Use MinMaxScaler - Normalization.

Predict 20 days or full month of January.

Use 5 years historical data.



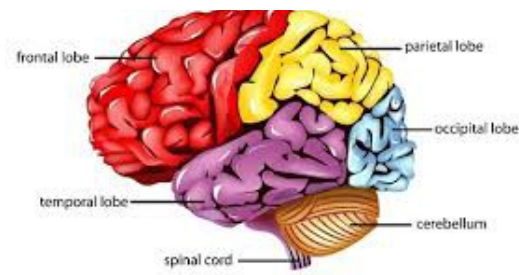
This project involves the use of Recurrent Neural Networks in the form of Long Short-Term Memory (LSTM) to predict the stock prices of Google.

Note — This project is a part of the Deep Learning A-Z course that I have been pursuing, along with some extra information that I gathered from different sources to make the topic more understandable. Please check out the course in Udemy for more insight on different deep learning algorithms.

About RNN

Recurrent Neural Network (RNN) was designed, like any other Neural Network, to function as a specific part of the human brain. When looking at our brain's cerebrum, we can divide it into the Temporal, Parietal, Occipital, and Frontal lobe.

Parts of the Human Brain



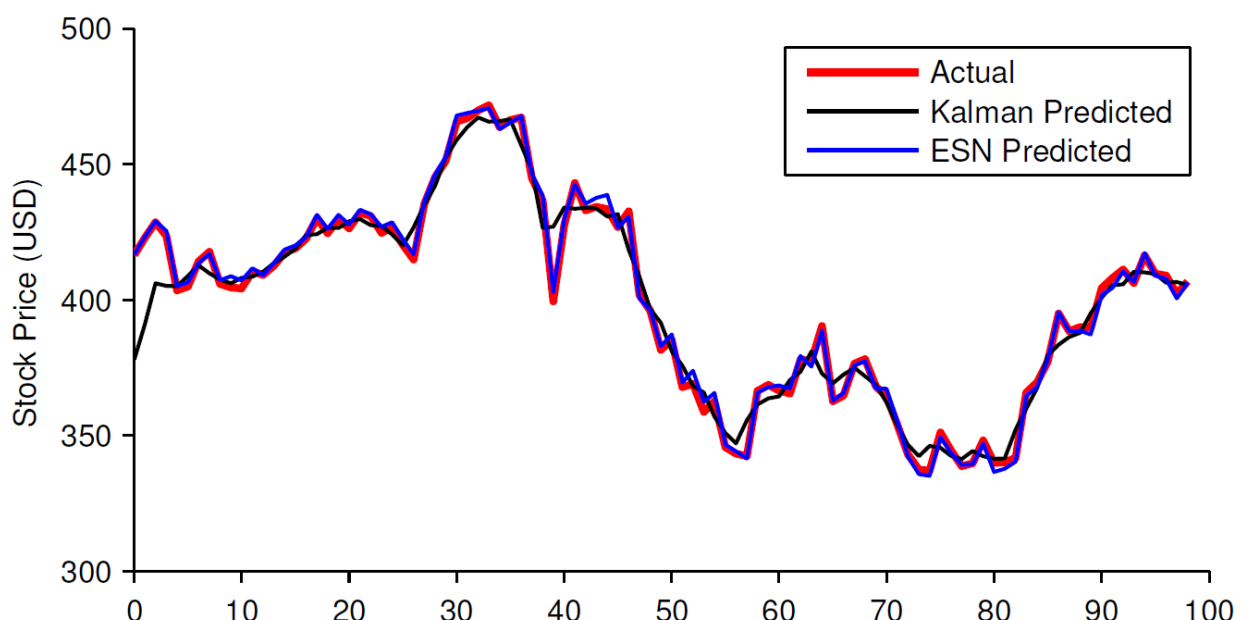
Out of these, the **Frontal lobe** is the part which deals with **short-term memory** and remembers what happened in the **immediate present** and use it for decision making in the near future. It is this Frontal lobe that the RNN tries to replicate.

About LSTM

The Long Short-Term Memory (LSTM) is a variation of the RNN which solves the Vanishing Gradient Problem, which leads to the decrease in the gradient for each level in a regular RNN because of the Recurring Weight being close to 0. This leads to a difficulty in the manipulation of weights for a layer farther away from the “present” layer and thus makes predictions inaccurate. We will cover the details of this phenomenon in another blog post.

About the Dataset

We are using Google’s Stock price from 5 years till now from a financial website (Yahoo Finance). The idea of this project was based on a project by students from Stanford (Financial Market Time Series Prediction with Recurrent Neural Networks — Bernal, Fok, Pidaparthi). The team used an Echo State Network instead of an LSTM. We will use their findings as a comparison to how our LSTM performs. The team trained their model from late 2004 to early 2009 with data from Yahoo Finance. They created a visualization comparing their predictions with the actual data shown below.



Days After Training

Stock Prediction from the RNN Research Paper

We will also train our LSTM on **5 years** of data. We can see that their predictions are quite close to the actual Stock Price. We can try to get the same accuracy from our model as well.

We assume that the present day is **January 01, 2017**. We will then get the Google Stock price for the previous 5 years. Once we train our LSTM, we will try to **predict the stock price for the month of January 2017**.

Data Preprocessing

We have a training set of 5 years of Google Stock price. The test set contains the stock price for January 2017. We will first import it. As we can see in the data-frame, the dates range from 2012 to 2016.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

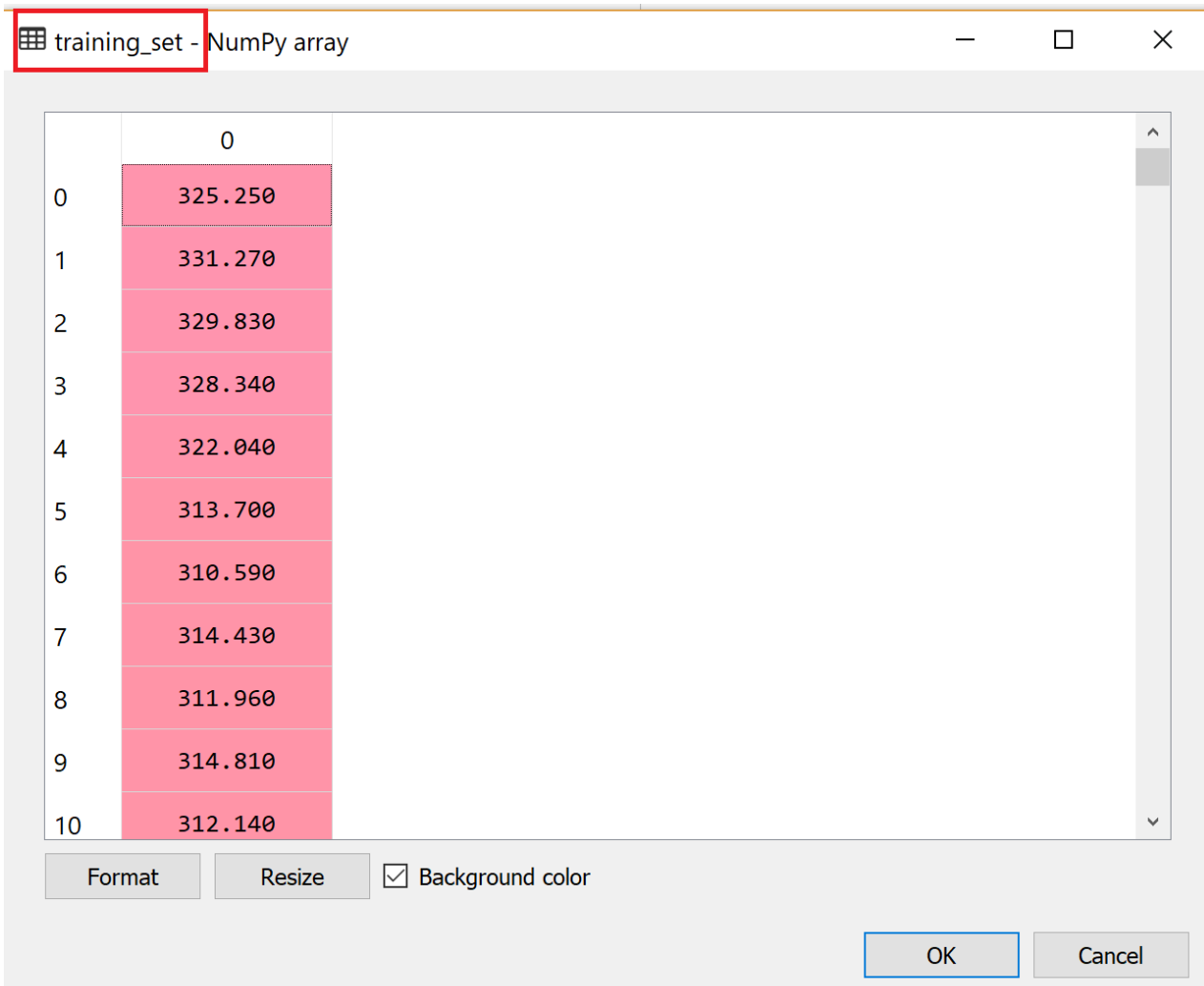
```
training_set = pd.read_csv('Google_Stock_Price_Train.csv')
```

training_set - DataFrame						
Index	Date	Open	High	Low	Close	Volume
0	1/3/2012	325	333	325	663.59	7,380,500
1	1/4/2012	331	334	329	666.45	5,749,400
2	1/5/2012	330	331	327	657.21	6,590,300
3	1/6/2012	328	329	324	648.24	5,405,900
4	1/9/2012	322	322	309	620.76	11,688,800
5	1/10/2012	314	316	307	621.43	8,824,000
6	1/11/2012	311	314	309	624.25	4,817,800
7	1/12/2012	314	315	312	627.92	3,764,400
8	1/13/2012	312	312	309	623.28	4,631,800
9	1/17/2012	315	315	312	626.86	3,832,800
10	1/18/2012	312	316	310	631.18	5,544,000
11	1/19/2012	319	319	315	637.82	12,657,800
12	1/20/2012	294	294	290	584.39	21,231,800

We can see that there are 3 types of stocks — **Open**, **High**, **Low** and **Close**. There is also a **Volume** column which contains the Volume of stocks for Google. We will focus on the Open stock price and will predict this price for January 2017.

The input for RNN will **NOT** be Date and Open Stock Price, it will just be the Open Stock Price for different time frames. We will get this by using just the Open Stock Price from our Dataframe using '**iloc**'.

```
training_set = training_set.iloc[:,1:2].values
```



	0
0	325.250
1	331.270
2	329.830
3	328.340
4	322.040
5	313.700
6	310.590
7	314.430
8	311.960
9	314.810
10	312.140

Format Resize ☒ Background color

OK Cancel

Getting just Open stock Price for the training set

For **feature scaling**, we have 2 options — **Standardisation** and **Normalisation**. Check this [StackExchange discussion](#) to learn the difference between Standardisation and Normalisation. Since LSTMs use many **sigmoid** functions, which work in 0s and 1s, it

makes sense to use **Normalisation**, that converts the data between 0 and 1. But you can check the results of both the methods and choose for yourself.

```
from sklearn.preprocessing import MinMaxScaler
```

```
sc = MinMaxScaler()  
training_set = sc.fit_transform(training_set)
```



	0
0	0.086
1	0.097
2	0.094
3	0.092
4	0.080
5	0.064
6	0.059
7	0.066
8	0.061
9	0.066
10	0.061
11	0.075
12	0.028

We will next try to determine what the **input (X_train)** and **output (y_train)** will be. The **input** will be the value that changes with time i.e. the **current open stock price** (time t). The **output** would obviously be the **future value** of the same i.e. the near future open stock price (time $t+1$). The trick behind choosing the ranges would be that the prediction would be a day after the current value. The training set contains 1258 values. The input should be therefore restricted to 1257. The output, on the other hand, cannot contain the 0th day's prediction, so it will start from 1 and end at 1258.

```
X_train = training_set[0:1257]
y_train = training_set[1:1258]
```



	0
0	0.086
1	0.097
2	0.094
3	0.092
4	0.080
5	0.064
6	0.059
7	0.066

Format Resize ☒ Background color

OK

Cancel

Input from training set

y_train - NumPy array			
	0		
0	0.097		
1	0.094		
2	0.092		
3	0.080		
4	0.064		
5	0.059		
6	0.066		
7	0.061		
Format		Resize	<input checked="" type="checkbox"/> Background color
		OK	Cancel

Output from training set

The use of LSTM (and RNN) involves the **prediction** of a particular value **along time**. Our input is currently 2-dimensional — we have 1257 rows and 1 column. We need to **add another dimension** to the input to account for time. This process is called **reshaping**. This format of the input is required by Keras and the **arguments** have to be in the order of **batch_size** (number of rows), **timesteps** (the number of time intervals or days between any 2 rows, in this case, it will be 1) and **input_dim** (number of columns). These 3 arguments are encapsulated together and come after the original data as the argument of Numpy's **reshape** function.

```
X_train = np.reshape(X_train, (1257, 1, 1))
```

Variable explorer			
Name	Type	Size	Value
X_train	float64	(1257, 1, 1)	array([[[0.08581368]],
training_set	float64	(1258, 1)	array([[0.08581368],
y_train	float64	(1257, 1)	array([[0.09701243],

Adding "time" dimension to input

Building the RNN with LSTM

We will use the Keras library along with Tensorflow to build this Deep Learning framework.

First, we will import 3 classes. The **Sequential** class that will **initialize our RNN**. The **Dense** class will create the **output layer** of our RNN. And finally, the **LSTM** class which will make our RNN have "**Long Memory**".

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
```

We are predicting a **continuous variable** and are hence using a **regression model** instead of a classification model and call our object '**regressor**'. To this regressor object, we will add the LSTM layer, which in itself will take the **input layer as input**. The arguments we add to the LSTM layer are:

1. **units** — number of **memory units**,

2. **activation function** — can be **tanh** or **sigmoid**. Other arguments will be **default**.

But there will be an additional argument — the **input shape** argument to specify the **format** of our **input layer**. This argument would be **none** and **1** — **none** to specify that model can expect **any time step** and **1** because we have just **1 column of input**. The **optimal number of memory units that we can use is 4**, the **activation function** is **sigmoid**, and the **input_shape** would be **(None, 1)**.

```
regressor = Sequential()
regressor.add(LSTM(units = 4, activation = 'sigmoid', input_shape =
(None, 1)))
```

The next layer that we will add is the **Output layer**. We will use the **Dense** class, with the **argument** being **units**, everything else being **default**. The **units** are the **number of neurons** that should be present in the output layer, which is **dependent** on the **dimensions of the output**. So, our units argument for Dense class will have value **1**.

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

To **compile** all the layers into a **single system**, we will use the compile function along with its arguments. The **Optimizer** can be **RMSprop** or **Adam**. Both the optimizers give similar results but **RMS is memory heavy**, so I went forward with Adam. But usually, RMSprop is recommended in Keras documentation. The **Loss** argument decides the **manipulation of weights**, so for this, we should be using **Mean Squared Error** for the **continuous variable**. For the **test set**, we might use **Root Mean Square Error** in its place. Other arguments will be **default**.

```
regressor.compile(optimizer = 'adam', loss = 'mean_squared_error')
```

Now we will **fit** this regressor to the **training dataset**. We will use the **fit** method for this. The important arguments include the **input**, **output**, **batch_size**, and **epochs**. We will keep the default **batch size of 32** but will change **epochs to 200** for **better convergence**.

```
regressor.fit(X_train, y_train, batch_size = 32, epochs = 200)
```

```
Console 1/A
In [11]: regressor.compile(optimizer = 'adam', loss = 'mean_squared_error')

In [12]: regressor.fit(X_train, y_train, batch_size = 32, epochs = 200)
Epoch 1/200
1257/1257 [=====] - 2s - loss: 0.1816
Epoch 2/200
1257/1257 [=====] - 0s - loss: 0.1369
Epoch 3/200
1257/1257 [=====] - 0s - loss: 0.1112
Epoch 4/200
1257/1257 [=====] - 0s - loss: 0.0976
Epoch 5/200
1257/1257 [=====] - 0s - loss: 0.0912
Epoch 6/200
1257/1257 [=====] - 0s - loss: 0.0882
Epoch 7/200
1257/1257 [=====] - 0s - loss: 0.0866
Epoch 8/200
1257/1257 [=====] - 0s - loss: 0.0853
Epoch 9/200
1056/1257 [=====>.....] - ETA: 0s - loss: 0.0843
```

Fitting the RNN on the training dataset

With each **fitting epoch passage**, we see that the **loss keeps on decreasing**. But we will get **accurate results** only if we have the **same loss** in the **test set**.

```
IPython console
Console 1/A
Epoch 192/200
1257/1257 [=====] - 0s - loss: 2.5013e-04
Epoch 193/200
1257/1257 [=====] - 0s - loss: 2.5000e-04
Epoch 194/200
1257/1257 [=====] - 0s - loss: 2.4818e-04
Epoch 195/200
1257/1257 [=====] - 0s - loss: 2.4998e-04
Epoch 196/200
1257/1257 [=====] - 0s - loss: 2.4741e-04
Epoch 197/200
1257/1257 [=====] - 0s - loss: 2.4969e-04
Epoch 198/200
1257/1257 [=====] - 0s - loss: 2.4788e-04
Epoch 199/200
1257/1257 [=====] - 0s - loss: 2.4898e-04
Epoch 200/200
1257/1257 [=====] - 0s - loss: 2.5114e-04
Out[12]: <keras.callbacks.History at 0x2385dd4aef0>

In [13]:
```

Decreasing loss with each epoch

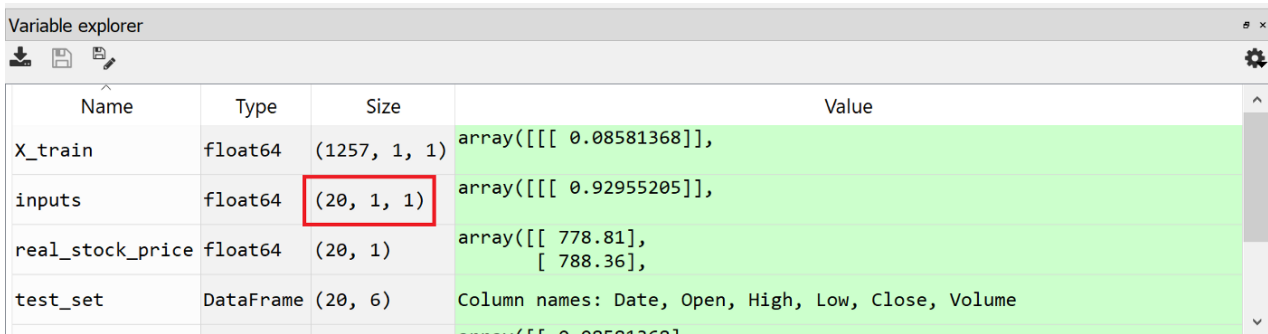
Making Predictions and Visualizing the Result

The methods of getting and transforming the **Test Dataset** are the same as for the Training Dataset. We will just **rename** it to **real_stock_price** so that we can distinguish between prediction and actual values.

The **model** that we have created is on **scaled** values. When used as it is, it will give incorrect predictions. So we will **convert** the input using the same “sc” **scaling object**

used for scaling the training data. We will also have to **reshape** the data according to the format expected by the predict method in a 3d format.

```
test_set = pd.read_csv('Google_Stock_Price_Test.csv')
real_stock_price = test_set.iloc[:,1:2].values
inputs = real_stock_price
inputs = sc.transform(inputs)
inputs = np.reshape(inputs, (20, 1, 1))
```



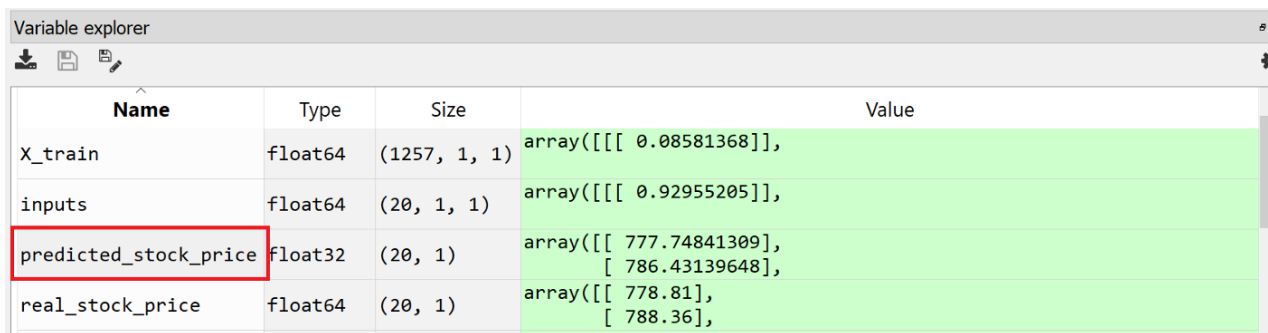
Name	Type	Size	Value
X_train	float64	(1257, 1, 1)	array([[[0.08581368]],
inputs	float64	(20, 1, 1)	array([[[0.92955205]],
real_stock_price	float64	(20, 1)	array([[778.81], [788.36],
test_set	DataFrame	(20, 6)	Column names: Date, Open, High, Low, Close, Volume

Transforming of Test dataset

Next, we use our model to **make predictions** on the **test** dataset. But we should keep in mind that every prediction is for the next day and not the present. We will use the **regressor model** to make predictions on the input and store it in the **predicted_stock_price**. The **argument** would obviously be the **input**. We now have the **predicted stock price** for the month of **January 2017**.

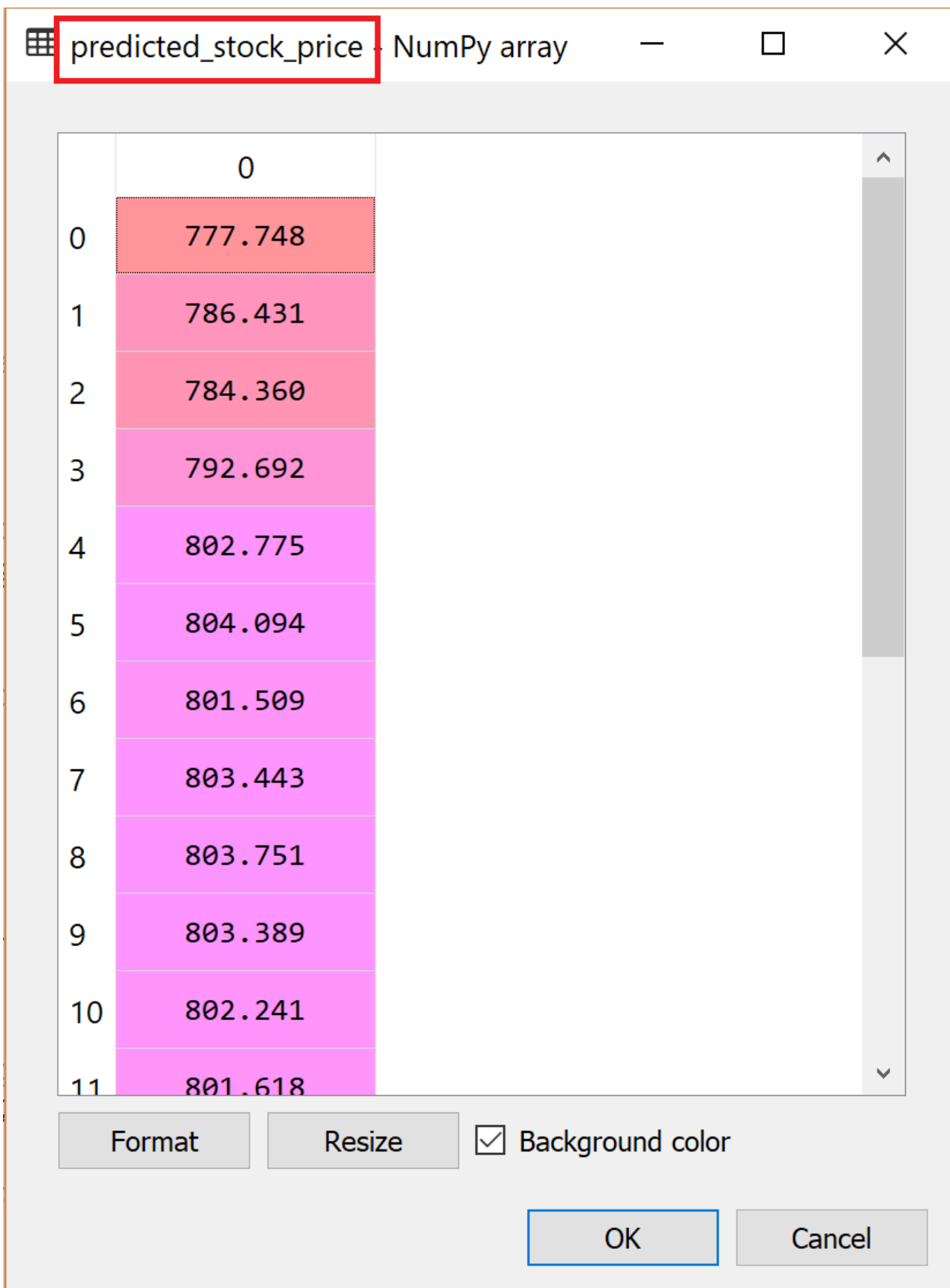
But this **output will be scaled**. We will have to use the **inverse transform** method of the same “sc” object we had used to scale the data to get the proper predicted values. This is the **final prediction**.

```
predicted_stock_price = regressor.predict(inputs)
predicted_stock_price = sc.inverse_transform(predicted_stock_price)
```



Name	Type	Size	Value
X_train	float64	(1257, 1, 1)	array([[[0.08581368]],
inputs	float64	(20, 1, 1)	array([[[0.92955205]],
predicted_stock_price	float32	(20, 1)	array([[777.74841309], [786.43139648],
real_stock_price	float64	(20, 1)	array([[778.81], [788.36],

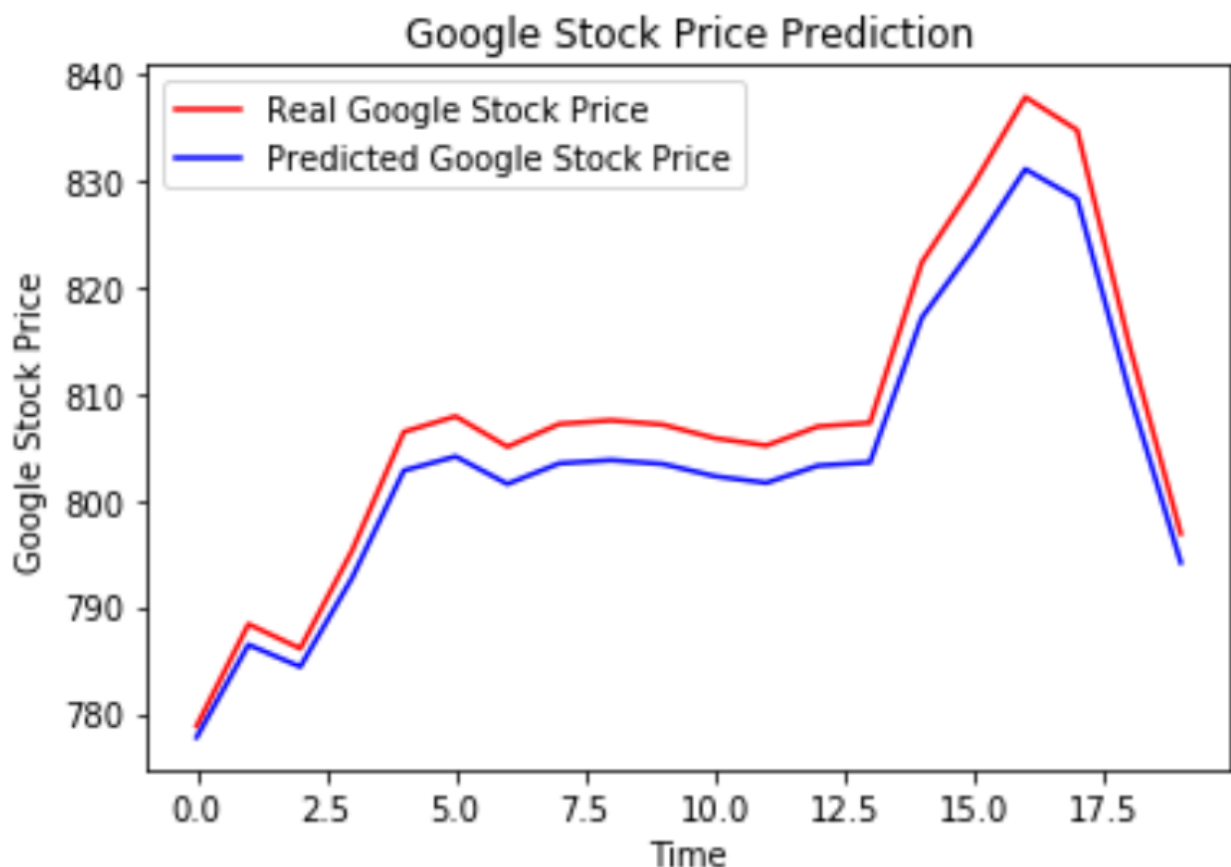
Prediction variable



Scaled back values for the predictions

Now we **visualize** our predictions with the actual stock prices of Google. For this, we use the **pyplot** module. We have some arguments with pyplot, like the use of color. For **real stock prices**, we will use **red**. We will also include a label mentioning the real stock price. We will keep **blue** the color for the **Predicted Stock price** and change the label as well. We will also add the axis labels and title and display it.

```
plt.plot(real_stock_price, color = 'red', label = 'Real Google Stock Price')
plt.plot(predicted_stock_price, 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()
```



Visualization between Predicted and Real stock prices

What's important to note is that this is a 1 time-step prediction i.e. **input** is of time t and **prediction** is of time $t+1$. We should note that we were able to make these predictions for 20 days only because we had the stock price for 20 days. This would **not** have been possible if we had the stock price for just 1 day. It would be wonderful to make predictions for a long future, but we would hardly get such amazing predictions. In finance, there a phenomenon called Brownian Motion, which makes **future values** of stock prices **independent of the past**, so it would be impossible to make long term predictions for a stock price.

Summary

Although the LSTM we have designed predicts quite accurately the stock prices of Google, the reason it is **so accurate** is that it is learning at **time-step of 1**. This leads to a

reset of the **hidden layer**, and this process goes on and the model is **not learning anything useful**. This output is not relevant because of this 1 time-step learning.

To make our model **useful** in the **real world**, we need to **increase** the **time-step**.

Clarification

The project discussed at the beginning of the post is indeed from Stanford, but it is created by undergraduate students of the CS229 course. The students of this course submitted a final project for evaluation; it was just put online but it is not published anywhere. There is a big difference between reading a Stanford paper published online and reading an assignment written by a group of undergrads.

Additional Materials

For sample code and additional materials related to this project, please visit my Github Repo for this project.

[Deep Learning](#)

[Recurrent Neural Network](#)

[Stock Market Prediction](#)

[TensorFlow](#)

[Keras](#)

[About](#)

[Help](#)

[Legal](#)