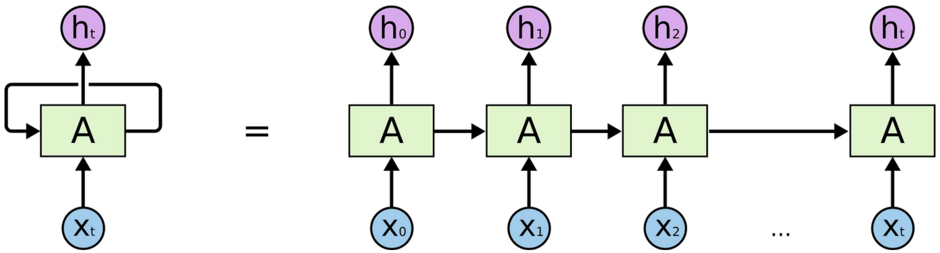
Things to remember

- If you are not familiar with keras or neural networks, refer to this kernel/tutorial of mine: https://www.kaggle.com/thebrownviking20/intro-to-keras-with-breast-cancer-data-ann
- Your doubts and curiousity about time series can be taken care of here: https://www.kaggle.com/thebrownviking20/everything-you-can-do-with-a-time-series
- Don't let the explanations intimidate you. It's simpler than you think.
- Eventually, I will add more applications of LSTMs. So stay tuned for more!
- The code is inspired from Kirill Eremenko's Deep Learning Course: https://www.udemy.com/deeplearning/

Recurrent Neural Networks

In a recurrent neural network we store the output activations from one or more of the layers of the network. Often these are hidden later activations. Then, the next time we feed an input example to the network, we include the previously-stored outputs as additional inputs. You can think of the additional inputs as being concatenated to the end of the "normal" inputs to the previous layer. For example, if a hidden layer had 10 regular input nodes and 128 hidden nodes in the layer, then it would actually have 138 total inputs (assuming you are feeding the layer's outputs into itself à la Elman) rather than into another layer). Of course, the very first time you try to compute the output of the network you'll need to fill in those extra 128 inputs with 0s or something.

Source: Quora



Source: Medium

Let me give you the best explanation of Recurrent Neural Networks that I found on internet: https://www.youtube.com/watch?v=UNmqTiOnRfg&t=3s

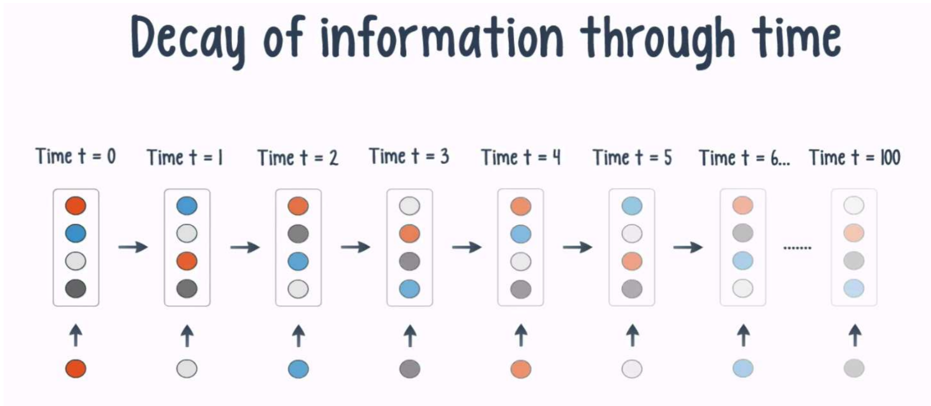
Now, even though RNNs are quite powerful, they suffer from **Vanishing gradient problem ** which hinders them from using long term information, like they are good for storing memory 3-4 instances of past iterations but larger number of instances don't provide good results so we don't just use regular RNNs. Instead, we use a better variation of RNNs: **Long Short Term Networks(LSTM).**

What is Vanishing Gradient problem?

Vanishing gradient problem is a difficulty found in training artificial neural networks with gradient-based learning methods and backpropagation. In such methods, each of the neural network's weights receives an update proportional to the partial derivative of the error function with respect to the current weight in each iteration of training. The problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training. As one example of the problem cause, traditional activation functions such as the hyperbolic tangent function have gradients in the range (0, 1), and backpropagation computes gradients by the chain rule. This has the effect of multiplying n of these small numbers to compute gradients of the "front" layers in an n-layer

network, meaning that the gradient (error signal) decreases exponentially with n while the front layers train very slowly.

Source: Wikipedia



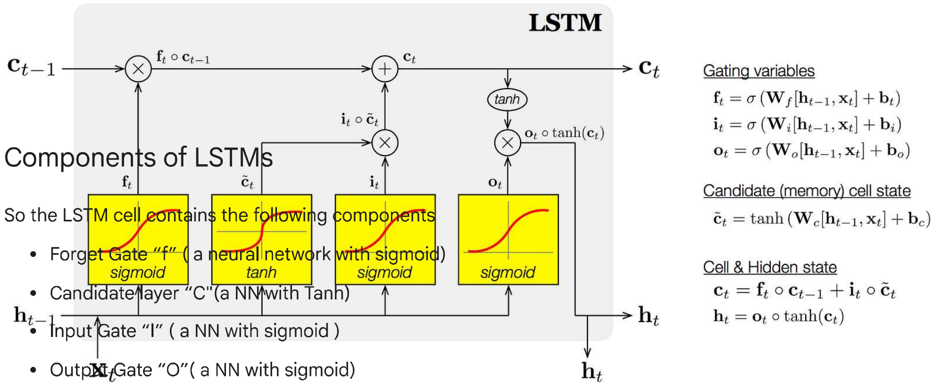
Source: Medium

Long Short Term Memory(LSTM)

Long short-term memory (LSTM) units (or blocks) are a building unit for layers of a recurrent neural network (RNN). A RNN composed of LSTM units is often called an LSTM network. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell is responsible for "remembering" values over arbitrary time intervals; hence the word "memory" in LSTM. Each of the three gates can be thought of as a "conventional" artificial neuron, as in a multi-layer (or feedforward) neural network: that is, they compute an activation (using an activation function) of a weighted sum. Intuitively, they can be thought as regulators of the flow of values that goes through the connections of the LSTM; hence the denotation "gate". There are connections between these gates and the cell.

The expression long short-term refers to the fact that LSTM is a model for the short-term memory which can last for a long period of time. An LSTM is well-suited to classify, process and predict time series given time lags of unknown size and duration between important events. LSTMs were developed to deal with the exploding and vanishing gradient problem when training traditional RNNs.

Source: Wikipedia



Components of LSTMs

So the LSTM cell contains the following components

- Forget Gate "f" (a neural network with sigmoid)
- Candidate layer "C"(a NN with Tanh)
- Input Gate "I" (a NN with sigmoid)
- Output Gate "O"(a NN with sigmoid)
- Hidden state "H" (a vector)
- Memory state "C" (a vector)
- Inputs to the LSTM cell at any step are X_t (current input) , H_{t-1} (previous hidden state) and C_{t-1} (previous memory state).
- Outputs from the LSTM cell are H_t (current hidden state) and C_t (current memory state)

Source: Medium

The best LSTM explanation on internet: https://medium.com/deep-math-machine-learning-ai/chapter-10-1-deepnlp-lstm-long-short-term-memory-networks-with-math-21477f8e4235

Refer above link for deeper insights.

Working of gates in LSTMs

First, LSTM cell takes the previous memory state C_{t-1} and does element wise multiplication with forget gate (f) to decide if present memory state C_t. If forget gate value is 0 then previous memory state is completely forgotten else f forget gate value is 1 then previous memory state is completely passed to the cell (Remember f gate gives values between 0 and 1).

$C_t = C_{t-1} * f_t$

Calculating the new memory state:

$C_t = C_t + (I_t * C`_t)$

Now, we calculate the output:

$H_t = \tanh(C_t)$

And now we get to the code...

I will use LSTMs for predicting the price of stocks of IBM for the year 2017

```
# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM, Dropout, GRU, Bidirectional
from keras.optimizers import SGD
import math
from sklearn.metrics import mean_squared_error
```

```
# Some functions to help out with
def plot_predictions(test,predicted):
```



```
    plt.plot(test, color='red',label='Real IBM Stock Price')
    plt.plot(predicted, color='blue',label='Predicted IBM Stock Price')
    plt.title('IBM Stock Price Prediction')
    plt.xlabel('Time')
    plt.ylabel('IBM Stock Price')
    plt.legend()
    plt.show()

def return_rmse(test,predicted):
    rmse = math.sqrt(mean_squared_error(test, predicted))
    print("The root mean squared error is {}.".format(rmse))
```

```
# First, we get the data
dataset = pd.read_csv('/content/IBM_2006-01-01_to_2018-01-01.csv', index_col='Date', parse_dates=['Date
dataset.head()
```

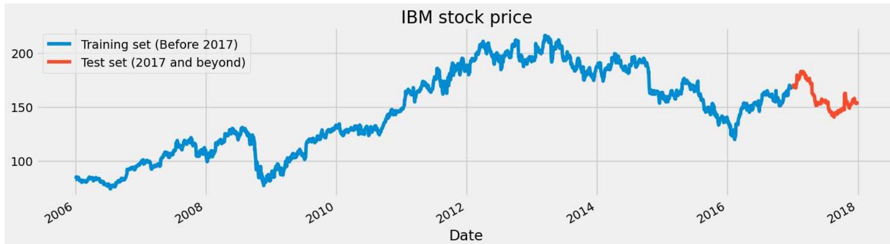
Date	Open	High	Low	Close	Volume	Name
2006-01-03	82.45	82.55	80.81	82.06	11715200	IBM
2006-01-04	82.20	82.50	81.33	81.95	9840600	IBM
2006-01-05	81.40	82.90	81.00	82.50	7213500	IBM
2006-01-06	83.95	85.03	83.41	84.95	8197400	IBM
2006-01-09	84.10	84.25	83.38	83.73	6858200	IBM

Next steps: Generate code with dataset View recommended plots New interactive sheet

```
# Checking for missing values
training_set = dataset[:'2016'].iloc[:,1:2].values
test_set = dataset['2017':].iloc[:,1:2].values
```

Start coding or generate with AI.

```
# We have chosen 'High' attribute for prices. Let's see what it looks like
dataset["High"][:'2016'].plot(figsize=(16,4),legend=True)
dataset["High"]['2017':].plot(figsize=(16,4),legend=True)
plt.legend(['Training set (Before 2017)','Test set (2017 and beyond)'])
plt.title('IBM stock price')
plt.show()
```




```
# Scaling the training set
sc = MinMaxScaler(feature_range=(0,1))
training_set_scaled = sc.fit_transform(training_set)
```

```
# Since LSTMs store long term memory state, we create a data structure with 60 timesteps and 1 output
# So for each element of training set, we have 60 previous training set elements
X_train = []
```

```
y_train = []
for i in range(60,2769):
    X_train.append(training_set_scaled[i-60:i,0])
    y_train.append(training_set_scaled[i,0])
X_train, y_train = np.array(X_train), np.array(y_train)
```

```
# Reshaping X_train for efficient modelling
X_train = np.reshape(X_train, (X_train.shape[0],X_train.shape[1],1))
```

```
# The LSTM architecture
regressor = Sequential()
# First LSTM layer with Dropout regularisation
regressor.add(LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1],1)))
regressor.add(Dropout(0.2))
# Second LSTM layer
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
# Third LSTM layer
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
# Fourth LSTM layer
regressor.add(LSTM(units=50))
regressor.add(Dropout(0.2))
# The output layer
regressor.add(Dense(units=1))

# Compiling the RNN
regressor.compile(optimizer='rmsprop',loss='mean_squared_error')
# Fitting to the training set
regressor.fit(X_train,y_train,epochs=50,batch_size=32)
```



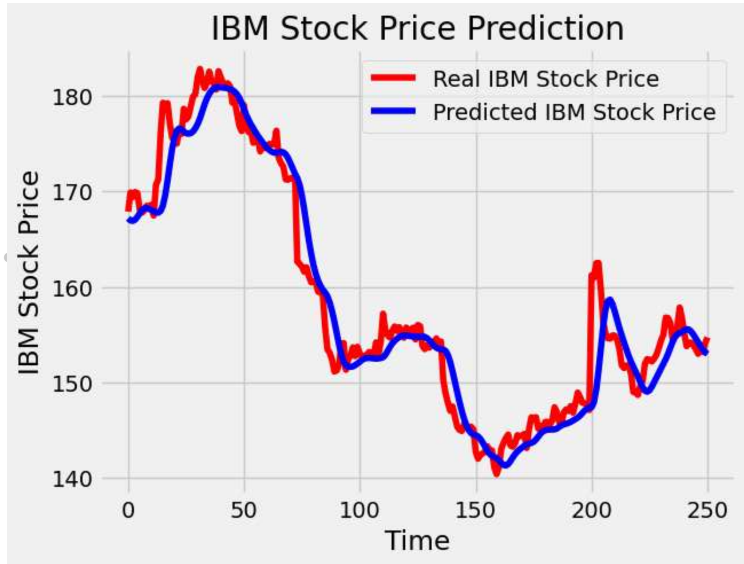
```
85/85 ━━━━━━━━━━━━━━━━━━━━ 10s 118ms/step - loss: 0.0017
Epoch 43/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 9s 104ms/step - loss: 0.0017
Epoch 44/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 10s 118ms/step - loss: 0.0017
Epoch 45/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 10s 117ms/step - loss: 0.0017
Epoch 46/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 10s 118ms/step - loss: 0.0015
Epoch 47/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 9s 108ms/step - loss: 0.0015
Epoch 48/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 10s 104ms/step - loss: 0.0015
Epoch 49/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 11s 118ms/step - loss: 0.0016
Epoch 50/50
85/85 ━━━━━━━━━━━━━━━━━━━━ 10s 119ms/step - loss: 0.0016
<keras.src.callbacks.history.History at 0x7b0a37c79b80>
```

```
# Now to get the test set ready in a similar way as the training set.
# The following has been done so forst 60 entires of test set have 60 previous values which is imposs
# 'High' attribute data for processing
dataset_total = pd.concat((dataset["High"][:'2016'],dataset["High"]['2017':]),axis=0)
inputs = dataset_total[len(dataset_total)-len(test_set) - 60:].values
inputs = inputs.reshape(-1,1)
inputs  = sc.transform(inputs)
```

```
# Preparing X_test and predicting the prices
X_test = []
for i in range(60,311):
    X_test.append(inputs[i-60:i,0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0],X_test.shape[1],1))
predicted_stock_price = regressor.predict(X_test)
predicted_stock_price = sc.inverse_transform(predicted_stock_price)
```

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 1s 108ms/step
```

```
# Visualizing the results for LSTM
plot_predictions(test_set,predicted_stock_price)
```



```
# Evaluating our model
return_rmse(test_set,predicted_stock_price)
```

```
The root mean squared error is 3.01292644862059.
```

Truth be told. That's one awesome score.

LSTM is not the only kind of unit that has taken the world of Deep Learning by a storm. We have **Gated Recurrent Units(GRU)**. It's not known, which is better: GRU or LSTM becuase they have comparable performances. GRUs are easier to train than LSTMs.

Gated Recurrent Units

In simple words, the GRU unit does not have to use a memory unit to control the flow of information like the LSTM unit. It can directly makes use of the all hidden states without any control. GRUs have fewer parameters and thus may train a bit faster or need less data to generalize. But, with large data, the LSTMs with higher expressiveness may lead to better results.

They are almost similar to LSTMs except that they have two gates: reset gate and update gate. Reset gate determines how to combine new input to previous memory and update gate determines how much of the previous state to keep. Update gate in GRU is what input gate and forget gate were in LSTM. We don't have the second non linearity in GRU before calculating the outpu, .neither they have the output gate.

Source: [Quora]

```
# The GRU architecture
regressorGRU = Sequential()
# First GRU layer with Dropout regularisation
regressorGRU.add(GRU(units=50, return_sequences=True, input_shape=(X_train.shape[1],1), activation='t
regressorGRU.add(Dropout(0.2))
# Second GRU layer
regressorGRU.add(GRU(units=50, return_sequences=True, input_shape=(X_train.shape[1],1), activation='t
regressorGRU.add(Dropout(0.2))
# Third GRU layer
regressorGRU.add(GRU(units=50, return_sequences=True, input_shape=(X_train.shape[1],1), activation='t
regressorGRU.add(Dropout(0.2))
# Fourth GRU layer
regressorGRU.add(GRU(units=50, activation='tanh'))
regressorGRU.add(Dropout(0.2))
# The output layer
regressorGRU.add(Dense(units=1))
# Compiling the RNN
regressorGRU.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9, nesterov=False),loss='mean_squar
# Fitting to the training set
regressorGRU.fit(X_train,y_train,epochs=50,batch_size=150)
```



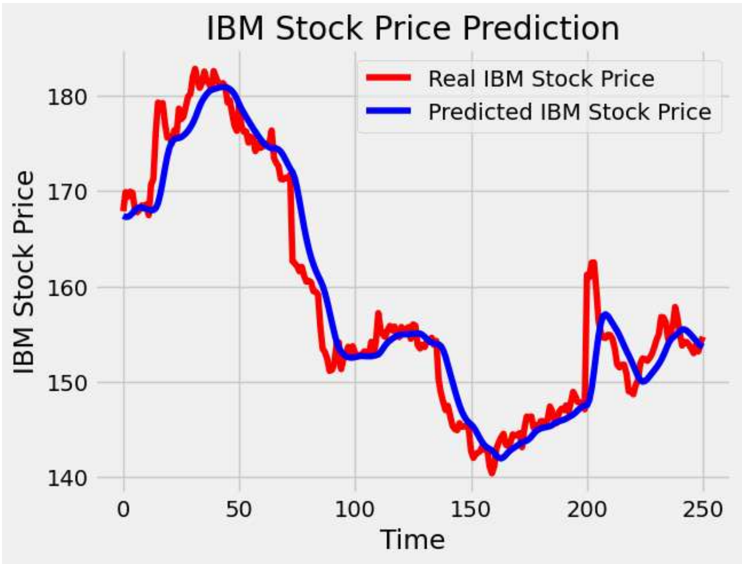
```
19/19 ━━━━━━━━━━━━━━━━━━━━ 8s 274ms/step - loss: 0.0024
Epoch 32/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 10s 275ms/step - loss: 0.0024
Epoch 33/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 10s 275ms/step - loss: 0.0021
Epoch 34/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 10s 272ms/step - loss: 0.0023
Epoch 35/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 11s 311ms/step - loss: 0.0022
Epoch 36/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 11s 345ms/step - loss: 0.0023
Epoch 37/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 9s 301ms/step - loss: 0.0022
Epoch 38/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 10s 272ms/step - loss: 0.0021
Epoch 39/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 10s 275ms/step - loss: 0.0022
Epoch 40/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 7s 344ms/step - loss: 0.0020
Epoch 41/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 5s 276ms/step - loss: 0.0021
Epoch 42/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 11s 280ms/step - loss: 0.0021
Epoch 43/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 5s 274ms/step - loss: 0.0022
Epoch 44/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 6s 331ms/step - loss: 0.0021
Epoch 45/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 10s 349ms/step - loss: 0.0020
Epoch 46/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 9s 275ms/step - loss: 0.0020
Epoch 47/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 10s 274ms/step - loss: 0.0020
Epoch 48/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 10s 278ms/step - loss: 0.0020
Epoch 49/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 10s 276ms/step - loss: 0.0020
Epoch 50/50
19/19 ━━━━━━━━━━━━━━━━━━━━ 11s 301ms/step - loss: 0.0021
<keras.src.callbacks.history.History at 0x7b0a351a9e20>
```

The current version version uses a dense GRU network with 100 units as opposed to the GRU network with 50 units in previous version

```
# Preparing X_test and predicting the prices
X_test = []
for i in range(60,311):
    X_test.append(inputs[i-60:i,0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0],X_test.shape[1],1))
GRU_predicted_stock_price = regressorGRU.predict(X_test)
GRU_predicted_stock_price = sc.inverse_transform(GRU_predicted_stock_price)
```

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 2s 144ms/step
```

```
# Visualizing the results for GRU
plot_predictions(test_set,GRU_predicted_stock_price)
```




```
# Evaluating GRU
return_rmse(test_set,GRU_predicted_stock_price)
```

```
The root mean squared error is 3.164312664864666.
```

Sequence Generation

Here, I will generate a sequence using just initial 60 values instead of using last 60 values for every new prediction. **Due to doubts in various comments about predictions making use of test set values, I have decided to include sequence generation.** The above models make use of test set so it is using last 60 true values for predicting the new value(I will call it a benchmark). This is why the error is so low. Strong models can bring similar results like above models for sequences too but they require more than just data which has previous values. In case of stocks, we need to know the sentiments of the market, the movement of other stocks and a lot more. So, don't expect a remotely accurate plot. The error will be great and the best I can do is generate the trend similar to the test set.

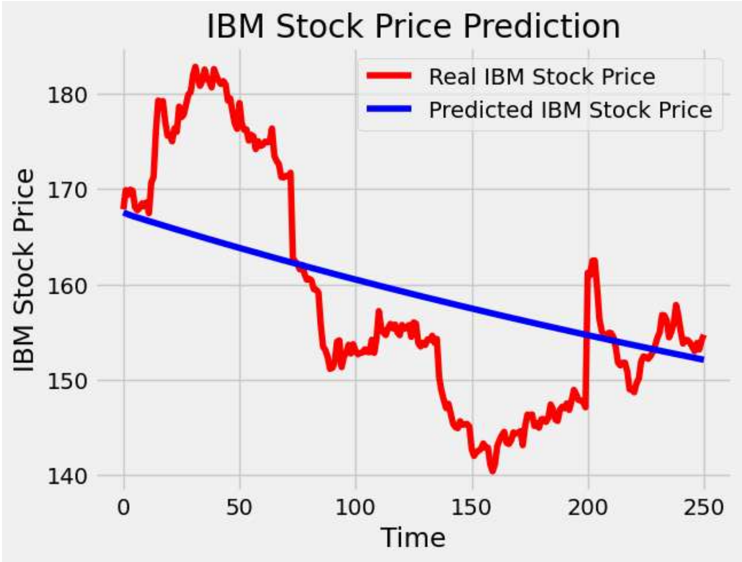
I will use GRU model for predictions. You can try this using LSTMs also. I have modified GRU model above to get the best sequence possible. I have run the model four times and two times I got error of around 8 to 9. The worst case had an error of around 11. Let's see what this iterations.

The GRU model in the previous versions is fine too. Just a little tweaking was required to get good sequences. **The main goal of this kernel is to show how to build RNN models. How you predict data and what kind of data you predict is up to you. I can't give you some 100 lines of code where you put the destination of training and test set and get world-class results. That's something you have to do yourself.**


```
# Preparing sequence data
initial_sequence = X_train[2708,:]
sequence = []
for i in range(251):
    new_prediction = regressorGRU.predict(initial_sequence.reshape(initial_sequence.shape[1],initial_
    initial_sequence = initial_sequence[1:]
    initial_sequence = np.append(initial_sequence,new_prediction,axis=0)
    sequence.append(new_prediction)
sequence = sc.inverse_transform(np.array(sequence).reshape(251,1))
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 48ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 80ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 80ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 82ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 76ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 71ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 69ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 72ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 69ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 73ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 73ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 74ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 77ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 72ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 76ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 84ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 81ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 86ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 52ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 73ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 87ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 79ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 82ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 78ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 73ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 81ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 91ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 75ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 78ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 113ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 95ms/step
```

```
# Visualizing the sequence
plot_predictions(test_set,sequence)
```




```
# Evaluating the sequence
return_rmse(test_set,sequence)
```

```
The root mean squared error is 9.239127953383672.
```

So, GRU works better than LSTM in this case. Bidirectional LSTM is also a good way so make the model stronger. But this may vary for different data sets. **Applying both LSTM and GRU together gave even better results.**

I was going to cover text generation using LSTM but already an excellent kernel by [Shivam Bansal] on the mentioned topic exists. Link for that kernel here: https://www.kaggle.com/shivamb/beginners-guide-to-text-generation-using-lstms

This is certainly not the end. Stay tuned for more stuff!