Recurrent Neural Networks

In a recurrent neural network, we store the output activations from one or more layers, often hidden layer activations. Subsequently, when we present a new input example to the network, these previously-stored outputs are included as additional inputs. These additional inputs can be envisioned as concatenated to the end of the 'normal' inputs for the preceding layer. For instance, if a hidden layer comprises 10 regular input nodes and 128 hidden nodes, it effectively has 138 total inputs (assuming the layer's outputs feed back into itself rather than another layer). Initially, when computing the network's output for the first time, those additional 128 inputs must be initialized, typically with 0s or similar values.

Despite the considerable power of RNNs, they encounter the vanishing gradient problem, restricting their ability to effectively leverage long-term information. While they excel at retaining memory for 3-4 instances of past iterations, larger numbers of instances yield suboptimal results. As a result, traditional RNNs alone are insufficient. Instead, a superior variation of RNNs called Long Short-Term Memory (LSTM) networks is utilized.

Long Short Term Memory(LSTM)

LSTM units indeed form the building blocks for recurrent neural networks (RNNs). When multiple LSTM units are stacked together, the resulting network is often referred to as an LSTM network. Each LSTM unit comprises essential components: a cell responsible for retaining information across various time steps, along with the input, output, and forget gates. These gates act as control mechanisms, regulating the flow of information through the connections within the LSTM.

The term "long short-term" in LSTM signifies its capability to retain short-term memories over prolonged periods. This attribute makes LSTMs particularly adept at tasks involving time series analysis, where there might be unpredictable time lags between significant events. Additionally, LSTMs were specifically designed to combat the challenges of the exploding and vanishing gradient problems encountered during the training of traditional RNNs, making them more effective in learning and retaining long-range dependencies within sequential data.

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)

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).

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

Calculating the new memory state:

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

Now, we calculate the output:

$H_t = \tanh(C_t)$

Use LSTM and GRU for predicting the price of stocks of IBM

Import libraries

In [1]:
```
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
```

```
Using TensorFlow backend.
```

In [2]:
```
# 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))
```

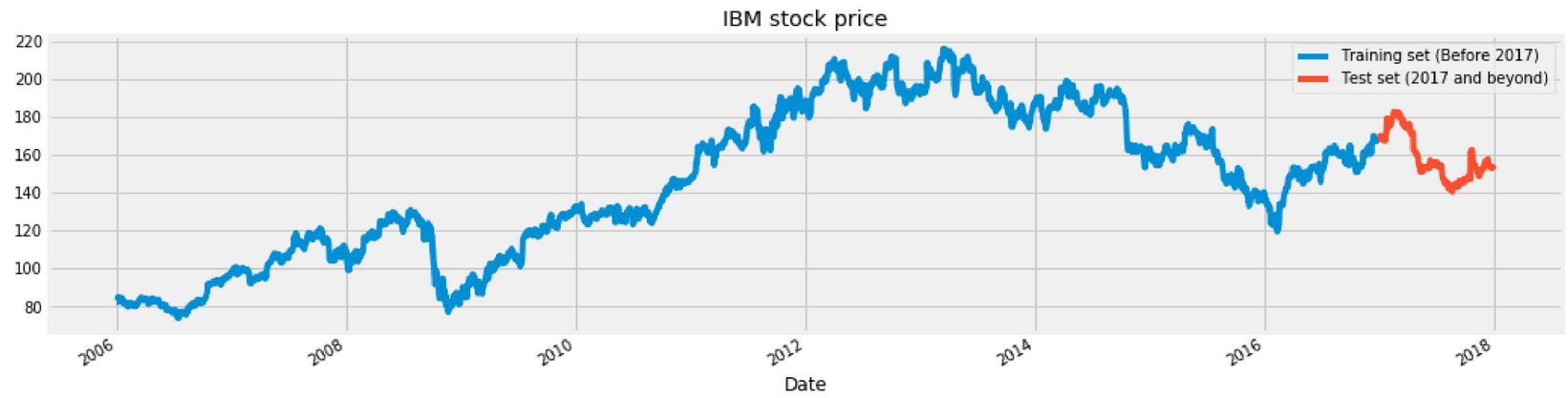
In [3]:
```
# read data
dataset = pd.read_csv('../input/IBM_2006-01-01_to_2018-01-01.csv', index_col='Date', parse_dates=['Date'])
dataset.head()
```

Out[3]:

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

In [4]:
```
# Check for missing values
training_set = dataset[:'2016'].iloc[:,1:2].values
test_set = dataset['2017':].iloc[:,1:2].values
```

In [5]:
```
# 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()
```



In [6]:
```
# Scaling the training set
sc = MinMaxScaler(feature_range=(0,1))
training_set_scaled = sc.fit_transform(training_set)
```

In [7]:
```
# 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)
```

In [8]:
```
# Reshaping X_train for efficient modelling
X_train = np.reshape(X_train, (X_train.shape[0],X_train.shape[1],1))
```

In [9]:
```
# 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)
```

```
Epoch 1/50
2709/2709 [==============================] - 30s 11ms/step - loss: 0.0222
Epoch 2/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0102
Epoch 3/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0086
Epoch 4/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0070
Epoch 5/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0059
Epoch 6/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0056
Epoch 7/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0053
Epoch 8/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0047
Epoch 9/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0042
Epoch 10/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0040
Epoch 11/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0037
Epoch 12/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0033
Epoch 13/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0035
Epoch 14/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0035
Epoch 15/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0030
Epoch 16/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0030
Epoch 17/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0029
Epoch 18/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0027
Epoch 19/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0027
Epoch 20/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0024
Epoch 21/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0025
Epoch 22/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0026
Epoch 23/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0023
Epoch 24/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0024
Epoch 25/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0021
Epoch 26/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0022
Epoch 27/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0021
Epoch 28/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0022
Epoch 29/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0020
Epoch 30/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0021
```

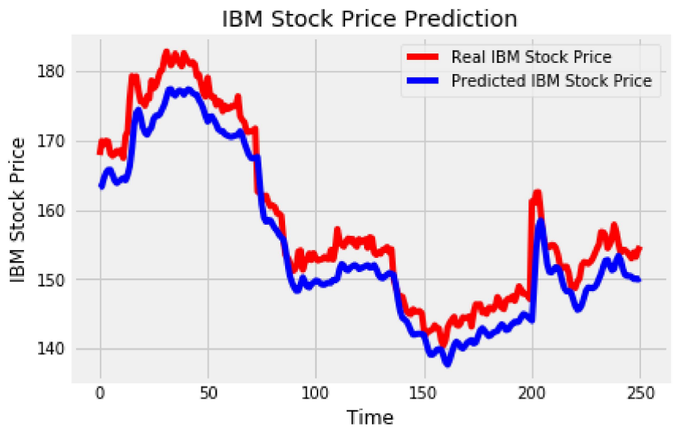
```
Epoch 31/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0020
Epoch 32/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0018
Epoch 33/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0018
Epoch 34/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0018
Epoch 35/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0018
Epoch 36/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0018
Epoch 37/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0017
Epoch 38/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0017
Epoch 39/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0017
Epoch 40/50
2709/2709 [==============================] - 25s 9ms/step - loss: 0.0016
Epoch 41/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0016
Epoch 42/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0017
Epoch 43/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0017
Epoch 44/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0016
Epoch 45/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0015
Epoch 46/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0015
Epoch 47/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0015
Epoch 48/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0015
Epoch 49/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0015
Epoch 50/50
2709/2709 [==============================] - 24s 9ms/step - loss: 0.0014
```

Out[9]:
```
<keras.callbacks.History at 0x7c5f18d0ae10>
```

In [10]:
```
# 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 impossible to get unless we take the whole
# '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)
```

In [11]:
```
# 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)
```

In [12]:
```
# Visualizing the results for LSTM
plot_predictions(test_set,predicted_stock_price)
```



In [13]:
```
# Evaluating our model
return_rmse(test_set,predicted_stock_price)
```

```
The root mean squared error is 4.100627220394395.
```

That's a great score.

Regarding LSTM and its impact on the world of Deep Learning, it's worth noting that it's not the only groundbreaking unit out there. Gated Recurrent Units (GRUs) have also made a significant impact. Determining which is superior, GRU or LSTM, remains uncertain as they exhibit comparable performances. However, GRUs are often considered easier to train compared to LSTMs.

Gated Recurrent Units

In simple terms, the GRU unit doesn't require a separate memory unit to regulate information flow, unlike the LSTM unit. It can directly utilize all hidden states without additional control. GRUs have fewer parameters, potentially resulting in faster training or requiring less data for generalization. However, when dealing with extensive datasets, LSTMs, due to their higher expressiveness, might yield better results.

GRUs closely resemble LSTMs but feature two gates: the reset gate and the update gate. The reset gate determines how to integrate new input with the previous memory, while the update gate decides the extent of the previous state to retain. In GRUs, the update gate performs a role similar to that of the input gate and forget gate in LSTMs. Notably, GRUs lack the second non-linearity before calculating the output, and they also lack the output gate.

In [14]:
```
# 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='tanh'))
regressorGRU.add(Dropout(0.2))
# Second GRU layer
regressorGRU.add(GRU(units=50, return_sequences=True, input_shape=(X_train.shape[1],1), activation='tanh'))
regressorGRU.add(Dropout(0.2))
# Third GRU layer
regressorGRU.add(GRU(units=50, return_sequences=True, input_shape=(X_train.shape[1],1), activation='tanh'))
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(lr=0.01, decay=1e-7, momentum=0.9, nesterov=False),loss='mean_squared_error')
# Fitting to the training set
regressorGRU.fit(X_train,y_train,epochs=50,batch_size=150)
```

```
Epoch 1/50
2709/2709 [==============================] - 6s 2ms/step - loss: 0.1177
Epoch 2/50
2709/2709 [==============================] - 5s 2ms/step - loss: 0.0474
Epoch 3/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0193
Epoch 4/50
2709/2709 [==============================] - 5s 2ms/step - loss: 0.0055
Epoch 5/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0046
Epoch 6/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0039
Epoch 7/50
2709/2709 [==============================] - 5s 2ms/step - loss: 0.0039
Epoch 8/50
2709/2709 [==============================] - 5s 2ms/step - loss: 0.0038
Epoch 9/50
2709/2709 [==============================] - 5s 2ms/step - loss: 0.0036
Epoch 10/50
2709/2709 [==============================] - 5s 2ms/step - loss: 0.0033
Epoch 11/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0033
Epoch 12/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0032
Epoch 13/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0032
Epoch 14/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0030
Epoch 15/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0033
Epoch 16/50
2709/2709 [==============================] - 5s 2ms/step - loss: 0.0027
Epoch 17/50
2709/2709 [==============================] - 5s 2ms/step - loss: 0.0030
Epoch 18/50
2709/2709 [==============================] - 5s 2ms/step - loss: 0.0029
Epoch 19/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0029
Epoch 20/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0028
Epoch 21/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0027
Epoch 22/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0026
Epoch 23/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0027
Epoch 24/50
2709/2709 [==============================] - 5s 2ms/step - loss: 0.0026
Epoch 25/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0027
Epoch 26/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0024
Epoch 27/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0025
Epoch 28/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0026
Epoch 29/50
2709/2709 [==============================] - 5s 2ms/step - loss: 0.0025
Epoch 30/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0024
```

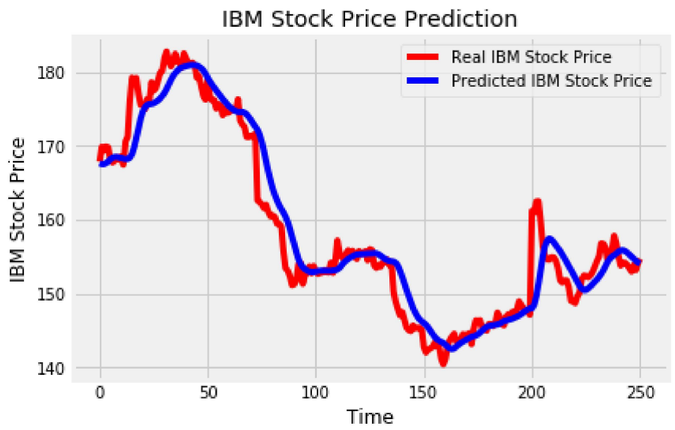
```
Epoch 31/50
2709/2709 [==============================] - 5s 2ms/step - loss: 0.0024
Epoch 32/50
2709/2709 [==============================] - 5s 2ms/step - loss: 0.0025
Epoch 33/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0023
Epoch 34/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0023
Epoch 35/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0024
Epoch 36/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0024
Epoch 37/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0023
Epoch 38/50
2709/2709 [==============================] - 5s 2ms/step - loss: 0.0022
Epoch 39/50
2709/2709 [==============================] - 5s 2ms/step - loss: 0.0022
Epoch 40/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0024
Epoch 41/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0022
Epoch 42/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0022
Epoch 43/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0022
Epoch 44/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0022
Epoch 45/50
2709/2709 [==============================] - 5s 2ms/step - loss: 0.0021
Epoch 46/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0021
Epoch 47/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0022
Epoch 48/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0023
Epoch 49/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0020
Epoch 50/50
2709/2709 [==============================] - 4s 2ms/step - loss: 0.0020
```

Out[14]:
```
<keras.callbacks.History at 0x7c5ade5caeb8>
```

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

In [15]:
```
# 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)
```

In [16]:
```
# Visualizing the results for GRU
plot_predictions(test_set,GRU_predicted_stock_price)
```



In [17]:
```
# Evaluating GRU
return_rmse(test_set,GRU_predicted_stock_price)
```

```
The root mean squared error is 3.2036882102040933.
```

Sequence Generation

Here, I'll generate a sequence using only the initial 60 values instead of relying on the last 60 values for each new prediction. Due to uncertainties expressed in various comments regarding predictions using test set values, I've chosen to include sequence generation. The aforementioned models utilize the test set, hence employing the last 60 true values to predict the new value (which I'll consider as a benchmark). Consequently, the error appears remarkably low. Robust models can yield comparable outcomes to the aforementioned models even for sequences, but they demand more than just historical data.

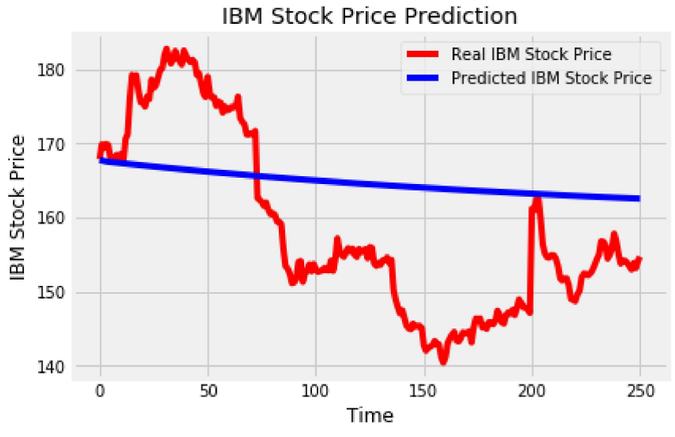
In the case of stocks, accurate predictions necessitate knowledge of market sentiments, movements of other stocks, and various additional factors. Therefore, expecting remotely precise plots is unrealistic. The anticipated error will be substantial, and the best achievable outcome might be generating a trend similar to the test set.

For predictions, I'll employ the GRU model, although you can attempt this using LSTMs as well. I've refined the GRU model above to produce the most optimal sequence. Upon running the model four times, I obtained error values around 8 to 9 in two instances, while the worst case registered an error of around 11. Let's observe the outcomes of these iterations.

The earlier versions of the GRU model are also adequate, requiring only slight adjustments to generate effective sequences. The primary objective of this kernel is to illustrate the construction of RNN models. The manner and nature of data prediction remain at your discretion. I cannot provide a one-size-fits-all code snippet where you input the training and test set destinations and obtain top-tier results. That's a task you'll need to undertake yourself.

In [18]:
```
# 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_sequence.shape[0],1))
    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))
```

In [19]:
```
# Visualizing the sequence
plot_predictions(test_set,sequence)
```



In [20]:
```
# Evaluating the sequence
return_rmse(test_set,sequence)
```

```
The root mean squared error is 12.653342687914995.
```

So, GRU works better than LSTM in this case. Using a bidirectional LSTM is also an effective method to strengthen the model. However, this performance comparison might differ across various datasets. Interestingly, applying both LSTM and GRU together yielded even better results.