**Forecasting stocks**

In this module, you will learn about how a neural network can benefit from predicting a sequence data set like time series or sentence formation. To initiate with we use the Infosys Equities data set from January 1st, 2000 till December 31st, 2009 giving us a total of 10 years of data. The National Stock Exchange market opens only on Weekdays excluding weekends and national holidays, therefore, you can't except data for all 365/366 days a year.

* **Symbol**: INFOSYSTCH throughout the dataset.
* **Series**: EQ (Equity) throughout the dataset.
* **Date**: Date corresponding to the data.
* **Prev Close**: Previous closing price.
* **Open Price**: Corresponding date's open price.
* **High Price**: Corresponding date's high price.
* **Low** **Price**: Corresponding date's low price.
* **Last Price**: Corresponding date's last price.
* **Close Price**: Corresponding date's closing price.
* **Average Price**: Corresponding date's average price.
* **Total Traded Quantity**: Number of quantity traded.
* **Turnover**: Corresponding date's turnover.
* **No.** **of Trades**: Total number of trades.
* **Deliverable** **Qty**: Deliverable stock volume
* **% Dly Qt to Traded Qty**: Ratio of deliverable volume to traded volume.

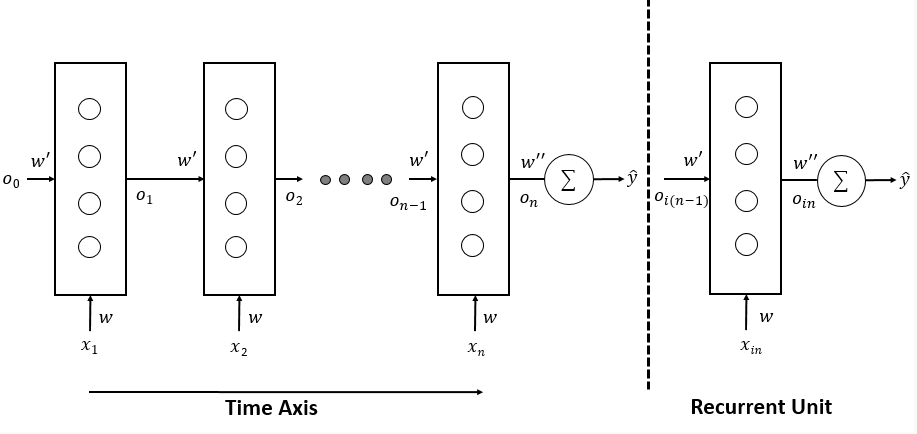
For our time series, we will be considering only two features, **Date** and **Average Price**.

**Understanding RNN**

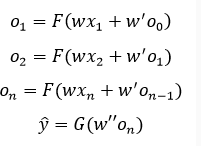
We prefer the Recurrent Neural Network (RNN) in the scenarios where the current state of the input has a dependency on the previous state of the input and so on so forth. For example, if you're to forecast the temperature of today's weather then you're going to decide your results based on historical data. Similarly, forecasting future gold price, predicting the next word while you're writing an e-mail, etc. have dependencies on past data.

We can, of course, go with multi-layer perceptron or convolutional neural network approaches but the former will create an overhead of weights by creating too many unnecessary connections and the latter is more suitable for image data. Therefore, a need for a new network is raised which can work on applications which are solely based upon historical data. The RNN is the building block for the forecasting on such sequence data.

Take a look at the diagram shown below:



This is a basic architecture of an RNN. To understand it, take a look at the L.H.S. diagram of the dashed line. Assuming you have a time series data with N values gathered over N days. You start by feeding the first value (**x1**) from your input layer to the hidden layer having **r** number of nodes. The weights connecting your input layer and the hidden layer is **w**. When you perform the dot product and have the final output (**o1**) ready, you feed the new input **x2** along with the previous output **o1** in the same hidden layer. Do note that only time changes when second input is feed, but hidden layer and weights **w**and **w'** remains the same. This process keeps *repeating*unless you arrive at the final Nth data value. Once you do, then you can apply a final activation function to get the forecasted value y (hat).

If you observe the diagram, the architecture follows a repeating pattern of fetching an input value (**xt**) and previous output (**ot-1**) and resulting in a new output (**ot**). Therefore, if we can just add a random initialized output (**o0**) before the process begin then the whole network can be represented as a recurrent unit as shown in the R.H.S. of the dashed line.

If we assume that the hidden layer employs an activation function **F** and final activation function is **G**, then the forward propagation of given architecture can be inferred from the given equations:

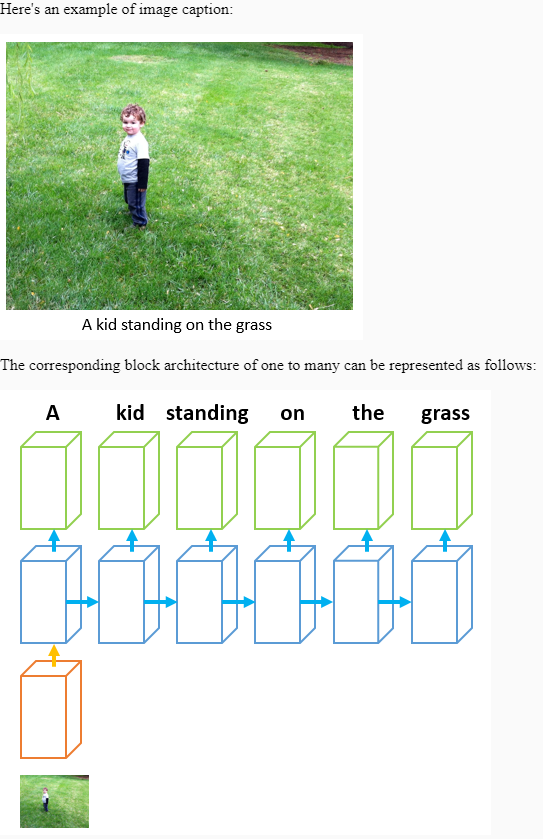
Notice that in each step, the output is dependent upon the current input and last output. For instance, **o2** is dependent upon **x2** and **o1**. This time, backpropagation happens over time on the same hidden layer and for **n** times steps, we will get **n** gradients for **w**as well as **w'.** Later on, we accumulate those gradients and update **w**and**w'**for the next review.

Now, the problem is, if the sequence is large enough, then while computing the backpropagation to arrive at the first value, either the gradient becomes too small or it becomes too large for the optimization algorithm to converge.

To recover the algorithm from the exploding gradient, you can either deploy penalty or follow variants of backpropagation over time like truncated BPTT. To recover from the vanishing gradient you can improve the weight initialization or follow more robust RNN variants like LSTM and GRU which we are going to discuss further in the next resource.

In the last page, we learned about a basic RNN architecture. Now, let us look on a few other RNN architecture variants:

# One to One

This is the architecture which we have seen previously where we have one input and corresponding one output as shown in the block architecture.

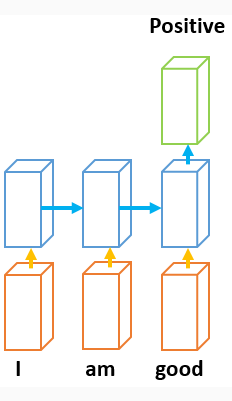
# One to Many

In this architecture, you provide input and expect multiple outputs. Generally, such architecture is seen in the process of image captioning where you provide an input image and expect multiple words (output) corresponding to the image.

Here's an example of image caption:

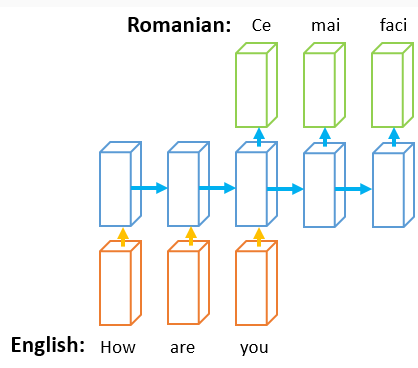
# Many to One

In this architecture, there are multiple inputs mapped to a single output. Consider a case of finding a sentence sentiment (positive/neutral/negative). So, you pass various words (inputs) which need to arrive at a single output (sentiment of the sentence).

For example:

**Positive sentiment**: The food in this restaurant is very delicious.

**Negative sentiment**: It is sad to hear the recent earthquake news.

**Neutral sentiment**: Yes!

# Many to Many

Here, you provide multiple inputs and expect multiple outputs from the network. Consider machine translation where you convert a sentence from language A to language B. So, both input and output have multiple entries. The block diagram for the same is given below:

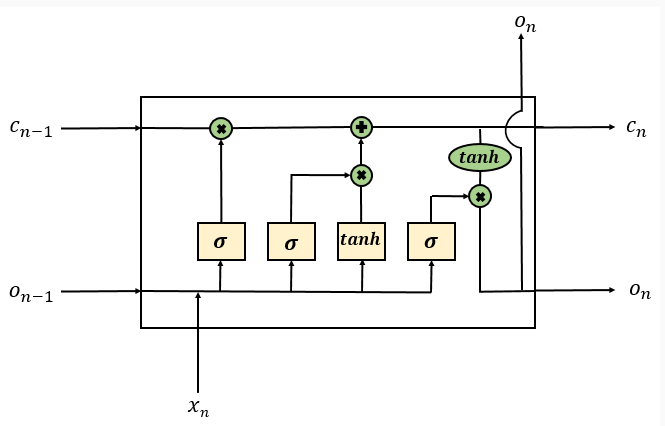
In this resource, you will learn about the few variants of RNN namely Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU) and Bidirectional RNN (BRNN).

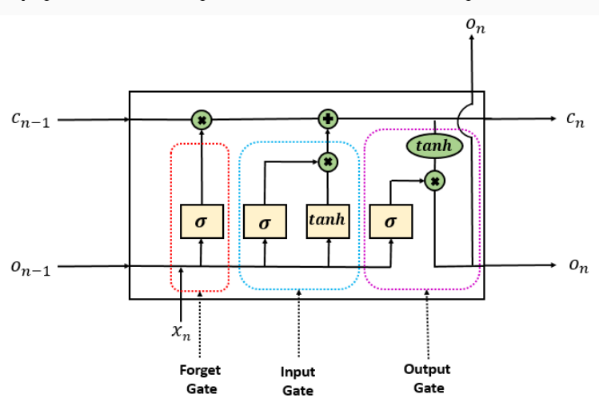
# **RNN Types**

# **Long Short-Term Memory**

Recall from the previous basic RNN structure where, at a given timestamp we have only two inputs (**xn** and **on-1**) resulting only in one output (**on**).

As we know that such structures have problems with gradients which can either vanish or explode. To recover from such a situation, researchers presented a new variant to RNN named as Long Short-Term Memory which has an ability to preserve the effects of old inputs in a long sequence as well as keep the effects of new inputs too. To understand this process visually, observe the given figure:

This is an LSTM cell where the yellow units represent the network layer, green units represent the pointwise operation (either multiplication or addition), two arrow lines meeting together represent the concatenation and an arrow line splitting represents copying the values.

As you can observe at a given timestamp, you got three inputs along with two outputs rather than usual two inputs and one output when compared with basic RNN structure. The above LSTM unit is divided into three separate parts Forget gate, Input gate, and Output gate. Let us discuss the usage of each one of them to cover the functioning of whole LSTM cell. To start with, we provide you with the following diagram which has each gate highlighted for your convenience.

**Forget gate**

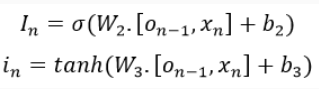
The forget gate (highlighted in the red box) consists of a sigmoid activation layer which helps in taking control of how much previous data needs to be retained and how much of it needs to be expelled. A definite 1 represents keep all the data whereas a definite 0 represents exclude all the data.

Observe that we are not doing any dot product between **xn** and **y (hat) n-1** anywhere. Rather we are performing the concatenation or merging operation. The governed equation of the resulting function.

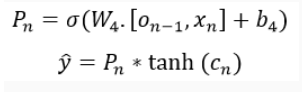
Where **W1** represents the combined weights and **b1** represents bias. Note, we have used comma in the equation which represents concatenation. The output **ft** proceeds with the pointwise multiplication operator as seen in the diagram to control the amount of information to be sent ahead.

**Input gate**

The input gate (highlighted in the blue box) helps to add new information to the final output **on** by the pointwise multiplication of the following two terms:

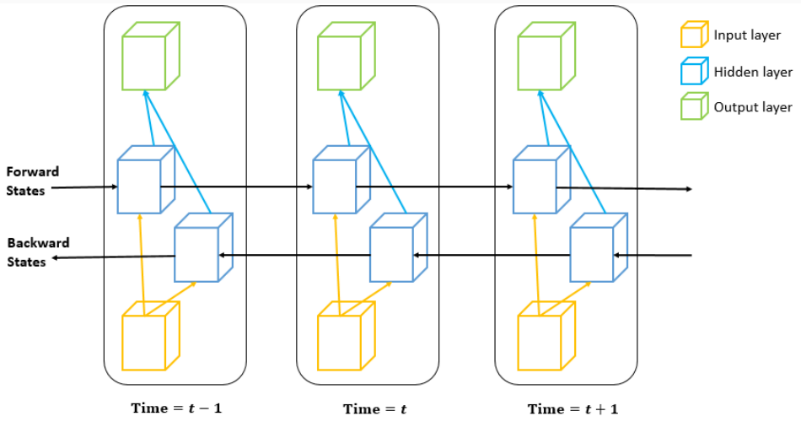
Giving us the final output as the pointwise addition of the forget gate output and the input gate output as shown.

**Output gate**

Finally, to produce the forecasted output **yt**, we need the pointwise multiplication of the following two terms inside the output gate (highlighted in the pink box):

So, this completes a brief introduction of LSTM, their working and how they are efficient than basic RNN architecture. However, there's another variant of LSTM referred to as **Gated Recurrent Unit (GRU)** which has only two gates, reset gate and update gate (a combination of the forget and input gates). This helps in increasing the performance by reducing the number of gradients to be computed.

# **Bidirectional RNN**

In the basic RNN as well as LSTM structures we have observed that apart from a multi-layer perceptron which uses all of the input information, they are constrained to utilize information up to the current timestamp. However, information residing in the future can also play a vital role in predicting the output of a current timestamp layer. For instance, consider a timestamp **t** with the following information: **(t-1, t, t+1):(Kid is hungry)**. Here, you need to know the information present at time **t-1** as well as **t+1** to know that the talk is revolving around a kid who is hungry.

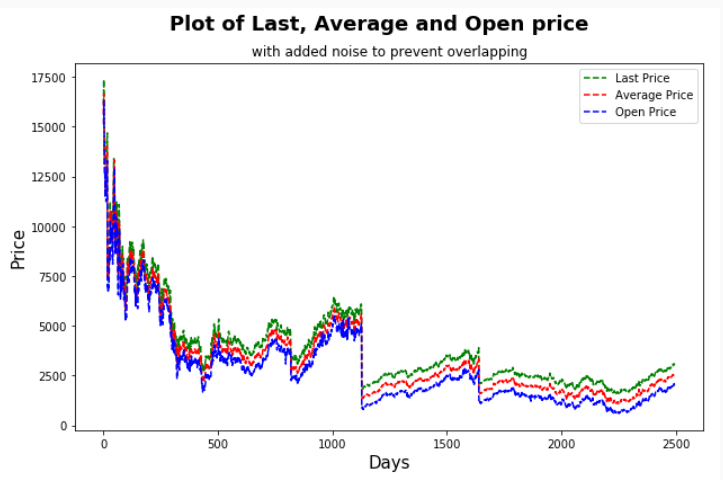
To realize this structure, Bidirectional RNN is introduced which uses information both from the past as well as the future from a specific timeframe. The core idea is to split the hidden neurons into **forward**and **backward states** and keeping both of them independent from one another. So, the output of forward state is not used as an input to the backward state or vice-versa. A general block architecture is presented below unfolded for three-time steps:

So, for the time state t, information from both t-1 and t+1 can be utilized to achieve the output of the current state. Notice that the number of timestamps to be used to result in the current state output is dependent upon the structure chosen. For an LSTM, it can use information from near as well as far timestamps (both past and future). The training happens similar to a basic RNN as both the states (forward and backward) are independent of one another.

Applications of BRNN include next word prediction, handwriting recognition, speech recognition, etc.

In the previous page, you have learned how to perform forecasting based on a single input. In this section, you are required to perform the multivariate time series forecast using three inputs.

Here's the starter code which takes three inputs: Last Price, Average Price, and Open Price and plot them.

1. *# Necessary libraries*
2. import numpy as np
3. import pandas as pd
4. import matplotlib.pyplot as plt
5. import keras
6. from keras.models import Sequential
7. from keras.layers.recurrent import LSTM
8. from keras.layers.core import Dense, Activation, Dropout
9. from sklearn.preprocessing import MinMaxScaler
10. *# Loading data*
11. data = pd.read\_csv('INFY\_2000\_2008.csv')
12. *# Selecting three columns as input*
13. data = data[['Last Price', 'Average Price', 'Open Price']].values
14. plt.figure(figsize=(10, 6))
15. plt.plot(data[:,0]+500, '--g', label='Last Price')
16. plt.plot(data[:,1], '--r', label='Average Price')
17. plt.plot(data[:,2]-500, '--b', label='Open Price')
18. plt.xlabel('Days', fontsize=15)
19. plt.ylabel('Price', fontsize=15)
20. plt.suptitle('Plot of Last, Average and Open price', fontsize=18, weight='bold')
21. plt.title('with added noise to prevent overlapping')
22. plt.legend()
23. plt.show()

You are required to split the dataset in the ratio of 75:25 and train an LSTM model with four units, single LSTM layer till three epochs. Here's the visualized output w.r.t. the Average Price.

Forecasting stocks using LSTM

To start with the implementation of a basic LSTM on time series forecasting, we import the necessary libraries and load the data set:

1. *# Importing libraries*
2. import numpy as np
3. import pandas as pd
4. import matplotlib.pyplot as plt
5. import keras
6. from keras.models import Sequential
7. from keras.layers.recurrent import LSTM
8. from keras.layers.core import Dense, Activation, Dropout
9. from sklearn.preprocessing import MinMaxScaler
10. from sklearn.metrics import mean\_squared\_error
11. *# Loading data*
12. data = pd.read\_csv('INFY\_2000\_2008.csv')
13. data.info()

Next, we use subset only the Date and Average Price attributes out of which we are going to focus upon Average Price and Date is left just for labels.

1. *# Selecting only Date and Average Price columns*
2. data = data[['Date', 'Average Price']]

Let us now proceed with scaling the values and splitting the data set in train and test portions. Remember don't shuffle the dataset while splitting. It should be split following a sequence.

1. *# Scaling the values in the range of 0 to 1*
2. scaler = MinMaxScaler(feature\_range = (0, 1))
3. scaled\_price = scaler.fit\_transform(data.loc[:, 'Average Price'].values.reshape(-1, 1))
4. *# Splitting dataset in the ratio of 75:25 for training and test*
5. train\_size = int(data.shape[0] \* 0.75)
6. train, test = scaled\_price[0:train\_size, :], scaled\_price[train\_size:data.shape[0], :]
7. print("Number of entries (training set, test set): " + str((len(train), len(test))))

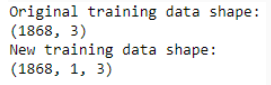
**Number of Entries (training set, test set) : (1872, 624)**

Next, we need to construct a data set from the array of Average Price values along with defining a window size. Window is used to define how many values need to be taken while forecasting the new value. By default, in the function, we set the window size as 1, however, while constructing the data set, we set the window size to 3. You can change it and observe the corresponding effect on the forecasted value.

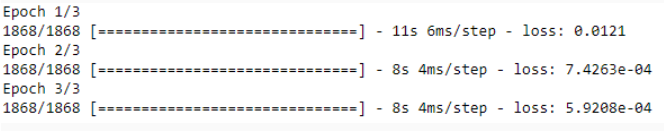
1. def create\_dataset(scaled\_price, window\_size=1):
2. data\_X, data\_Y = [], []
3. for i in range(len(scaled\_price) - window\_size - 1):
4. a = scaled\_price[i:(i + window\_size), 0]
5. data\_X.append(a)
6. data\_Y.append(scaled\_price[i + window\_size, 0])
7. return(np.array(data\_X), np.array(data\_Y))

Next, we call the function and reset the dataset to make it fit for Keras:

1. *# Create test and training sets for one-step-ahead regression.*
2. window\_size = 3
3. train\_X, train\_Y = create\_dataset(train, window\_size)
4. test\_X, test\_Y = create\_dataset(test, window\_size)
5. print("Original training data shape:")
6. print(train\_X.shape)
7. *# Reshape the input data into appropriate form for Keras.*
8. train\_X = np.reshape(train\_X, (train\_X.shape[0], 1, train\_X.shape[1]))
9. test\_X = np.reshape(test\_X, (test\_X.shape[0], 1, test\_X.shape[1]))
10. print("New training data shape:")
11. print(train\_X.shape)

Now, we design our LSTM network with four blocks and just one layer using the MSE as loss function with three epochs.

1. *# Designing the LSTM model*
2. model = Sequential()
3. model.add(LSTM(4, input\_shape = (1, window\_size)))
4. model.add(Dense(1))
5. *# Compiling the model*
6. model.compile(loss = "mean\_squared\_error", optimizer = "adam")
7. *# Training the model*
8. model.fit(train\_X, train\_Y, epochs=3, batch\_size=1)



**Note**: The loss in your case can be different as we have not set the seed while constructing the model.

# Forecasting and visualization

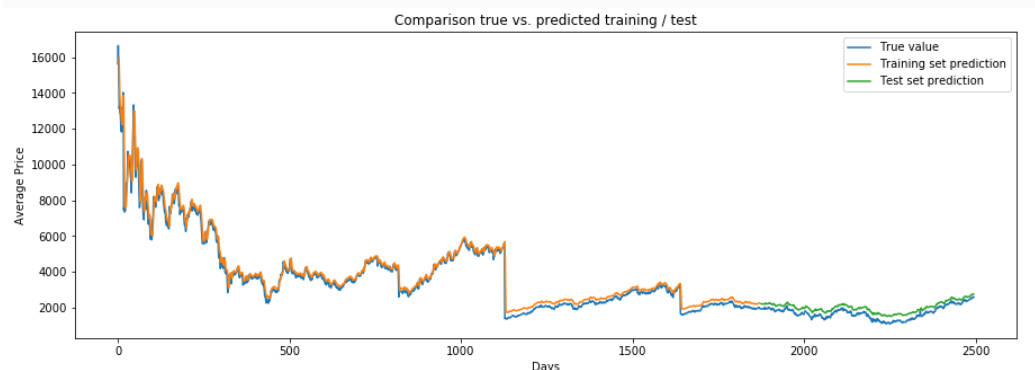
Let us now check the MSE in train and test data and perform the corresponding visualization:

1. def predict\_and\_score(model, X, Y):
2. *# Make predictions on the original scale of the data.*
3. pred = scaler.inverse\_transform(model.predict(X))
4. *# Prepare Y data to also be on the original scale for interpretability.*
5. orig\_data = scaler.inverse\_transform([Y])
6. *# Calculate RMSE.*
7. score = np.sqrt(mean\_squared\_error(orig\_data[0], pred[:, 0]))
8. return(score, pred)
9. rmse\_train, train\_predict = predict\_and\_score(model, train\_X, train\_Y)
10. rmse\_test, test\_predict = predict\_and\_score(model, test\_X, test\_Y)
11. print("Training data score: %.2f RMSE" % rmse\_train)
12. print("Test data score: %.2f RMSE" % rmse\_test)

Training data score: 394.74 RMSE

Test data set score: 313.33 RMSE

1. *# Initiating with training predictions.*
2. train\_predict\_plot = np.empty\_like(scaled\_price)
3. train\_predict\_plot[:, :] = np.nan
4. train\_predict\_plot[window\_size:len(train\_predict) + window\_size, :] = train\_predict
5. *# Adding test predictions.*
6. test\_predict\_plot = np.empty\_like(scaled\_price)
7. test\_predict\_plot[:, :] = np.nan
8. test\_predict\_plot[len(train\_predict) + (window\_size \* 2) + 1:len(scaled\_price) - 1, :] = test\_predict
9. *# Creating the plot.*
10. plt.figure(figsize = (15, 5))
11. plt.plot(scaler.inverse\_transform(scaled\_price), label = "True value")
12. plt.plot(train\_predict\_plot, label = "Training set prediction")
13. plt.plot(test\_predict\_plot, label = "Test set prediction")
14. plt.xlabel("Days")
15. plt.ylabel("Average Price")
16. plt.title("Comparison true vs. predicted training / test")
17. plt.legend()
18. plt.show()

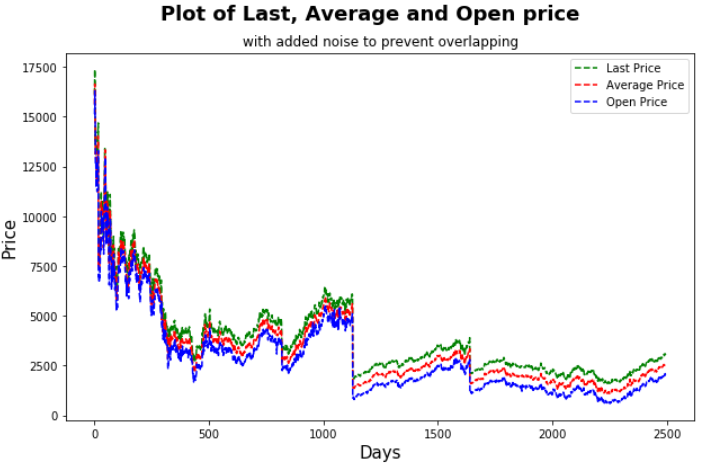


As you can observe just with four LSTM blocks and three epochs, the results are quite astonishing. However, mostly this is due to overfitting which can be reduced by adding Dropout layer in the network design and tuning basic hyperparameters.

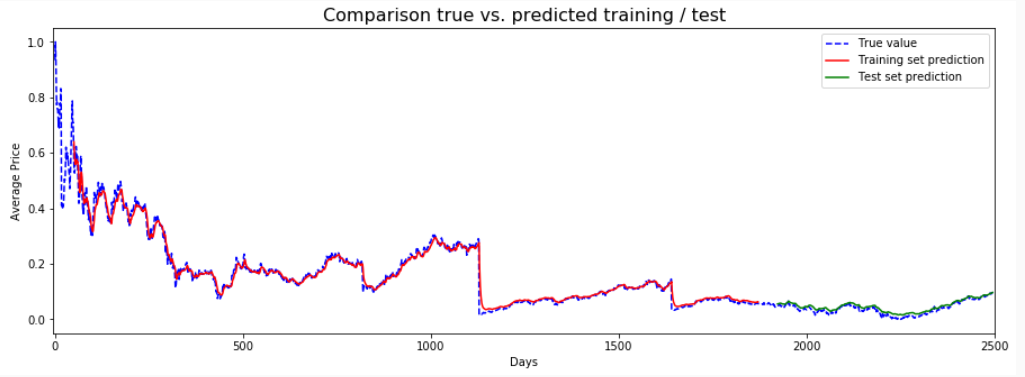
In the previous page, you have learned how to perform forecasting based on a single input. In this section, you are required to perform the multivariate time series forecast using three inputs.

Here's the starter code which takes three inputs: Last Price, Average Price, and Open Price and plot them.

1. *# Necessary libraries*
2. import numpy as np
3. import pandas as pd
4. import matplotlib.pyplot as plt
5. import keras
6. from keras.models import Sequential
7. from keras.layers.recurrent import LSTM
8. from keras.layers.core import Dense, Activation, Dropout
9. from sklearn.preprocessing import MinMaxScaler
10. *# Loading data*
11. data = pd.read\_csv('INFY\_2000\_2008.csv')
12. *# Selecting three columns as input*
13. data = data[['Last Price', 'Average Price', 'Open Price']].values
14. plt.figure(figsize=(10, 6))
15. plt.plot(data[:,0]+500, '--g', label='Last Price')
16. plt.plot(data[:,1], '--r', label='Average Price')
17. plt.plot(data[:,2]-500, '--b', label='Open Price')
18. plt.xlabel('Days', fontsize=15)
19. plt.ylabel('Price', fontsize=15)
20. plt.suptitle('Plot of Last, Average and Open price', fontsize=18, weight='bold')
21. plt.title('with added noise to prevent overlapping')
22. plt.legend()
23. plt.show()



You are required to split the dataset in the ratio of 75:25 and train an LSTM model with four units, single LSTM layer till three epochs. Here's the visualized output w.r.t. the Average Price.



Additional topics

# Handling Variable-Length Sequences

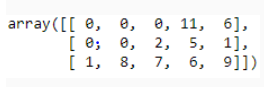
While building your model, there can be cases when the model may encounter variable-length sequences. For example:

* Sequence 1: [32, 45, 78, 98]
* Sequence 2: [1, 8]

Here, sequence 1 has a length four whereas sequence two has a length two. To handle such situations, Keras provides a method named **pad\_sequences** which helps in handling the length in a variety of ways. Given below are few ways by which you can control the length of sequences:

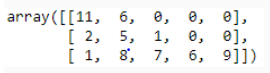
1. *# Importing method*
2. from keras.preprocessing.sequence import pad\_sequences
3. *# Creating dummy sequences stored in a Python list*
4. seq = [[11, 6], [2, 5, 1], [1, 8, 7, 6, 9]]

**1. Pre-sequence padding**

It adds zero at the beginning of each sequence to make them equal to the length of the largest sequence. This method is present in the pad\_sequences method by default. You can also call it using the argument padding='pre'.

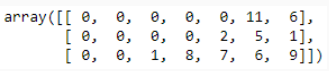
1. pad\_sequences(seq)
2. *# pad\_sequences(seq, padding='pre')*

**2. Post-sequence padding**

It adds zero at the end of each sequence to make them equal to the length of the largest sequence.

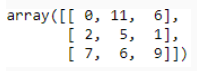
1. pad\_sequences(seq, padding='post')

**3. Maximum length padding**

It adds zero at the beginning to each sequence to make them equal to the value passed in the *maxlen*argument.

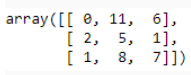
1. pad\_sequences(seq, maxlen=7)

**4. Minimum length padding: Pre-sequence padding**

If you pass a small value in the argument *maxlen*then it truncates each sequence by making their length equal to the value passed in it. Observe that padding takes places at the beginning and sequences are truncated from the beginning.

1. pad\_sequences(seq, maxlen=3)

**5. Minimum length padding: Post-sequence padding**

To perform the above operation but to truncate sequences from the end, use *truncating='post'*in the method.

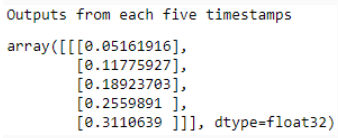
1. pad\_sequences(seq, maxlen=3, truncating='post')

# Fetching Hidden and Cell States of an LSTM Cell

While building an LSTM network, we can fetch the output value of the previous timestamp from the hidden layer using the return\_sequences argument passed in the LSTM method. This way we not only have the output of the final timestamp but also the subsequent timestamp outputs. It is not always beneficial to get the hidden state output every time, only for a few cases, this may be helpful like machine translation.

We will use one LSTM cell along with one hidden layer and try to get the output for five timestamps:

1. *# Importing necessary methods*
2. from keras.models import Model
3. from keras.layers import Input, LSTM
4. import numpy as np
5. *# Defining five inputs*
6. inputs = np.array([0.2, 0.3, 0.4, 0.5, 0.6]).reshape((1, 5, 1))
7. *# Defining LSTM network*
8. np.random.seed(42)
9. feed = Input(shape=(5, 1))
10. lstm = LSTM(1, return\_sequences=True)(feed)
11. model = Model(inputs=feed, outputs=lstm)
12. *# Predictions*
13. print('Outputs from each five timestamps')
14. model.predict(inputs)

Not only output (hidden state) but you can also fetch the cell state using the return\_state argument. Modify the above code with these two lines and observe the change:

1. lstm, state\_h, state\_c = LSTM(1, return\_sequences=True, return\_state=True)(feed)
2. model = Model(inputs=feed, outputs=(lstm, state\_h, state\_c))

# Dropouts

Usually, to apply dropout in a neural network you may use the Dropout method from the Keras such as:

1. ...
2. model.add(Dropout(0.3))
3. ...

However, if you try to use this mechanism of dropout for RNNs (like LSTM/GRU) it can interfere with the timestamps (can even drop them) unless you rely on its argument noise\_shape. Since we know that a recurrent layer takes two inputs at a timestamp, your input and the internal input (can be just the output of the previous state or including the cell state, depending upon the architecture used). Therefore, it is not always necessary that the output and/or cell state from the previous state may match the dimension of the current input. So, Keras provides two different dropouts to handle this situation.

* **dropout**: Current input
* **recurrent\_dropout**: Recurrent state (previous output and/or cell state)

Both of these arguments take a value between 0 and 1 to drop a fraction of units for the linear transformation of the respective input.

Predicting next word

To perform text generation using RNN, we need to show the model various examples to make a better prediction character by character. For this task, we will be using the Project Gutenberg's The Adventures of Sherlock Holmes, by Arthur Conan Doyle available as a **.txt** file. Download the file from [here](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_0127914004370472964/web-hosted/assets/SherlockHolmes.txt) and read the file in Python.

1. *# Necessary libraries*
2. from keras.models import Sequential
3. from keras.layers import Dense, Activation, LSTM, Dropout
4. from keras.optimizers import RMSprop
5. from keras.utils.data\_utils import get\_file
6. import keras
7. import random
8. import numpy as np
9. *# Reading the data*
10. text = open('Sherlock Holmes.txt').read().lower()
11. print('Given script has ' + str(len(text)) + ' characters')

**Given the script has 581862 characters**

Since the dataset is too long, therefore, let us strip the dataset and perform basis preprocessing.

1. text = text[1302:]
2. for ch in ['0','1', '2', '3', '4', '5', '6', '7', '8', '9', '!', '"', '$', '%', '&', '~', '`', '(', ')', '\*',
3. '-', '/', ';', '@', '?', ':', '©', '¢', 'ã', '\xa0', '\n', '\r', '.']:
4. if ch in text:
5. text=text.replace(ch,' ')
6. print(set(text))



Now, we can create a sliding window function in which all the characters inside the window are treated as input and the following character is treated as output. We use the window size of 50 and step size as 3.

1. def window\_transform(text, window\_size, step\_size):
2. inputs = []
3. outputs = []
5. n\_batches = int((len(text)-window\_size) / step\_size)
7. for i in range(n\_batches-1):
8. a = text[i \* step\_size:((i \* step\_size) + window\_size)]
9. inputs.append(a)
10. b = text[(i \* step\_size) + window\_size]
11. outputs.append(b)
12. return inputs,outputs
13. *# Calling the window function*
14. window\_size = 50
15. step\_size = 3
16. inputs, outputs = window\_transform(text, window\_size, step\_size)

Let us verify the results from the above function:

1. inputs[502], outputs[502]



As you can observe the length of the window (input size) is 50 which you can verify using **len(inputs[502])** and the corresponding output is the following character of the ongoing sentence which here is 'd'.

For a confirmation here is the sentence at input[503] sliding with a step size of 3 (taking three new characters).



Now, let us try to formulate the problem in the context of machine learning. Above we saw the output of **set(text)** resulted in 33 unique characters, therefore, the given problem formulates in the multi-class classification problem.

To begin with, we first sort the output of set(text) and map them to a unique numerical value.

1. *# Sorting the unique elements*
2. chars = sorted(list(set(text)))
3. *# Encoding*
4. chars\_to\_indices = dict((c, i) for i, c in enumerate(chars))
5. *# Decoding*
6. indices\_to\_chars = dict((i, c) for i, c in enumerate(chars))

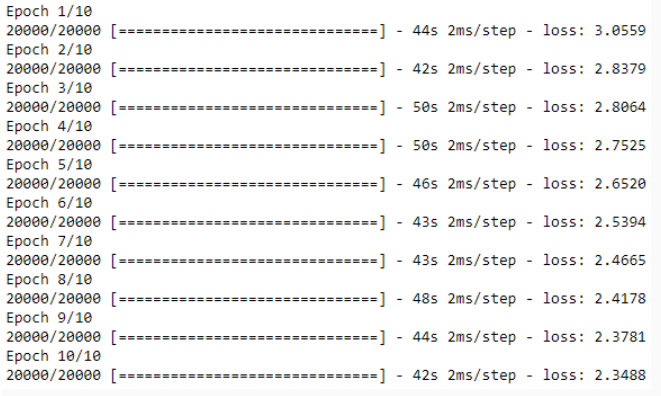
For instance, **chars\_to\_indices['r']**results in **20** which defines that character **r** is mapped to value **20**.

Now, we have each character mapped to a numeric value, it is time to transform the input/output vector in the same numeric format:

1. def encode\_io\_pairs(text, window\_size, step\_size):
2. num\_chars = len(chars)
4. *# cut up text into character input/output pairs*
5. inputs, outputs = window\_transform(text, window\_size, step\_size)
7. *# create empty vessels for one-hot encoded input/output*
8. X = np.zeros((len(inputs), window\_size, num\_chars), dtype=np.bool)
9. y = np.zeros((len(inputs), num\_chars), dtype=np.bool)
11. *# loop over inputs/outputs and tranform and store in X/y*
12. for i, sentence in enumerate(inputs):
13. for t, char in enumerate(sentence):
14. X[i, t, chars\_to\_indices[char]] = 1
15. y[i, chars\_to\_indices[outputs[i]]] = 1
17. return X,y
18. X, y = encode\_io\_pairs(text, window\_size, step\_size)

This completes the formatting of the data set. Now, we can build the LSTM network starting with the first layer having 120 nodes followed by a fully-connected linear layer and a softmax layer.

1. *# Designing the model*
2. model = Sequential()
3. model.add(LSTM(120, input\_shape=(window\_size, len(chars))))
4. model.add(Dropout(0.22))
5. model.add(Dense(len(chars), activation='linear'))
6. model.add(Dense(y.shape[1], activation='softmax'))
7. *# Compiling the model*
8. model.compile(loss='categorical\_crossentropy', optimizer='adam')
9. *# Subsetting data for an example*
10. Xsmall = X[:20000,:,:]
11. ysmall = y[:20000,:]
12. *# Model training*
13. model.fit(Xsmall, ysmall, batch\_size=500, epochs=10)

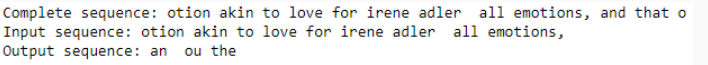
To proceed with the prediction, we need to follow a simple rule of thumb. We know that at a time, our script accepts a window size of 50 and takes the output as the 51st character. Following this rule, we need to predict a character, later remove the first character from our previous window and add the newly predicted character at the end making it still a window of size 50 then predict the second character and keep following the process.

This method along with the number of characters to be predicted is coded below:

1. def predict\_next\_chars(model, input\_chars, num\_to\_predict):
2. pred\_chars = ''
3. for i in range(num\_to\_predict):
4. *# Converting this round's predicted characters to numerical input*
5. x\_test = np.zeros((1, window\_size, len(chars)))
6. for t, char in enumerate(input\_chars):
7. x\_test[0, t, chars\_to\_indices[char]] = 1.
8. *# make this round's prediction*
9. test\_predict = model.predict(x\_test,verbose = 0)[0]
10. *# translate numerical prediction back to characters*
11. r = np.argmax(test\_predict)
12. d = indices\_to\_chars[r]
13. *# update predicted\_chars and input*
14. pred\_chars+=d
15. input\_chars+=d
16. input\_chars = input\_chars[1:]
17. return pred\_chars

Now, all you're left with is the prediction of the new characters which can be performed as shown:

1. *# Prediction*
2. start = 89
3. num\_to\_predict = 10
4. input\_chars = text[start: start + window\_size]
5. print('Complete sequence:', text[start:start + window\_size + num\_to\_predict])
6. print('Input sequence:', input\_chars)
7. print('Output sequence:', predict\_next\_chars(model, input\_chars, num\_to\_predict = num\_to\_predict))



With the given network design, the above results track exactly for the initial three characters which are **' ', 'a' and 'n'**. Predicting text is a hot topic in today's era and therefore it requires knowledge on linguistics, semantics and much more to develop a robust model.