8 ARTIFICAL NEURAL NETWORK II

In this chapter we discuss a major type of neural networks especially for financial analyses with time series – the recurrent neural network (RNN). This involves more a complicated architecture, and is sometimes referred to as deep learning NN to distinguish them from the feedforward NN (also called ANN) discussed in the previous chapter. We also discuss variants of the RNN such as the Long short-term memory (LSTM) NN, the Bidirectional RNN (BRNN), and Gated Recurrent Unit (GRU). Another major type of NN – the convolutional network (CNN) that is primarily used in image classification and recognition, is not covered in this chapter.

8.1 Recurrent Neural Network (RNN)

In Figure 8.1, we show N multilayer perceptrons (MLP), an identical one for each case/subject k. For each k, the inputs $\{X_{i,k}\}$ are features (i=1,2,…,3) related to subject k. The features/inputs are marked as dotted circles in the input layer. For examples, $\{X_{i,k}\}$ could be attributes/features i of customer k of a bank in the prediction of the outcome/output of exit or not (Y_k = 1 or 0); $\{X_{i,k}\}$ could be characteristics/features i of a house k in a particular region in the prediction of the house k price in the region (during a specified period). In Chapter 7, we see that when cases k are independent of one another, i.e., their inputs and output do not affect other cases, then the common (across k) input weights and biases, and weights and biases in hidden layer(s) are computed via backpropagation to minimize the error/loss function L(.) that combines all cases $\sum_{k=1}^{N} \frac{1}{N} L\left(Y_k, \hat{Z}_k\right)$ or else via mini-batches on smaller numbers of cases each iteration, or via stochastic gradient descent iteratively for each case. Y_k is the actual output in case k, and $\hat{Z}_k$ is the corresponding predicted output.

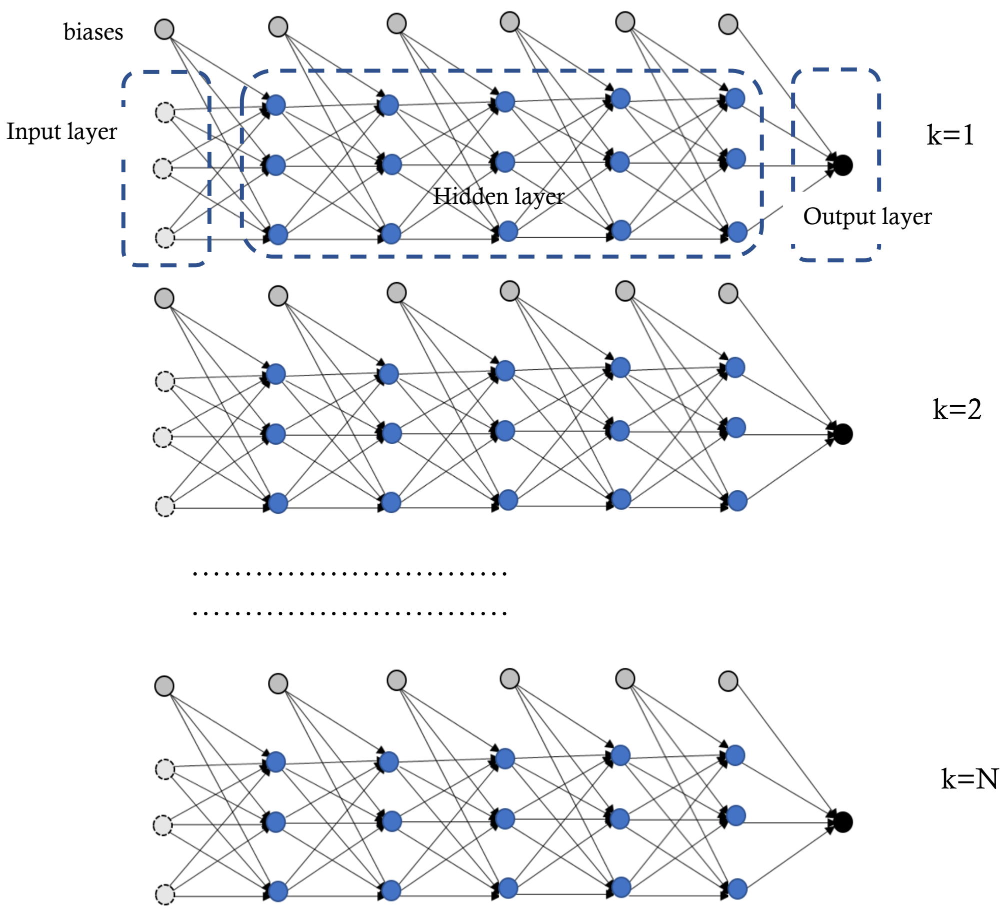


Figure 8.1

For each k, there are 3 × (3 + 1) = 12 parameters with forward pass to the first hidden layer – each first hidden layer neuron takes a forward pass of 3 input weights and 1 bias. Then there are 4 × 3 × (3 + 1) = 48 parameters in the next 4 hidden layers. Finally, there are (3+1) = 4 parameters in the output layer. Altogether there are 12 + 48 + 4 = 64 different parameters in this ANN. Note that the parameters in edges to each hidden layer neuron are all different. The trained 64 parameters are, however, the same for each k since the training is across the total loss function $\sum_{k=1}^{N} \frac{1}{N} L(Y_k, \hat{Z}_k)$ or else in mini-batches or stochastic gradient descent across each k, though keeping the final iteration set of parameters as common to all k. This trained set of parameters is then applied to predict the test data cases j.

Suppose now the inputs, instead of given altogether at the initial input layer, are given in a sequence. In the common setup, the number of hidden layers now equals to the number of sequential inputs so that at each additional hidden layer, there is a fresh sequential input – see dotted arrow in Figure 8.2. We consider the simple case where each sequential input is an item, e.g., a scalar number. More general cases include a number of inputs or features each time.

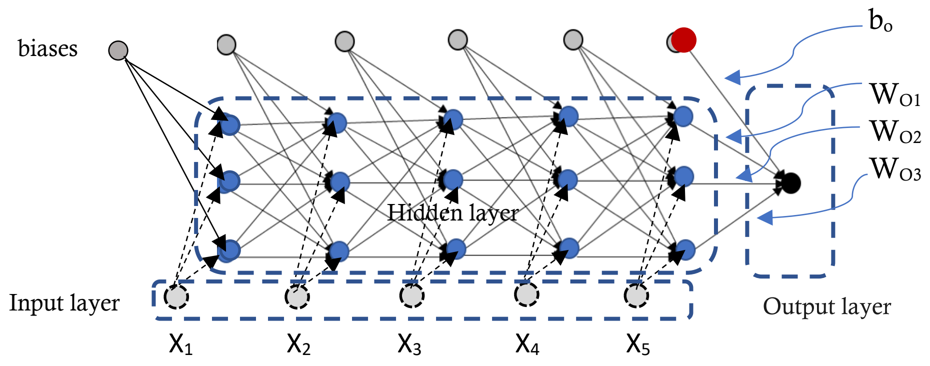


Figure 8.2

Consider the second hidden layer in the NN in Figure 8.2. Each neuron in this layer receives 5 ‘inputs’ and thus 5 associated weights/parameters in the edges/synapses. For example, the parameters to the first (uppermost) neuron in the layer are (1) W_{P1}, the weight on exogenous input X_2 from the input layer I, (2) the bias b_1, (3) W_{11}, the weight on forward pass from previous hidden layer first neuron, (4) W_{21}, the weight on forward pass from previous hidden layer second neuron, (5) W_{31}, the weight on forward pass from previous hidden layer third neuron.

The parameters to the second neuron in the layer are (1) W_{P2}, the weight on exogenous input X_2 from the input layer I, (2) the bias b_2, (3) W_{12}, the weight on forward pass from previous hidden layer first neuron, (4) W_{22}, the weight on forward pass from previous hidden layer second neuron, (5) W_{32}, the weight on forward pass from previous hidden layer third neuron.

The parameters to the third neuron in the layer are (1) W_{P3}, the weight on exogenous input X_2 from the input layer I, (2) the bias b_3, (3) W_{13}, the weight on forward pass from previous hidden layer first neuron, (4) W_{23}, the weight on forward pass from previous hidden layer second neuron, (5) W_{33}, the weight on forward pass from previous hidden layer third neuron. The above constitute 3 × (1 + 1 + 3) = 15 parameters. These are the same constants for each hidden layer forward – this is where it differs from the ANN or forward pass NN. At the output layer, there are the bias parameter b_0, and the 3 weight parameters W_{O1}, W_{O2}, and W_{O3}. In general, for this architecture, there is a total of m × (m+2) + (m+1) number of parameters to be optimized, where m is the number of neurons in each hidden layer and there is only one output neuron.

In Figure 8.2, the output from the first neuron in the first hidden layer is

$$\tanh(W_{P1} \times X_1 + b_1 + W_{11}\, h_{01} + W_{21}\, h_{02} + W_{31}\, h_{03}) \qquad (8.1)$$

where h_{tj} is a hidden state (explicitly unobserved) at sequence step t and from neuron j counting from the top. The initialized values of h_{01}, h_{02}, h_{03} could be zeros or it could be set by a random function in Keras App based on random normal numbers. Set this output as the hidden state h_{11} to be forward pass to the sequence at t=2. Note that tanh is commonly used as the activation function in this type of architecture. Output from the second neuron in the first hidden layer is

$$h_{12} \equiv \tanh(W_{P2} \times X_1 + b_2 + W_{12}\, h_{01} + W_{22}\, h_{02} + W_{32}\, h_{03}).$$

Output from the third neuron in the first hidden layer is

$$h_{13} \equiv \tanh(W_{P3} \times X_1 + b_3 + W_{13}\, h_{01} + W_{23}\, h_{02} + W_{33}\, h_{03}).$$

The output from the first neuron in the second hidden layer is

$$h_{21} \equiv \tanh(W_{P1} \times X_2 + b_1 + W_{11}\, h_{11} + W_{21}\, h_{12} + W_{31}\, h_{13}).$$

Output from the second neuron in the second hidden layer is

$$h_{22} \equiv \tanh(W_{P2} \times X_2 + b_2 + W_{12}\, h_{11} + W_{22}\, h_{12} + W_{32}\, h_{13}).$$

Output from the third neuron in the second hidden layer is

$$h_{23} \equiv \tanh(W_{P3} \times X_2 + b_3 + W_{13}\, h_{11} + W_{23}\, h_{12} + W_{33}\, h_{13}),$$

and so on.

The hidden states (or output from last hidden layer neurons) are h_{51}, h_{52}, and h_{53}. Therefore, the output is

$$\tanh(W_{O1} \times h_{51} + W_{O2}\, h_{52} + W_{O3}\, h_{53} + b_0).$$

It is noted that the hidden states (at each level of neuron) change with each step in the sequence. Each hidden state at sequence step t contains information of lagged exogenous inputs X_{t-1}, X_{t-2}, X_{t-3}, and so on. This type of architecture is important if indeed current inputs are not independent from lagged inputs that have useful information on predicting the output. This type of neural network is called Recurrent Neural Network (RNN).

The RNN of Figure 8.2 may be represented in a concise manner as follows.

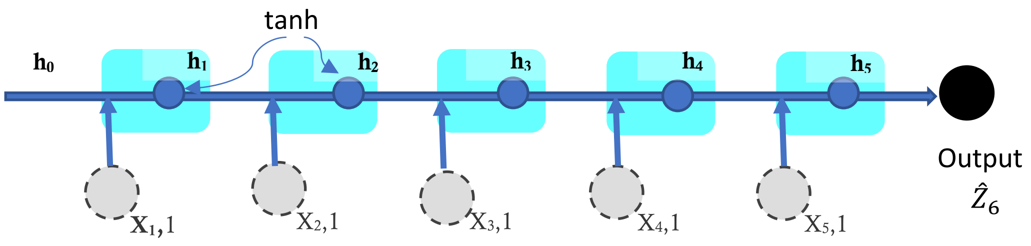


Figure 8.3

The additional "1" in the input box reflects the "input" to multiply with the bias coefficient. It is sometimes also represented schematically as follows in Figure 8.4 by compacting hidden layers and sequential inputs. In this diagram, the recurrence of the hidden layer outputs is made more obvious.

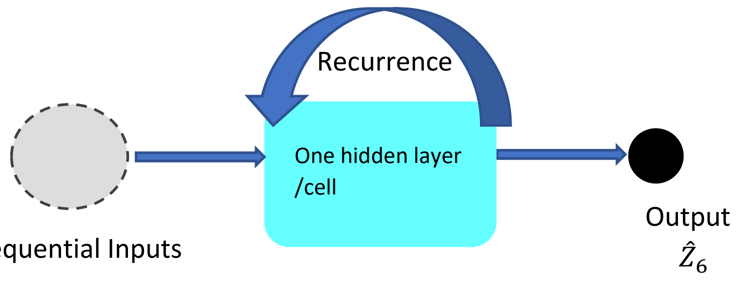


Figure 8.4

Figures 8.2, 8.3 are sometimes called the unfolded NN of the above diagram. In Figure 8.3, there is concatenation of the inputs (X_t , 1) and $\mathbf{h_{t-1}} \equiv (h_{t-1,1}$, $h_{t-1,2}$, $h_{t-1,3})$ at their junction, and this is passed through the hidden layer (neurons inside layer and synapses or edges of weights and biases are not shown) where the dark circle represents the activation function on the dot product of the concatenated inputs and parameters as in Eq. (8.1). The output of that hidden layer is then $\mathbf{h_t}$ which enters into the computations at the next hidden layer. Each of the hidden layer may be viewed as a recurrent cell since the computations are recurrent on the updated hidden states as the sequence progresses until the output.

In time series, the input features at each time point, … X_{t-1}, X_t, X_{t+1}, and so on, may not be independent. For example, the vector X_t may be autocorrelated with X_{t-1} and so on, i.e., X_t is a vector autoregressive (VaR) stochastic process. This means that some, if not all, of each variable in the vector, e.g., X_{it}, may be autocorrelated. In general, X_t is a function of X_{t-1}, X_{t-2}, and so on, for every t. Thus, when the features are not independent across time, past history of X_t such as X_{t-1}, X_{t-2}, X_{t-3}, and so on, may, besides current input X_t, affect predicted output $\hat{Y}_t$. This can be modelled by adjusting the architecture of the NN such that the activation function at the hidden layer at t, $f_1(t)$ includes not just input X_t but also lagged X_{t-1}, X_{t-2}, X_{t-3}, and so on. Hence the RNN is suitable for use in predicting time series sequences. It is also useful in other tasks such as speech recognition, optical music recognition with sequences of music scores, emotional facial recognition using snapshots of facial expressions, language translation via text sequences, gesture recognition (useful in robotics), sentiment classification, and so on. Popular applications incorporating such software technologies are Siri, Voice Search, Google Translate, and so on.

In the training of the RNN shown in Figures 8.2, 8.3, we treat each of the 5 recurrent cells like 5 hidden layers and use backpropagation to compute the required partial derivatives of the loss function with respect to the (m+2) (m+1) – 1 number of parameters where m is the number of neurons in each hidden layer and there is only one output neuron. Figure 8.2 shows a particular case of 5 sequential inputs. In general, a total sample could have a time series or a sequence of T cells. The cells are then divided into cases or packs each with R, e.g., 5, number of recurrent cells/hidden layers. The number of cases or packs is T/R. While the sequence within each pack clearly considers dependence and recurrence, the different cases or packs are treated as independent when it comes to iterations in optimizing the parameters as batches of packs can be randomly selected in updating.

A certain number of cases/packs are grouped into batches with batch size S. So, there are T/(RS) number of mini-batches (each size S), each of which requires one parameter update in one epoch. The number of iterations is equal to T/(RS) times the number of epochs as in the mini-batch method in MLP. When S = 1, it is similar to the stochastic gradient descent method when the selected case is random. Some gradient descent approach such as Adam (with changing learning rate) can be applied to adjust the parameters to minimize the loss function via iterations.

Consider the loss function of one case, e.g., Figure 8.2, $L(Y_{t+R}, \hat{Y}_{t+R})$ where $\hat{Y}_{t+R} \equiv \hat{Z}_{t+R}$ and the sequence of inputs are (X_t, X_{t+1}, X_{t+2}, …, X_{t+R-1}). In the back propagation, each of the parameters W_{P1} , W_{P2} , … , W_{Pm} , W_{11} , W_{21} , … , W_{m1} , W_{12} , W_{22} , … ,W_{m2} , W_{13} , W_{23} , … ,W_{m3} , ………, W_{1m} ,

W_{2m} , … , W_{mm} , b_1 , b_2 , … , b_m, occurs as the same constants in every hidden layer or time-sequence step, and parameters W_{O1}, W_{O2}, W_{O3}, b_o, occur at the output layer.

The partial derivative with respect to each of the output layer parameter is, where R=5 and $\hat{Y}_{t+5} =$ tanh($W_{O1} \times h_{51} + W_{O2}\, h_{52} + W_{O3}\, h_{53} + b_0$), for j = 1,2, or 3:

$$\frac{\partial L(Y_{t+5}, \hat{Y}_{t+5})}{\partial W_{Oj}} = \frac{\partial L(Y_{t+5}, \hat{Y}_{t+5})}{\partial \hat{Y}_{t+5}} \times \left(1 - \hat{Y}_{t+5}{}^2\right) \times \frac{\partial (W_{O1} \times h_{51} + W_{O2} \times h_{52} + W_{O3} \times h_{53} + b_o)}{\partial W_{Oj}}$$
$$= \frac{\partial L(Y_{t+5}, \hat{Y}_{t+5})}{\partial \hat{Y}_{t+5}} \times \left(1 - \hat{Y}_{t+5}{}^2\right) \times h_{5j}$$

Note that d tanh(z) / dz = 1 – (tanh(z))^2.

$$\frac{\partial L(Y_{t+5}, \hat{Y}_{t+5})}{\partial b_O} = \frac{\partial L(Y_{t+5}, \hat{Y}_{t+5})}{\partial \hat{Y}_{t+5}} \times \left(1 - \hat{Y}_{t+5}{}^2\right) \times \frac{\partial (W_{O1} \times h_{51} + W_{O2} \times h_{52} + W_{O3} \times h_{53} + b_o)}{\partial b_O}$$
$$= \frac{\partial L(Y_{t+5}, \hat{Y}_{t+5})}{\partial \hat{Y}_{t+5}} \times \left(1 - \hat{Y}_{t+5}{}^2\right)$$

Now,

$h_{51} \equiv \tanh(W_{P1} \times X_5 + b_1 + W_{11}\, h_{41} + W_{21}\, h_{42} + W_{31}\, h_{43})$,

$h_{52} \equiv \tanh(W_{P2} \times X_5 + b_2 + W_{12}\, h_{41} + W_{22}\, h_{42} + W_{32}\, h_{43})$,

$h_{53} \equiv \tanh(W_{P3} \times X_5 + b_3 + W_{13}\, h_{41} + W_{23}\, h_{42} + W_{33}\, h_{43})$,

$h_{41} \equiv \tanh(W_{P1} \times X_4 + b_1 + W_{11}\, h_{31} + W_{21}\, h_{32} + W_{31}\, h_{33})$,

$h_{42} \equiv \tanh(W_{P2} \times X_4 + b_2 + W_{12}\, h_{31} + W_{22}\, h_{32} + W_{32}\, h_{33})$,

$h_{43} \equiv \tanh(W_{P3} \times X_4 + b_3 + W_{13}\, h_{31} + W_{23}\, h_{32} + W_{33}\, h_{33})$,

…………….

$h_{11} \equiv \tanh(W_{P1} \times X_1 + b_1 + W_{11}\, h_{01} + W_{21}\, h_{02} + W_{31}\, h_{03})$

$h_{12} \equiv \tanh(W_{P2} \times X_1 + b_2 + W_{12}\, h_{01} + W_{22}\, h_{02} + W_{32}\, h_{03})$.

$h_{13} \equiv \tanh(W_{P3} \times X_1 + b_3 + W_{13}\, h_{01} + W_{23}\, h_{02} + W_{33}\, h_{03})$,

$h_{01} = h_{02} = h_{03} = 0$.

The partial derivative with respect to each of the hidden layer parameters is

$$\frac{\partial L(Y_{t+5}, \hat{Y}_{t+5})}{\partial W_{P1}} = \frac{\partial L(Y_{t+5}, \hat{Y}_{t+5})}{\partial \hat{Y}_{t+5}} \times \frac{\partial \hat{Y}_{t+5}}{\partial W_{P1}} = \frac{\partial L(Y_{t+5}, \hat{Y}_{t+5})}{\partial \hat{Y}_{t+5}} \times \left(1 - \hat{Y}_{t+5}{}^2\right) \times \frac{\partial (W_{O1} \times h_{51} + W_{O2} \times h_{52} + W_{O3} \times h_{53} + b_o)}{\partial W_{P1}} \qquad (8.2)$$

where

$$\frac{\partial h_{51}}{\partial W_{P1}} = \left(1 - h_{51}{}^2\right) \frac{\partial (W_{P1}\, X_5 + W_{11}\, h_{41} + W_{21}\, h_{42} + W_{31}\, h_{43} + b_1)}{\partial W_{P1}}$$
$$= \left(1 - h_{51}{}^2\right) \left(X_5 + W_{11} \frac{\partial h_{41}}{\partial W_{P1}} + W_{21} \frac{\partial h_{42}}{\partial W_{P1}} + W_{31} \frac{\partial h_{43}}{\partial W_{P1}}\right)$$

$$\frac{\partial h_{52}}{\partial W_{P1}} = \left(1 - {h_{52}}^2\right) \frac{\partial(W_{P2}\, X_5 + W_{12}\, h_{41} + W_{22}\, h_{42} + W_{32}\, h_{43} + b_2)}{\partial W_{P1}}$$

$$= \left(1 - {h_{52}}^2\right)\left(0 + W_{12}\frac{\partial h_{41}}{\partial W_{P1}} + W_{22}\frac{\partial h_{42}}{\partial W_{P1}} + W_{32}\frac{\partial h_{43}}{\partial W_{P1}}\right)$$

$$\frac{\partial h_{53}}{\partial W_{P1}} = \left(1 - {h_{53}}^2\right) \frac{\partial(W_{P3}\, X_5 + W_{13}\, h_{41} + W_{23}\, h_{42} + W_{33}\, h_{43} + b_3)}{\partial W_{P1}}$$

$$= \left(1 - {h_{53}}^2\right)\left(0 + W_{13}\frac{\partial h_{41}}{\partial W_{P1}} + W_{23}\frac{\partial h_{42}}{\partial W_{P1}} + W_{33}\frac{\partial h_{43}}{\partial W_{P1}}\right)$$

$$\frac{\partial h_{41}}{\partial W_{P1}} = \left(1 - {h_{41}}^2\right) \frac{\partial(W_{P1}\, X_4 + W_{11}\, h_{31} + W_{21}\, h_{32} + W_{31}\, h_{33} + b_1)}{\partial W_{P1}}$$

$$= \left(1 - {h_{41}}^2\right)\left(X_4 + W_{11}\frac{\partial h_{31}}{\partial W_{P1}} + W_{23}\frac{\partial h_{32}}{\partial W_{P1}} + W_{33}\frac{\partial h_{33}}{\partial W_{P1}}\right)$$

and so on. Hence Eq. (8.2) clearly involves summation of products of inputs at all times, previous parameter values $\mathrm{W}_{\mathrm{ij}}^{[\mathrm{t}]}$, and so on. This is unlike the feedforward NN case where a partial derivative would involve summation across paths originating from the parameter edge and not across each layer. After the updated partial derivatives are found, the usual adjustment of the parameter weights are done, viz.

$$\mathrm{W}_{\mathrm{ij}}^{[\mathrm{t}+1]} = \mathrm{W}_{\mathrm{ij}}^{[\mathrm{t}]} - \alpha(\mathrm{t}+1)\, \frac{\partial \mathrm{L}}{\partial \mathrm{W}_{\mathrm{ij}}^{[\mathrm{t}]}}$$

While the advantage of RNN over MLP is the ability to recognize the input of lagged (historical) information in affecting current output, and the ability to take in long sequence of inputs (since the number of parameters is not blown up as the parameters are constant at each recurrent cell), the computational time can be longer as the back propagation involves more computations of the complicated partial derivatives. RNN also has two drawbacks: (1) the partial derivatives in the RNN back propagation may in some situations progressively collapse to zero (as the tanh activation on arguments close to zero in turn produces output close to zero) or else they may explode exponentially rendering further computations/iterations impossible; (2) the design does not allow it to consider future input for training since the time-sequence uses information or input from the past till the current.

Problem (1) may be remedied by pruning the model such as reducing the number of recurrent cells to make the model less complex. It may also be remedied by a variant RNN called the Long Short-term Memory NN (LSTM) or use additional gates to control loss of gradient. Problem (2) may be remedied by using a variant of RNN, or LSTM, or other deep NN, called the Bidirectional RNN or BRNN. We shall discuss these later in section 8.3.

Besides the example in Figure 8.2 of many inputs to one output (many-to-one structure), there are also other structures such as one-to-many, e.g., for music generation or predicting a sequence of notes from one input, where at each step the activation becomes an explicit output that can be trained with actual output, and many-to-many as in text translation. We can also use the structure in Figure 8.2 for many-to-many if there is training on an output at the end of each hidden layer in a case.

8.2 Worked Example I – Data

The CBOE Volatility Index (VIX) is a real-time index derived from the prices of SPX (S&P 500) index options with near-term (approximately 1 month) expiration dates. It is market expectation of the SPX risk-neutral volatility over the short term. VIX has been called a ‘fear gauge’ and is often seen as a measure of negative market sentiment. Data is downloaded from Yahoo Finance.

The following exercise uses daily VIX data from 2 Jan 1990 to 26 Oct 2022 (8270 time-sequenced sample points) to perform a RNN prediction of next day VIX based on the observed VIX of the past 10 days (approximately 2 trading weeks), the inputs. The total data set is split into 80% training data (6616 observations) and 20% test data (1654 observations) – see code line [5], [6], [7]. See demonstration file Chapter8-1.ipynb.

```
In [1]: ### this notebook is about RNN on predicting daily S&P500 VIX
        import pandas as pd ### this automatically will call up 'from pandas import read_csv' when pd.read.csv... is used
        import numpy as np
        from keras.models import Sequential
        from keras.layers import Dense, SimpleRNN
        from sklearn.preprocessing import MinMaxScaler
        from sklearn.metrics import mean_squared_error
        import math
        import matplotlib.pyplot as plt
        ### imports apps (specialized functions)
```

```
In [2]: Vx = pd.read_csv("VIX.csv")
        print(Vx)
```

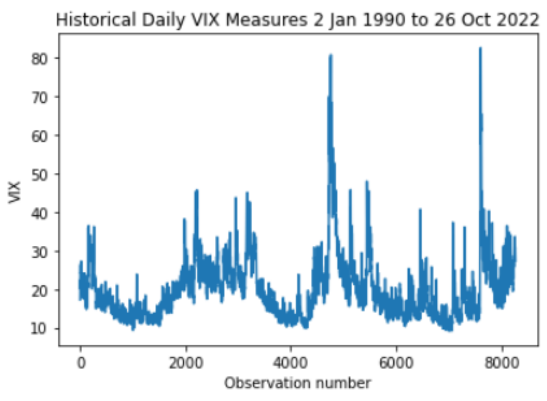
Snapshot of the data is shown as follows.

```
           Date       Open       High        Low      Close  Adj Close Volume
0      2/1/1990  17.240000  17.240000  17.240000  17.240000  17.240000      0
1      3/1/1990  18.190001  18.190001  18.190001  18.190001  18.190001      0
2      4/1/1990  19.219999  19.219999  19.219999  19.219999  19.219999      0
3      5/1/1990  20.110001  20.110001  20.110001  20.110001  20.110001      0
4      8/1/1990  20.260000  20.260000  20.260000  20.260000  20.260000      0
...         ...        ...        ...        ...        ...        ...    ...
8265 20/10/2022  31.299999  31.320000  29.760000  29.980000  29.980000      0
8266 21/10/2022  30.209999  30.440001  29.240000  29.690001  29.690001      0
8267 24/10/2022  30.650000  30.950001  29.780001  29.850000  29.850000      0
8268 25/10/2022  29.799999  30.000000  28.219999  28.459999  28.459999      0
8269 26/10/2022  28.440001  28.520000  27.270000  27.280001  27.280001      0

[8270 rows x 7 columns]
```

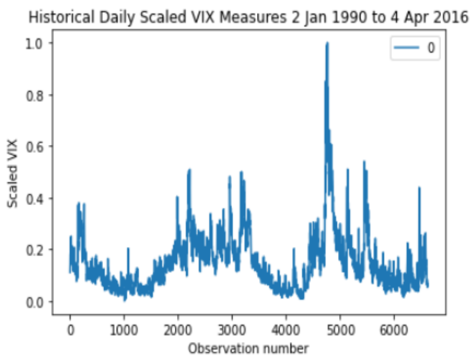
```
In [3]: VIX=Vx["Adj Close"]
        ### if we use VIX=Vx.iloc[:, 5:6].values -- then we get VIX as a numpy array and we need to convert back to pandas before
        ### we can use some pandas print and plot functions
```

```
In [4]: ### view time series graph of data -- is it stationary?
        VIX.plot()
        plt.xlabel('Observation number')
        plt.ylabel('VIX')
        plt.title('Historical Daily VIX Measures 2 Jan 1990 to 26 Oct 2022')
```

```
Out[4]: Text(0.5, 1.0, 'Historical Daily VIX Measures 2 Jan 1990 to 26 Oct 2022')
```



```
In [5]: ### we cannot use 'train_data = VIX.sample(frac=0.8, random_state=1); test_data = VIX.drop(train_data.index)'
        ### as this is time series data that is sequenced and cannot be randomly chopped up
        split_percent=0.8
        n = len(VIX) ### len() is a numpy array function
        split = int(n*split_percent) ### point for splitting data into train and test
        train_data = VIX[range(split)]
        test_data = VIX[split:]

In [6]: train_data.shape, test_data.shape  ### 0.8 x 8270 = 6616
Out[6]: ((6616,), (1654,))

In [7]: type(train_data), type(test_data)
Out[7]: (pandas.core.series.Series, pandas.core.series.Series)

In [8]: train_data.values.reshape(-1, 1)
Out[8]: array([[17.24    ],
               [18.190001],
               [19.219999],
               ...,
               [13.95    ],
               [13.1     ],
               [14.12    ]])

In [9]: #Performing Feature Scaling
        #from sklearn.preprocessing import StandardScaler
        from sklearn.preprocessing import MinMaxScaler
        scaler = MinMaxScaler(feature_range=(0, 1))  ### putting all positive is suitable as vola are all pos nos.
        train_data = scaler.fit_transform(train_data.values.reshape(-1, 1)) ### (-1,1) reshapes it to a 2D array;
        ### without .values - it may not work; .values .values convert to np array, the axes labels will be removed.
        test_data = scaler.fit_transform(test_data.values.reshape(-1, 1)) ### (-1,1) reshapes it to a 2D array

In [10]: type(train_data), type(test_data)
Out[10]: (numpy.ndarray, numpy.ndarray)

In [11]: train_data.shape, test_data.shape
Out[11]: ((6616, 1), (1654, 1))

In [12]: ### view scaled time series graph of data -- need to convert np array back to pandas df to use plot feature
         trdata=pd.DataFrame(train_data)
         trdata.plot()
         plt.xlabel('Observation number')
         plt.ylabel('Scaled VIX')
         plt.title('Historical Daily Scaled VIX Measures 2 Jan 1990 to 26 Oct 2022')

Out[12]: Text(0.5, 1.0, 'Historical Daily Scaled VIX Measures 2 Jan 1990 to 4 Apr 2016')
```



In codeline [13], the training data (train_data) and the test data (test_data) are arranged for the training and then for prediction as shown in Figure 8.5.

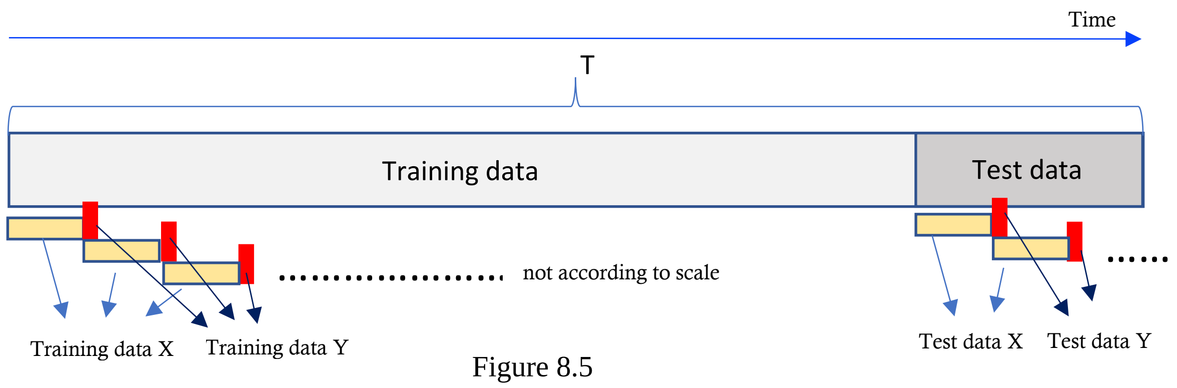


Figure 8.5

```
In [13]: ### Preparing the input X and target Y
         def get_XY(dat, time_steps):
             ### Indices of target array
             C_ind = np.arange(time_steps, len(dat), time_steps)
                 ### example np.arange(start=1, stop=10, step=3) gives array([1, 4, 7]), ends up to/before stop
             C = dat[C_ind]
                 ### example: ray=np.arange(2, stop=10, step=3); print(ray) --- gives [2 5 8]
                 ### c=np.array([1,3,6,8,9,10,12,15,18,20]); c[ray] --- gives array([ 6, 10, 18])
                 ### with elements from the 2nd, 5th, 8th positions of c. c's 1st position starts at '0'

             ### Prepare X
             rows_x = len(C)                          ### here len(C) is 661
             X = dat[range(time_steps*rows_x)]        ### range(L) is 0,1,2,..., L-1. L is 10 x 661 = 6610. X is array 1,2,...,6610
             X = np.reshape(X, (rows_x, time_steps, 1))  ### X rshaped as (661,10,1)
             return X, C

         time_steps = 10  ### hence C_ind = array ([10, 20, 30, 40, ....,661]), 661 number of 10 steps
                          ### C = array(10th position, 20th position of dat, etc.)
                          ### -- approx two weeks (10 trading days) interval for one prediction point of VIX
         trainX, trainY = get_XY(train_data, time_steps)
         testX, testY = get_XY(test_data, time_steps)
```

In this case we use a sequence of 10 steps (approximately two weeks of 10 trading days) to form one pack of 10 recurrent cells. These 10 daily VIX prices of training data X (inputs) are then followed by the following 11th day of VIX price as the training data for output Y. The training data set is divided into T/R number of the packs. The test data are similarly arranged into packs of 10 inputs followed by 1 output. The cases or packs are non-overlapping – meaning that any training does not use overlapped training data and thus avoids "double-counting" or biasing trained parameters when some overlapped packs could be more impactful in the updates. Using the non-overlapping design is reasonable if T is adequately large.

```
In [14]: len(train_data), type(train_data)

Out[14]: (6616, numpy.ndarray)

In [15]: trainX.shape, trainY.shape
         ### trainX has 661 elements in dimension 1, each with 10 elements in dimension 2; 1st element in dim 1 has first 10 data points,
         ### 2nd element in dim 1 has next 10 data points, etc.
         ### train Y has 661 elements in dimension 1 and trivial dimension 2 -- essentially one column

Out[15]: ((661, 10, 1), (661, 1))

In [16]: testX.shape, testY.shape
         ### testX has 165 elements in dimension 1, each with 10 elements in dimension 2; 1st element in dim 1 has first 10 data points,
         ### 2nd element in dim 1 has next 10 data points, etc.
         ### testY has 165 elements in dimension 1 and trivial dimension 2 -- essentially one column

Out[16]: ((165, 10, 1), (165, 1))
```

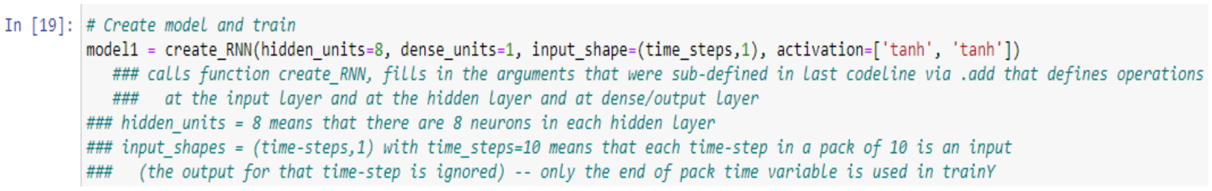
Code line [17] checks the training data X (trainX), training data Y (trainY), and so on to ensure all is in order.

```
In [17]: print(trainX[0,0:10,:],trainY[1,:])
         print(trainX[1,0:10,:],trainY[2,:])
         print(trainX[2,0:10,:],trainY[3,:])
         print(trainX[660,0:10,:],trainY[660,:])
         ### the printed shows first ten elements/inputs in trainX matched against the 11th element (1st element of trainY)
         ### then shows the last (661) ten elements in trainX against the 661th element in trainY
```

In code line [18], the SimpleRNN app in Keras Sequential is used to define the structure including the loss function of mean squared error for prediction and Adam for optimizing algorithm.

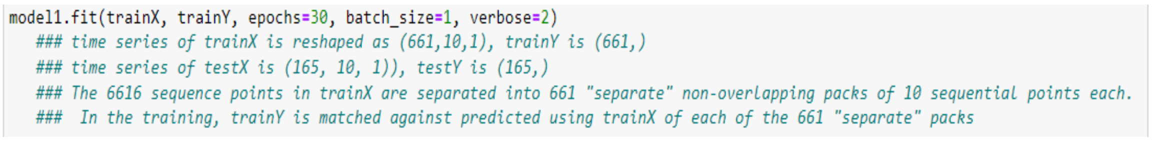
```
In [18]: def create_RNN(hidden_units, dense_units, input_shape, activation):
             model = Sequential()
             model.add(SimpleRNN(hidden_units, input_shape=input_shape, activation=activation[0]))
                ### See SimpleRNN apps in https://www.tensorflow.org/api_docs/python/tf/keras/layers/SimpleRNN
             model.add(Dense(units=dense_units, activation=activation[1]))
                ### Using model = Sequential() allows defining -- no. of inputs,#neurons in hidden,#neurons in output layer
             model.compile(loss='mean_squared_error', optimizer='adam')
                ### .compile in Sequential carries loss and optimizer options
             return model
```

Code line [19] shows the specifications of the structure in [18], e.g., m = 8 neurons in each recurrent hidden cell, output neuron (called ‘dense_units’ here) = 1, and activation function for hidden layer and for output layer that are both the tanh function. These arguments are defined in the Keras SimpleRNN app.



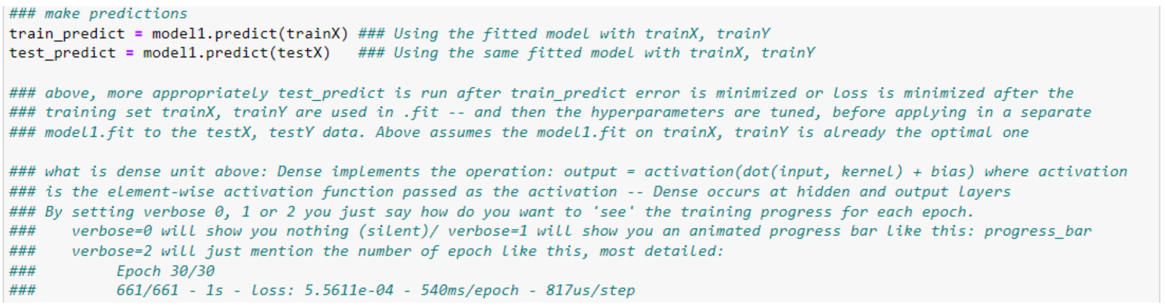
```
In [19]: # Create model and train
         model1 = create_RNN(hidden_units=8, dense_units=1, input_shape=(time_steps,1), activation=['tanh', 'tanh'])
           ### calls function create_RNN, fills in the arguments that were sub-defined in last codeline via .add that defines operations
           ###   at the input layer and at the hidden layer and at dense/output layer
         ### hidden_units = 8 means that there are 8 neurons in each hidden layer
         ### input_shapes = (time-steps,1) with time_steps=10 means that each time-step in a pack of 10 is an input
         ###   (the output for that time-step is ignored) -- only the end of pack time variable is used in trainY
```

Model1.fit is then called (via the Sequential app) to execute the training fit (minimizing loss based on number of iterations specified in number of epochs and batch size: 30 × 661 (T/R)) to find the optimal (m+2)(m+1) – 1 = 10 × 9 – 1 = 89 parameters.



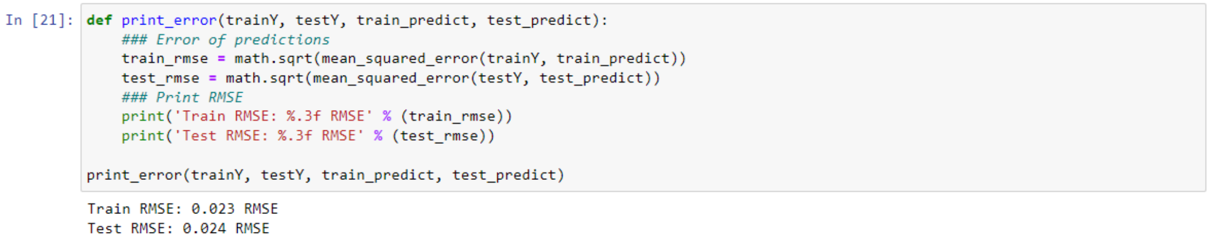
```
model1.fit(trainX, trainY, epochs=30, batch_size=1, verbose=2)
   ### time series of trainX is reshaped as (661,10,1), trainY is (661,)
   ### time series of testX is (165, 10, 1)), testY is (165,)
   ### The 6616 sequence points in trainX are separated into 661 "separate" non-overlapping packs of 10 sequential points each.
   ###  In the training, trainY is matched against predicted using trainX of each of the 661 "separate" packs
```

This is followed by making predictions on trainY based on trainX (using the optimized .fit parameters), and then making predictions on testY based on testX (using the optimized .fit parameters).



```
### make predictions
train_predict = model1.predict(trainX) ### Using the fitted model with trainX, trainY
test_predict = model1.predict(testX)   ### Using the same fitted model with trainX, trainY

### above, more appropriately test_predict is run after train_predict error is minimized or loss is minimized after the
### training set trainX, trainY are used in .fit -- and then the hyperparameters are tuned, before applying in a separate
### model1.fit to the testX, testY data. Above assumes the model1.fit on trainX, trainY is already the optimal one

### what is dense unit above: Dense implements the operation: output = activation(dot(input, kernel) + bias) where activation
### is the element-wise activation function passed as the activation -- Dense occurs at hidden and output layers
### By setting verbose 0, 1 or 2 you just say how do you want to 'see' the training progress for each epoch.
###    verbose=0 will show you nothing (silent)/ verbose=1 will show you an animated progress bar like this: progress_bar
###    verbose=2 will just mention the number of epoch like this, most detailed:
###         Epoch 30/30
###         661/661 - 1s - loss: 5.5611e-04 - 540ms/epoch - 817us/step
```

After iterating over 661 cases (packs of 10) each epoch, and then repeatedly over 30 epochs, the optimized .fit parameters produce the following root-mean-square-errors (RMSE) in the prediction of training data Y (trainY) and test data Y (testY) respectively:

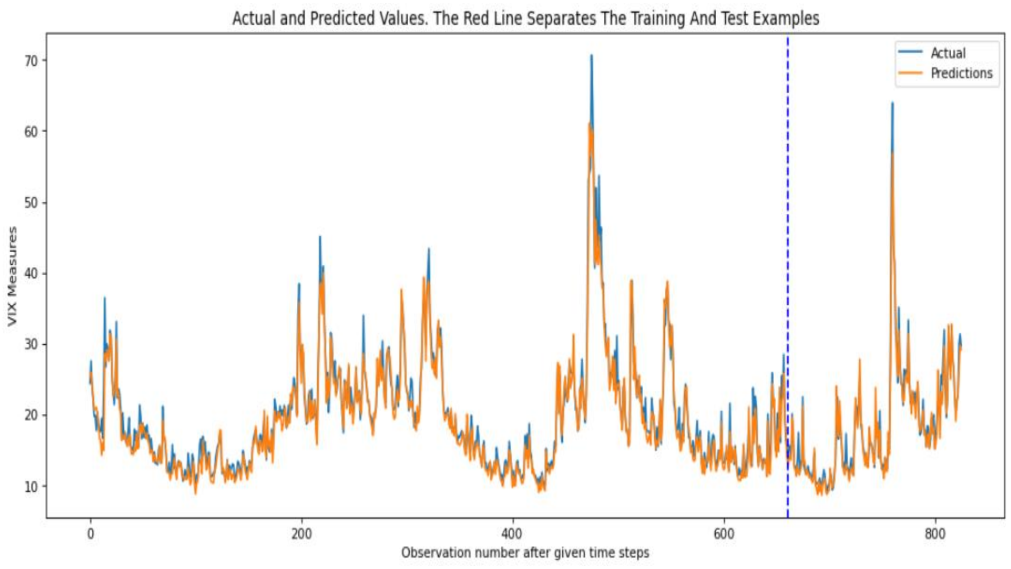


```
In [21]: def print_error(trainY, testY, train_predict, test_predict):
             ### Error of predictions
             train_rmse = math.sqrt(mean_squared_error(trainY, train_predict))
             test_rmse = math.sqrt(mean_squared_error(testY, test_predict))
             ### Print RMSE
             print('Train RMSE: %.3f RMSE' % (train_rmse))
             print('Test RMSE: %.3f RMSE' % (test_rmse))

         print_error(trainY, testY, train_predict, test_predict)

         Train RMSE: 0.023 RMSE
         Test RMSE: 0.024 RMSE
```

The RMSEs are about 2.3% for the training and 2.4% for the testing. This level of accuracy is not improved in the RNN for this study even if we perform overlapping data using all 6606 number of cases for training and increase the number of epochs to 100. The plot of the rescaled (inverse of scaling in [9] actual versus predicted outputs in both the training set and the test set are shown below.

```
In [24]: trainY1 = scaler.inverse_transform(trainY)
         testY1 = scaler.inverse_transform(testY)
         train_predict1 = scaler.inverse_transform(train_predict)
         test_predict1 = scaler.inverse_transform(test_predict)
```

```
In [25]: ### See also https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.axvline.html
         def plot_result(trainY1, testY1, train_predict1, test_predict1):
             actual = np.append(trainY1, testY1)
             predictions = np.append(train_predict1, test_predict1)
             rows = len(actual)
             plt.figure(figsize=(15, 6), dpi=80)
             plt.plot(range(rows), actual)
             plt.plot(range(rows), predictions)
             plt.axvline(x=len(trainY1), color='b', linestyle='--') ### note: vertical red line separates training part from test part
             plt.legend(['Actual', 'Predictions'])
             plt.xlabel('Observation number after given time steps')
             plt.ylabel('VIX Measures')
             plt.title('Actual and Predicted Values. The Red Line Separates The Training And Test Examples')
             ### Plot result -- this puts together Actual = (trainY,testY) or 187+46 points,
             ### and Predicted = (train_predict, test_predict)
         plot_result(trainY1, testY1, train_predict1, test_predict1)
```



8.3 Variants of RNN

As mentioned, the traditional RNN may run into vanishing gradient problem – when this happens, the effect of earlier inputs in the sequence becomes negligible, so the network does not learn from the long past – it has only short term memory with more recent inputs. To mitigate this problem, especially when long term memory is important, the Long Short Term Memory (LSTM) NN can be used.

In the LSTM architecture, the recurrent cell in Figure 8.3 is redesigned as follows in Figure 8.6. There is an additional state (also not explicitly observed) called the ‘cell state’. Like the hidden states $\mathbf{h}_t$ in a traditional RNN as in Figure 8.3 that stores ‘short-term’ memory of effects of past inputs, the cell state $\mathbf{c}_t$ stores ‘long-term’ memory of effects of past inputs. As in Figure 8.3, each “box” with each fresh input represents one hidden layer in the RNN.

Three gates are built into the cell to regulate or control flow strength of the data. The concatenated inputs (X_t,1) and the previous hidden states (outputs of last hidden layer) h_{t-1} are passed through (1) the Forget Gate before they flow through to update the last cell state c_{t-1} , (2) the Input Gate before they flow through to make the final update on the cell state that is preliminarily updated by (1), (3) the Output Gate before they update the hidden state.

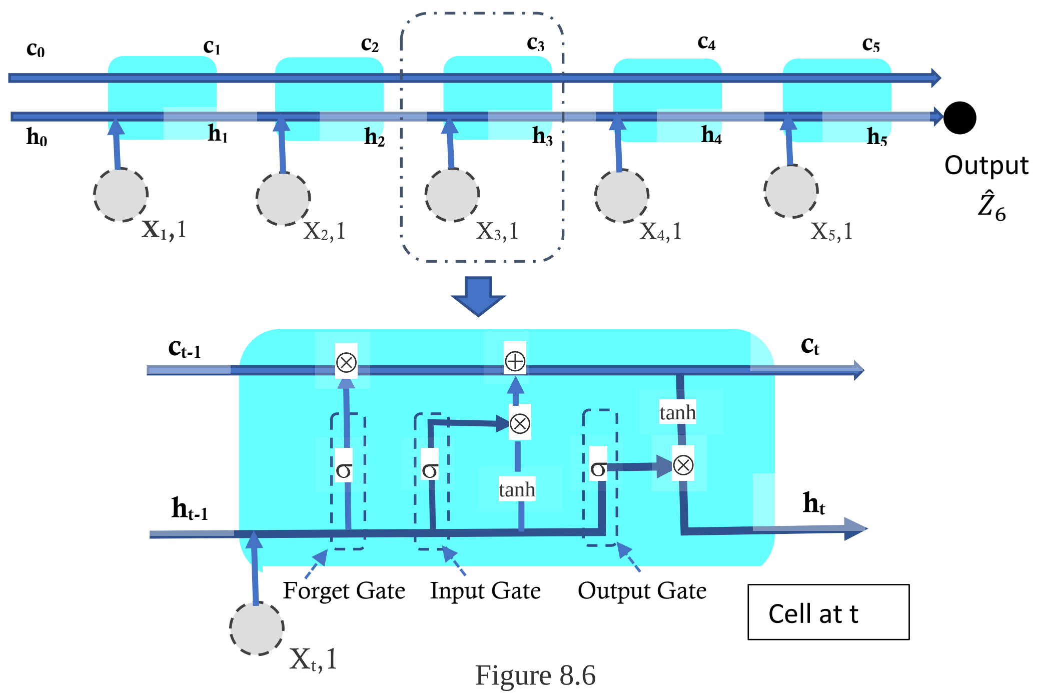


Figure 8.6

Suppose we create 50 neurons in the hidden layer or the cell. In (1), concatenated inputs (X_t,1, $\mathbf{h_{t-1}}$) after weighting is transformed by activation function (typically the sigmoid function, represented by the symbol σ in the diagram) on the weighting, viz.

$$\sigma\,(W_{FX}\,X_t + W_{Fh}\,\mathbf{h_{t-1}} + b_F\,)$$

where weights W_{FX} and W_{Fh} have appropriate dimensions to match the dimensions in X_t and $\mathbf{h_{t-1}}$. For example, at each time of input, X_t could be 1×1 and $\mathbf{h_{t-1}}$ could be 50 × 1 (typically the dimension of the hidden states is the same as the number of neurons). Then there would be 50 ×1 W_{FX} and 50 × 50 W_{Fh} and 50 × 1 b_F so that $\sigma(W_{FX}\,X_t + W_{Fh}\,\mathbf{h_{t-1}} + b_F)$ is a 50 × 1 output from the Forget Gate, each element being a sigmoid function of the respective linear weightings. There will typically be a similar dimension as the hidden states in the cell states, i.e., 50 × 1, or vector of $\mathbf{c_{t-1}}$. The output from the Forget Gate is a long-term retention that lies between 0 and 1 due to the nature of the sigmoid function. A value close to zero means that most of the past memory will be discarded. This Forget Gate output fraction vector $\sigma(W_{FX}\,X_t + W_{Fh}\,\mathbf{h_{t-1}} + b_F)$ is then multiplied by $\mathbf{c_{t-1}}$ (symbol ⊗ in Figure 8.6) using the Hadamard product or element to element multiplication. This provides the preliminarily updated cell state.

In (2), concatenated inputs (X_t,1, $\mathbf{h_{t-1}}$) after weighting $\sigma\,(W_{IX}\,X_t + W_{Ih}\,\mathbf{h_{t-1}} + b_I\,)$ is transformed by the tanh activation function, viz., $\tanh\,(W_{CX}\,X_t + W_{Ch}\,\mathbf{h_{t-1}} + b_C\,)$ as in traditional RNN. This provides a candidate for cell state update. However, this is further multiplied (Hadamard product ⊗) by a long-term retention factor as in (1), albeit with different weights. The outcome is then added (⊕) to the preliminarily updated cell state to obtain the final updated cell state c_t in this layer. Thus, this Input Gate identifies only important impacts from the concatenated inputs to be added to the long-term memory cell $\mathbf{c_t}$ where

$$\mathbf{c_t} = \sigma\,(W_{FX}\,X_t + W_{Fh}\,\mathbf{h_{t-1}} + b_F\,) \otimes \mathbf{c_{t-1}} + \sigma\,(W_{IX}\,X_t + W_{Ih}\,\mathbf{h_{t-1}} + b_I\,) \otimes \tanh\,(W_{CX}\,X_t + W_{Ch}\,\mathbf{h_{t-1}} + b_C\,).$$

In (3), the tanh activation on the final updated cell state c_t is then multiplied (Hadamard product ⊗) by a long-term retention factor as in (1) and (2), albeit with different weights in the Output Gate to yield the next hidden state $\mathbf{h_t}$ that flows into the next LSTM cell, viz.

$$\mathbf{h_t} = \sigma\,(W_{OX}\,X_t + W_{Oh}\,\mathbf{h_{t-1}} + b_O\,) \otimes \tanh\,(\mathbf{c_t}).$$

In a many-to-one structure as in Figure 8.6, the final output could be a scalar predicted number provided by the output layer comprising 51 weights (dimension of output $\mathbf{h_t}$ and the bias), viz. $W_{Yh}\,\mathbf{h_t} + b_Y$.

If in this LSTM NN with one hidden layer in each input point, there are 50 neurons (Keras App names it as number of units in the hidden layer/cell), then there is a total of 4 × [(50 + 1) × 50 + 50] parameters in the first hidden layer. In general, for N neurons or units in the first hidden layer of LSTM, and J number of inputs/features, there are 4 × [(N +J) × N + N] parameters to be optimized, where the (N+J) denotes the number of previous hidden state and feature inputs. This is multiplied into N, the number of neurons in the hidden layer. As said, the number of hidden states is typically structured to be the same as the number of neurons in the layer. The remaining N in the [ ] denotes the N biases, one for each neuron in the hidden layer. Thus for the 50 neurons (units) single layer LSTM, there are 4 × [(50 + 1) × 50 + 50] = 10,400 parameters.

However, single hidden layer (even with a large number of neurons per layer or width) and many epochs may not be able to perform good training results. A deeper version is the stacked LSTM model with more hidden layers (that are stacked up at an input time-sequence). The stacked LSTM with two hidden layers at each time-sequence is shown below in Figure 8.7. Superscripts to hidden states h_t and cell states c_t denote the first hidden layer 1 at t and the second (stacked) layer at t. The key idea is that the hidden state output h_t^1 at t in level 1 passes up to the second level layer as input together with the hidden state input h_{t-1}^2 at level 2. Note h_{t-1}^2 (level 2 hidden state at t-1) needs not be the same as h_{t-1}^1, the level 1 hidden state at t-1. The last output layer, if any, at t uses the level 2 cell state c_t^2 and the level 2 hidden state h_t^2, i.e. $W_{Yh}\,\mathbf{h_t}^2 + b_Y$.

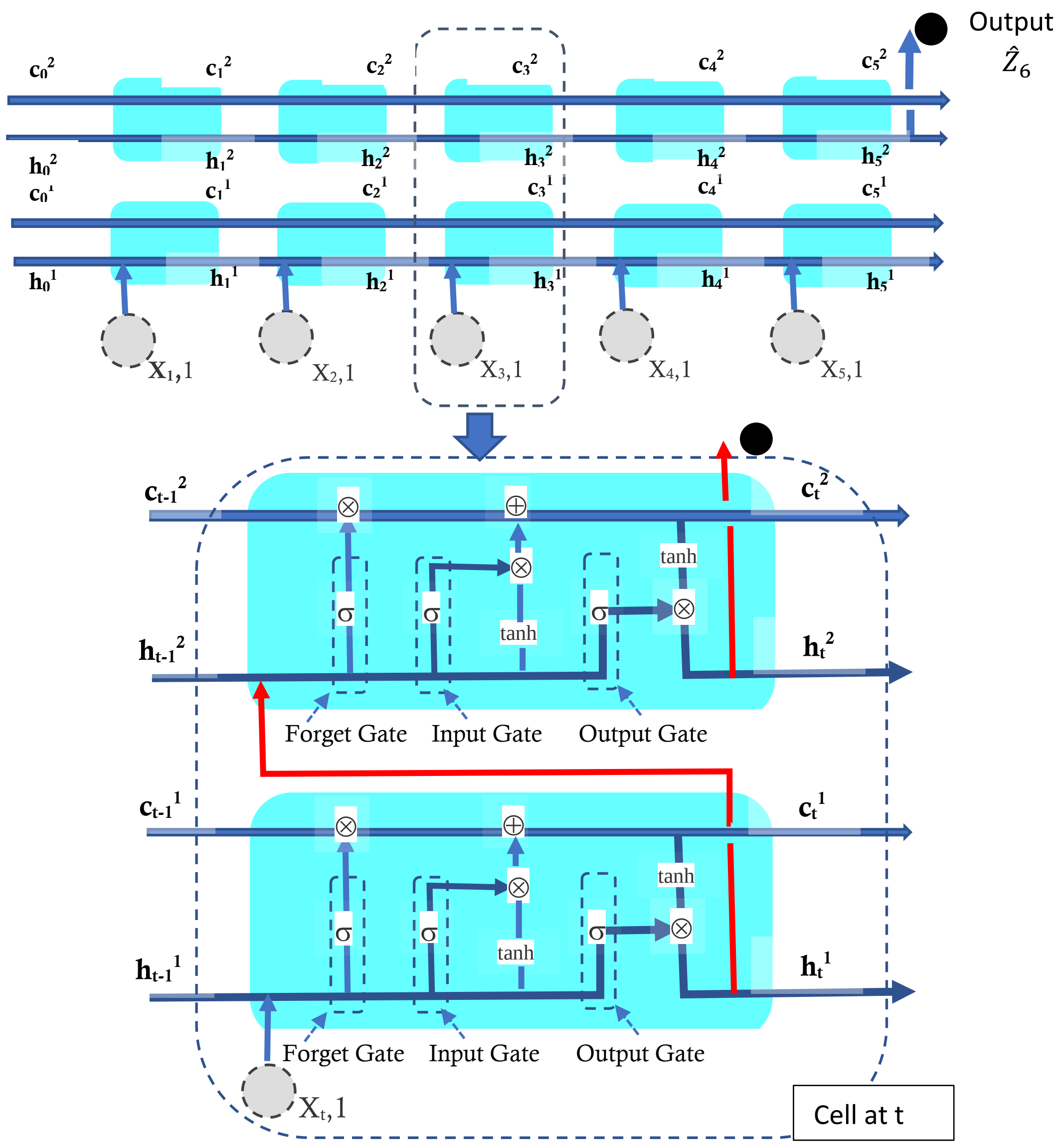


Figure 8.7

Suppose we have similarly 50 neurons in each LSTM cell – 50 neurons in the first level cell and 50 neurons in the second level cell of the same cell at t, as in Figure 8.7. The Output Gate from first level cell provides 50 hidden states at level 1, viz.

$$\mathbf{h}_t^{\mathbf{1}} = \sigma\,(W_{OX}^{1}\,X_t + W_{Oh}^{1}\,\mathbf{h}_{t\text{-}1} + b_O^{1}\,) \otimes \tanh\,(\mathbf{c}_t^{\mathbf{1}}).$$

where the superscripts to the parameters denote association with the cell level. Together with $\mathbf{h}_{t\text{-}1}^{\mathbf{2}}$, these are inputs to the Forget Gate, Input Gate, Candidate for Cell State Update, and the Output Gate. Since N = 50, and dimension of $\mathbf{h}_{t\text{-}1}^{\mathbf{2}}$ (same dimension as $\mathbf{h}_{t\text{-}1}^{\mathbf{1}}$) and of $\mathbf{h}_t^{\mathbf{1}}$ are both N, then (N + N) are connected to N, the number of neurons in the stacked second hidden layer. At the same time, there are N biases added, so there are ($N^2 + N^2 + N$) number of parameters at the Forget Gate. Then, the Forget Gate output fraction vector $\sigma(W_{FX}^{2}\,\mathbf{h}_t^{\mathbf{1}} + W_{Fh}^{2}\,\mathbf{h}_{t\text{-}1}^{\mathbf{2}} + b_F^{2})$ is then multiplied by $\mathbf{c}_{t\text{-}1}^{\mathbf{2}}$ using the Hadamard product or element to element multiplication, where W_{FX}^{2} is 50 × 50, W_{Fh}^{2} is 50 × 50, and b_F^{2} is 50 × 1. This provides the preliminarily updated cell state at level 2 at t. The same operations as in level 1 occur at the Input Gate, the candidate for cell update, and the Output Gate. Hence in total there are $4 \times [(N + N) \times N + N]$ parameters at the second level LSTM cell/stacked layer. The hidden state output from the second level cell and also the output from this stacked LSTM cell that goes into the next stacked LSTM cell with fresh feature inputs is $\mathbf{h}_t^{\mathbf{2}} = \sigma\,(W_{OX}^{2}\,\mathbf{h}_t^{\mathbf{1}} + W_{Oh}^{2}\,\mathbf{h}_{t\text{-}1}^{\mathbf{2}} + b_O^{2}\,) \otimes \tanh\,(\mathbf{c}_t^{\mathbf{2}})$.

8.4 Worked Example II – Data

In this second worked example, we show how a LSTM RNN can be used to predict Google stock prices. Historical data are collected from Yahoo Finance consisting of daily (distribution and split) adjusted closing Google stock prices from 3 Jan 2012 till 29 Dec 2016, with a total of 1257 sample points. See demonstration file Chapter8-2.ipynb.

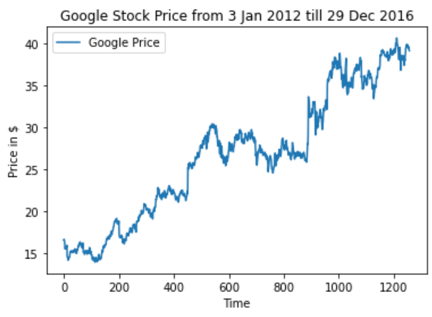
```
In [2]: ### import training set
        dataset=pd.read_csv('GOOG.csv')
        dataset.head()
```

Out[2]:

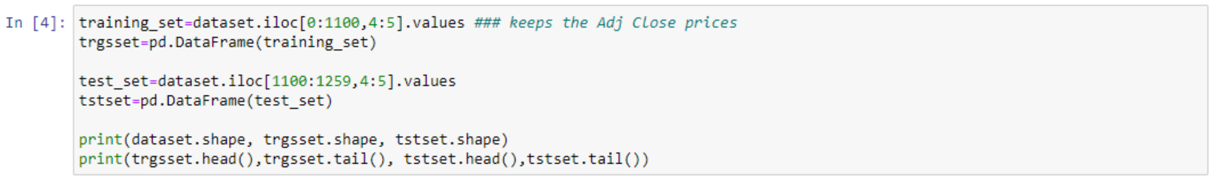
	Date	Open	High	Low	Close	Adj Close	Volume
0	3/1/2012	16.262545	16.641375	16.248346	16.573130	16.573130	147611217
1	4/1/2012	16.563665	16.693678	16.453827	16.644611	16.644611	114989399
2	5/1/2012	16.491436	16.537264	16.344486	16.413727	16.413727	131808205
3	6/1/2012	16.417213	16.438385	16.184088	16.189817	16.189817	108119746
4	9/1/2012	16.102144	16.114599	15.472754	15.503389	15.503389	233776981

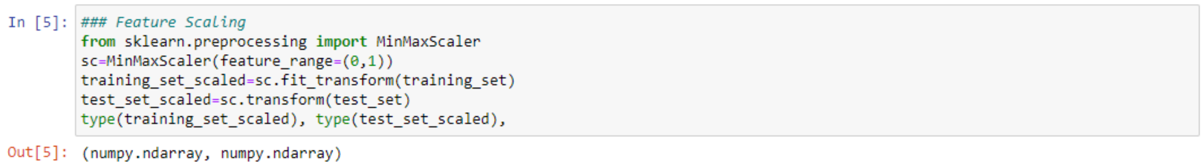
The graph of the price data series is shown as follows.

```
In [3]: p=dataset.iloc[0:1259,4:5]
        plt.plot(p,label="Google Price",linestyle='-')
        plt.title('Google Stock Price from 3 Jan 2012 till 29 Dec 2016')
        plt.xlabel('Time')
        plt.ylabel('Price in $')
        plt.legend()
        plt.show()
```

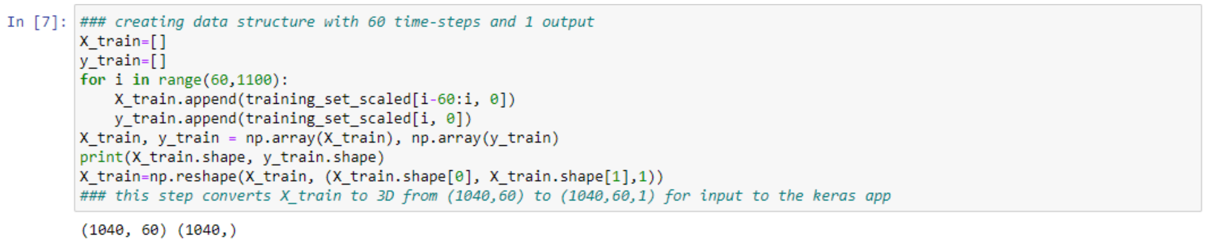


These are split into the front-end 1100 sample points as training set, and the back-end 157 sample points as test set. These are then scaled using Sklearn App MinMaxScaler to scale them to within (0,1), i.e., X is transformed to (X – min)/(max – min), where min, max refer to the minimum and maximum of the training set feature associated with X. The test set data are scaled using the training set scaling parameters of min, max, so there may be a few points that may slightly exceed 1 or negative if the test set has data that fall below the min in the training set.

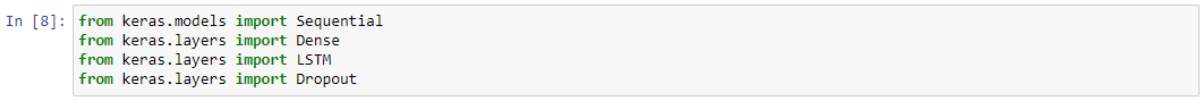

```
In [4]: training_set=dataset.iloc[0:1100,4:5].values ### keeps the Adj Close prices
        trgsset=pd.DataFrame(training_set)

        test_set=dataset.iloc[1100:1259,4:5].values
        tstset=pd.DataFrame(test_set)

        print(dataset.shape, trgsset.shape, tstset.shape)
        print(trgsset.head(),trgsset.tail(), tstset.head(),tstset.tail())
```

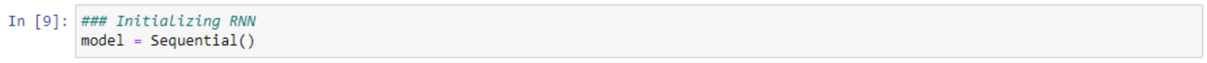

```
In [5]: ### Feature Scaling
        from sklearn.preprocessing import MinMaxScaler
        sc=MinMaxScaler(feature_range=(0,1))
        training_set_scaled=sc.fit_transform(training_set)
        test_set_scaled=sc.transform(test_set)
        type(training_set_scaled), type(test_set_scaled),

Out[5]: (numpy.ndarray, numpy.ndarray)
```

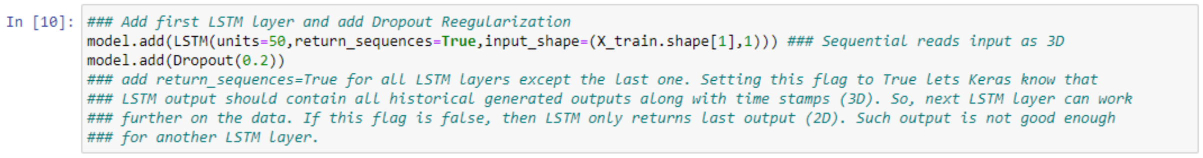
Next, the data structure is created to feed the inputs to the NN. As in Figure 8.5, T = 1100 for the training data set. The first 60 data points are used for training data X or X_train while the next data point (point 61) is used as training data Y or Y_train. The concatenated X_train is reshaped into 1040 rows (first dimension size) each with 60 columns (second dimension size) of timed inputs each.


```
In [7]: ### creating data structure with 60 time-steps and 1 output
        X_train=[]
        y_train=[]
        for i in range(60,1100):
            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)
        print(X_train.shape, y_train.shape)
        X_train=np.reshape(X_train, (X_train.shape[0], X_train.shape[1],1))
        ### this step converts X_train to 3D from (1040,60) to (1040,60,1) for input to the keras app

        (1040, 60) (1040,)
```

As shown below, stacked LSTM NN is used with four stacked LSTM cells at each timed input for each case of 60 inputs followed by prediction of the output at the end of 60 time-sequenced inputs. There are 1040 cases as we use overlapping cases here. Each stacked LSTM cell contains 50 neurons. The output layer specified one neuron, so there is only one predicted scalar output. Notice that activation functions throughout are built into the Keras LSTM App as discussed in the last section. The Recurrent Dropout is a regularization tool applied to reduce overfitting or over adjustments of the cell state and hence over-effect on the final outputs. If the Dropout rate is set to say 0.20, then the Dropout layer randomly sets input units to the cell state and to the input/update state to 0 with a frequency of 20% at each step during training time. Inputs that are not set to 0 are scaled up by 1/(1 – Dropout rate) so that the sum of all inputs remains the same.


```
In [8]: from keras.models import Sequential
        from keras.layers import Dense
        from keras.layers import LSTM
        from keras.layers import Dropout
```


```
In [9]: ### Initializing RNN
        model = Sequential()
```


```
In [10]: ### Add first LSTM layer and add Dropout Reegularization
         model.add(LSTM(units=50,return_sequences=True,input_shape=(X_train.shape[1],1))) ### Sequential reads input as 3D
         model.add(Dropout(0.2))
         ### add return_sequences=True for all LSTM layers except the last one. Setting this flag to True lets Keras know that
         ### LSTM output should contain all historical generated outputs along with time stamps (3D). So, next LSTM layer can work
         ### further on the data. If this flag is false, then LSTM only returns last output (2D). Such output is not good enough
         ### for another LSTM layer.
```

```
In [11]: ### Add second LSTM layer and Dropout
         model.add(LSTM(units=50,return_sequences=True))
         model.add(Dropout(0.2))
```

```
In [12]: ### Add third LSTM layer and Dropout
         model.add(LSTM(units=50,return_sequences=True))
         model.add(Dropout(0.2))
```

```
In [13]: ### Add fourth LSTM layer and Dropout
         model.add(LSTM(units=50)) ### note: last LSTM layer does not carry argument 'return_sequences=True'
         model.add(Dropout(0.2))
```

```
In [14]: ### Add output layer
         model.add(Dense(units=1))  ### not capital "U"nit
```

```
In [15]: ### Compiling the RNN
         model.compile(optimizer='adam',loss='mean_squared_error')
```

In codeline [15], we employ optimizer 'Adam' for the optimal adjustment toward loss function minimum in the back propagation, and mean squared error MSE as the loss function to be minimized. Code line [16] below performs the fitting or optimization of loss function based on the built stacked LSTM model. The batch size = 10, which means that 10 cases are grouped together into one batch for parameter update each time, so there are 1040/10 = 104 batches. Updating through the 104 batches is one epoch. Repeating the updates involve more epochs – we use 100 epochs.

```
In [16]: ### Run the training set with the LSTM (specialized RNN here)
         model.fit(X_train,y_train,epochs=100,batch_size=10)

         ...............
         Epoch 99/100
         104/104 [==============================] - 2s 22ms/step - loss: 0.0013
         Epoch 100/100
         104/104 [==============================] - 2s 22ms/step - loss: 0.0013
Out[16]: <keras.callbacks.History at 0x11afd977040>
```

```
In [17]: model.summary()

         Model: "sequential"
         _________________________________________________________________
          Layer (type)                Output Shape              Param #
         =================================================================
          lstm (LSTM)                 (None, 60, 50)            10400

          dropout (Dropout)           (None, 60, 50)            0

          lstm_1 (LSTM)               (None, 60, 50)            20200

          dropout_1 (Dropout)         (None, 60, 50)            0

          lstm_2 (LSTM)               (None, 60, 50)            20200

          dropout_2 (Dropout)         (None, 60, 50)            0

          lstm_3 (LSTM)               (None, 50)                20200

          dropout_3 (Dropout)         (None, 50)                0

          dense (Dense)               (None, 1)                 51

         =================================================================
         Total params: 71,051
         Trainable params: 71,051
         Non-trainable params: 0
         _________________________________________________________________
```

Next we perform the prediction (getting the LSTM output layer output) by feeding in the feature data in X_train. See code line [18]. As the output 'predict_train' is multiple array format, we use code line [20] to retrieve the column containing the 1040 predicted outputs with each case.

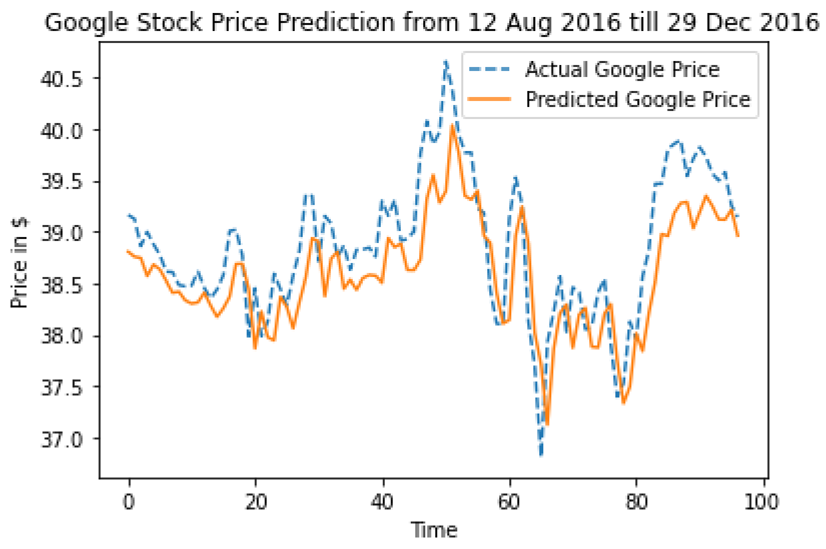
```
In [18]: predict_train=model.predict(X_train)
         print(predict_train.shape)
         ### this output is (1040,60,1), we want only the first no. in each row of 1040
         ### this 3D structure makes it more difficult to interpret comparison of prediction in training set vs output in trg set

         33/33 [==============================] - 1s 12ms/step
         (1040, 1)
```

```
In [19]: print(predict_train)

         [[0.11048877]
          [0.11068454]
          [0.10711372]
          ...
          [0.87018967]
          [0.8678323 ]
          [0.87677455]]

In [20]: predict_train=predict_train[:, 0]  ### select first column or first day of 60 days of each day prediction from 1 to 1040

In [21]: print(predict_train)

         [0.11048877 0.11068454 0.10711372 ... 0.87018967 0.8678323  0.87677455]
```

By this step, the model is trained using the training set data. We now structure the test set data to perform prediction using the built/trained stacked LSTM model. Inputs to the model from the test set are in X_test and the prediction is compared with actual y_test using the MSE loss function.

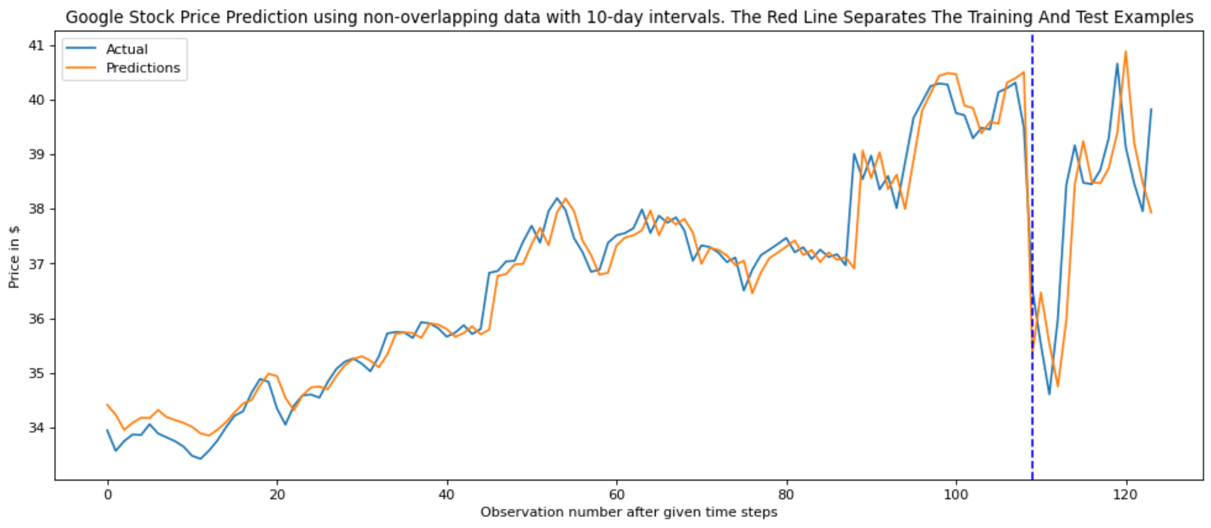
```
In [22]: ### creating data structure with 60 time-steps and 1 output
         X_test=[]
         y_test=[]  ### here y_test is to collect the predicted y values
         for i in range(60,157):
             X_test.append(test_set_scaled[i-60:i, 0])
             y_test.append(test_set_scaled[i, 0])
         X_test, y_test = np.array(X_test), np.array(y_test)  ### convert to np arrays as X_test is "list"
         X_test=np.reshape(X_test, (X_test.shape[0], X_test.shape[1],1))

In [23]: y_test.shape ### now (97.) is 1 Dim -- convert to two dim

Out[23]: (97,)

In [24]: y_test = np.reshape(y_test, (-1, 1))
         y_train=np.reshape(y_train,(-1,1))

In [25]: ### Prediction
         predicted_stock_price=model.predict(X_test)
         predicted_stock_price=predicted_stock_price[:,0]
         #predicted_stock_price1=sc.inverse_transform(predicted_stock_price)
         #predicted_stock_price.shape

In [27]: from sklearn.metrics import mean_squared_error
         import math
         def print_error(trainY, testY, train_predict, test_predict):
             ### Error of predictions
             train_rmse = math.sqrt(mean_squared_error(trainY, train_predict))
             test_rmse = math.sqrt(mean_squared_error(testY, test_predict))
             ### Print RMSE
             print('Train RMSE: %.3f RMSE' % (train_rmse))
             print('Test RMSE: %.3f RMSE' % (test_rmse))

         print_error(y_train, y_test, predict_train, predicted_stock_price)

         Train RMSE: 0.020 RMSE
         Test RMSE: 0.019 RMSE
```

The RMSE for the training set fitting and then the test set prediction are about 2.0 % and 1.9% respectively. The rescaled time series of the actual Google price from 12 Aug 2016 till 29 Dec 2016 is compared with the predicted price based on the LSTM model above – this is shown in the following matplotlib.pyplot graph. It is seen that the predicted price tended to follow/lag the actual price.



If in the above example, non-overlapping cases are used, i.e., we use time-steps = 10, so every 10 daily inputs gives rise to one predicted price at the 11th day. The next case has daily inputs from day 11 to day 20 with output on day 21, and so on. See demonstration file Chapter8-3.ipynb. Then we have a total of 109 predicted prices in the training set to be matched with the actual prices in the training set, and a total of 15 predicted prices in the test set to be matched with the actual price in the test set. Number of neurons in similar 4 stacked LSTM layers/cells is 50.

The RMSE for the training set fitting and then the test set prediction are about 5.5% and 16.4% respectively. The rescaled time series of the actual Google price from 12 Aug 2016 till 29 Dec 2016 is compared with the predicted price based on the LSTM model above – this is shown in the following matplotlib.pyplot graph. It is seen that the predicted price tended to follow/lag the actual price.

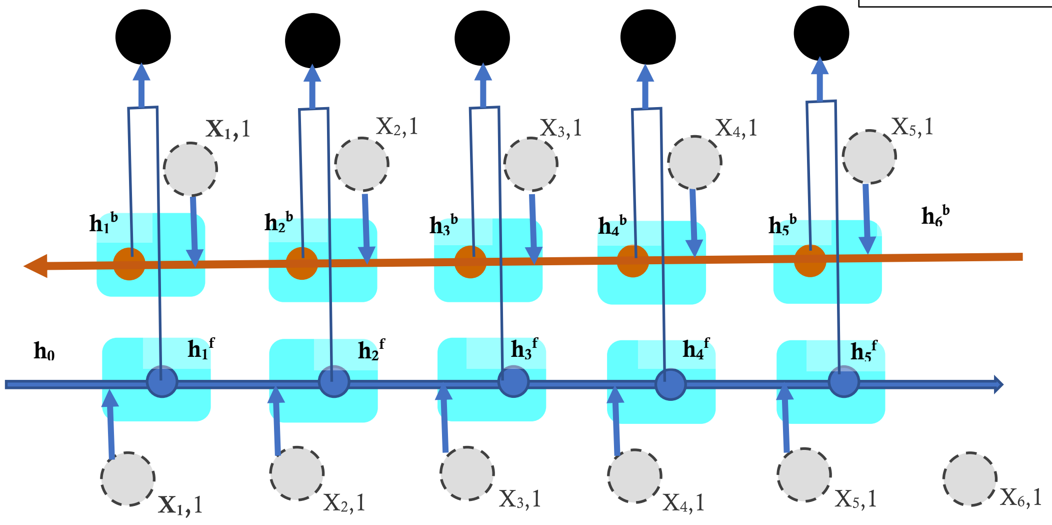


Bidirectional

As mentioned, the traditional RNN can be improved by taking into consideration future inputs. This is not as common in the financial context as prediction of the future with some future information either means that the future events are pre-committed, e.g., a mandatory or committed public policy to be effected, or there is private information on future events so that the private investor can use that to help the prediction. However, in other AI applications involving physical or else sensory situations, future items in a sequence can be naturally helpful in prediction, e.g., predicting missing words in a sentence when later parts of the sentence can help with a fuller context, predicting or language translation when the entire sequence of words including later words would help to make sense, and predicting or recognizing handwriting when the last sequence of strokes in pixel inputs are considered.

To enable past, current, and future inputs, Bidirectional RNNs, or BRNNs, are used. A BRNN is a combination of two RNNs (also LSTMs) - one RNN (LSTM) moving forward from the start of the data sequence, and the other moving backward from the end of the data sequence.

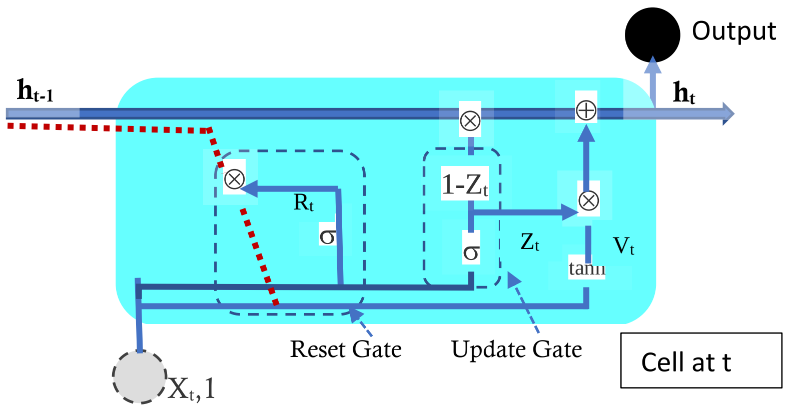
$$\text{Output} = W_{Ox}{}^{f} h_t{}^{f} + W_{Ox}{}^{b} h_t{}^{b} + b_O$$



The usual forward hidden state updates are $\mathbf{h_t}^{\mathbf{f}}$ based on inputs $(X_t,1)$ and past hidden state $\mathbf{h_{t-1}}^{\mathbf{f}}$. The additional backward hidden state updates are now $\mathbf{h_t}^{\mathbf{b}}$ based on inputs $(X_t,1)$ and future hidden state $\mathbf{h_{t+1}}^{\mathbf{b}}$. Their outputs are concatenated for final output. The diagram above shows a many-to-many structure where each input is related to one output. If it is a many-to-one structure as in Figure 8.3, then the forward and the backward updating should be synchronized to yield the output at the same time point.

Gated Recurrent Unit

Just like the LSTM, the gated recurrent unit (GRU) is a structure to retain memory so that the gradient (hence parameter updates) will not disappear. The GRU structure is seen as follows. This is a fully gated unit; there are other variations that are simpler.



GRU has only two, not three gates as in LSTM. It has only an Update Gate and a Reset Gate, and does not have the Output Gate in a LSTM NN.

In the Reset Gate, the output R_t is given by $R_t = \sigma (W_{RX} X_t + W_R \mathbf{h_{t-1}} + b_R)$. This R_t is multiplied with $\mathbf{h_{t-1}}$ in a Hadamard product (see dotted line) and then activated using tanh function to create a candidate activation vector, $\mathbf{V_t}$ **.** R_t is between 0 and 1 and could be close to zero, implying small V_t , where

$V_t = \tanh (W_{VX} X_t + W_V (R_t \otimes \mathbf{h_{t-1}}) + b_V)$.

The Reset Gate decides how much of the past information in $\mathbf{h_{t-1}}$ is to be neglected or if the hidden state is important or not.

In the Update Gate, the output Z_t is given by $Z_t = \sigma (W_{UX} X_t + W_Z \mathbf{h_{t-1}} + b_Z)$. The output hidden state $\mathbf{h_t} = Z_t \otimes V_t + (1- Z_t) \otimes \mathbf{h_{t-1}}$ **.** The Update Gate determines the amount of past information that would matter in the next hidden state.

A detailed explanation of an alternative variation – the Gated Recurrent Unit (GRU) can be found in https://towardsdatascience.com/ understanding-gru-networks-2ef37df6c9be.

Convolution Neural Network (CNN)

A detailed explanation of the Convolution Neural Network (CNN) can be found in https://towardsdata science.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

8.5 Summary

There are good predictions (1) and there are good predictions (2). Price chasing (1) using just a time series trained on past prices could be a good machine predictor but may be a bad idea for trading. In the latter, an alternative is to enable better trading if based on momentum trading – AR process – faster computing time than a complex NN, the few microseconds could mean gain or loss.

Big caveats when one proceeds to transport a predictive machine such as the Google price prediction to something economic/downright market finance such as trading – can you make a steady/consistent profit? The problems are as follows – any apparent profits can be decimated or vanquished by transaction fees/costs, impact costs/widening bid-ask spread if more want to buy (ask goes up), if more want to sell (bid goes down), liquidity risk – when it is not possible to trade as frequently as in the training/testing using past data, slippages in market order – getting higher purchase price or lower selling price for your order (effect same as in execution risk when latency - time delay between order and actual trade – is slower than several microseconds, 1/1,000,000 second).

A trend following prediction (that happens often in minimizing MSE) not considering above costs may do well in long runs of up or long runs of down – but will lose heavily in sharp market turnaround.

More meaningful ML models for trading (2) could include training based not just on past prices, but using market microstructure features in high frequency trading (HFT) such as queue order level 2 quotes (market depth) or even level 3 (who are the buyers/sellers in the queue), the momentum – e.g. average price movements in the last few intervals, bid-ask spread, volume, a market-sentiment measure, firm size, P/E, P/B ratios, profit news, competitor prices, industry price, market movements, interest rates, etc.

8.6 Other References

https://www.simplilearn.com/tutorials/deep-learning-tutorial
https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks
https://towardsdatascience.com/recurrent-neural-networks-explained-with-a-real-life-example-and-python-code-e8403a45f5de
https://keras.io/api/layers/recurrent_layers/lstm/
https://d2l.ai/chapter_recurrent-modern/lstm.html
https://medium.com/deep-learning-with-keras/lstm-understanding-output-types-e93d2fb57c77
https://keras.io/api/layers/regularization_layers/dropout/

https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/
https://towardsdatascience.com/gate-recurrent-units-explained-using-matrices-part-1-3c781469fc18
https://d2l.ai/chapter_recurrent-modern/bi-rnn.html