8 ARTIFICAL NEURAL NETWORK II

In this chapter we discuss a major type of neural network especially for financial analyses with time series – the recurrent neural network (RNN). This involves a more 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), the Gated Recurrent Unit (GRU), and the convolutional network (CNN).

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 revised via backpropagation to minimize the error/loss function L(.) that combines all cases $\sum_{k=1}^{N} \frac{1}{N} L(Y_k, \hat{Z}_k)$ 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. For each forward and then backward propagation (one iteration) when weights and biases of the ANN are revised before the next forward propagation to recompute the new prediction, these same weights and biases are applied to every case k in order to arrive at the loss function $\sum_{k=1}^{N} \frac{1}{N} L(Y_k, \hat{Z}_k)$. These computations in a single iteration can be done in parallel for each case k since the ANN is the same for each k.

In Figure 8.1, we show the parallel multiprocessing using N identical multilayer perceptrons (MLP), each to forward-propagate each case/subject or sample point k in the training data set of size N or N sample points.

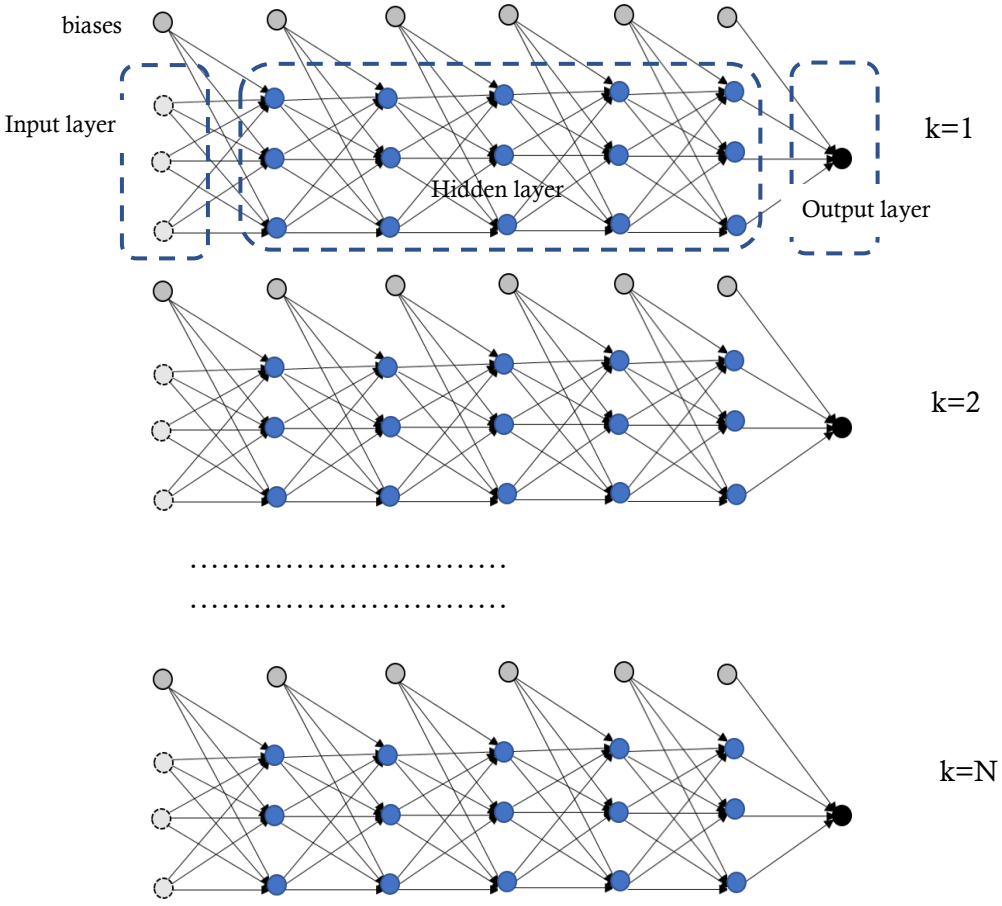


Figure 8.1

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).

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. The final iteration set of parameters as common to all k. In the parallel set of N MLPs shown in Figure 8.1, each iteration means all the N MLPs are updated simultaneously with the new set of revised weights and biases. This trained set of parameters is then applied to predict the test data cases j.

8.1 Recurrent Neural Network (RNN)

Suppose now the input features for each case, instead of given altogether at the initial input layer, are given one at a time in a sequence. In a common setup to this new way of input, the number of hidden layers is now equal to the number of sequential input features so that at each additional hidden layer, there is a fresh sequential input feature – see dotted arrow in Figure 8.2. We consider the simple case where each sequential input feature is a scalar number. More general cases include an input feature which is a vector of numbers. This new architecture is common in time series prediction such as predicting the next day stock price (label or target) using lagged prices in the past 5 days. These lagged prices form the features of this label. Each of these lagged prices becomes one feature in this one training case. A different window of price with lagged 5 prices forms another case, and so on.

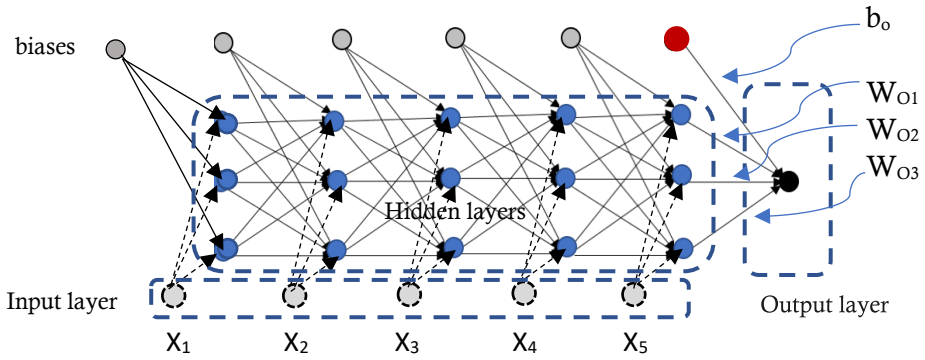


Figure 8.2

Figure 8.2 shows an architecture of 3 neurons per hidden layer whereby one training case is passed through the NN. We call this architecture Recurrent Neural Network. The inputs for this case consist of 5 sequential features X_1, X_2, X_3, X_4, and X_5. There is one output for the case in the output layer of one neuron. As in ANN or MLP, many cases in a batch or mini batch are required to pass through this RNN before the loss function, comprising sum of loss in each case, is to be reduced via revising the parameters in the model that include all the weights and biases. The following Figure 8.3 shows the labelling convention we adopt for the RNN.

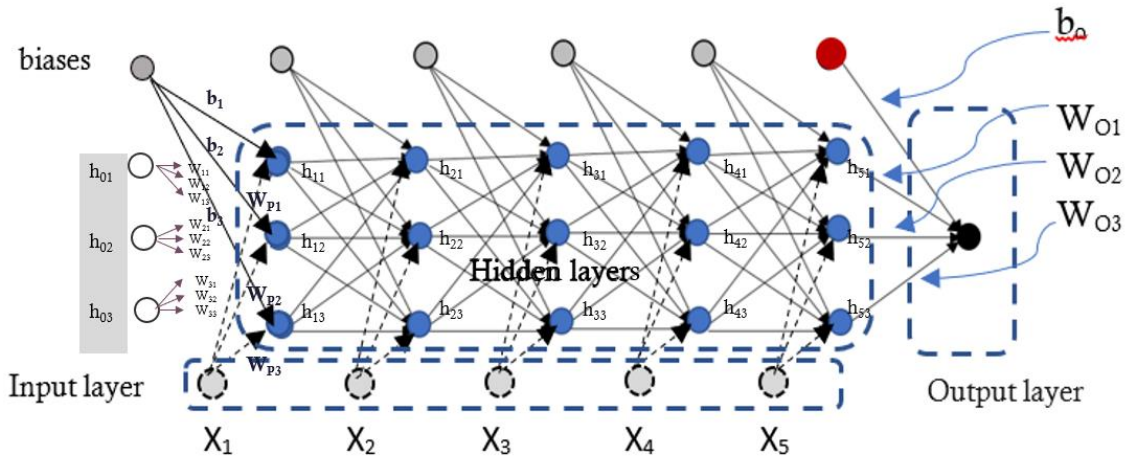


Figure 8.3

In RNN, the weights and biases at each neuron level do not change from one hidden layer to another. This is unlike that in the ANN. Consider the first hidden layer in this RNN in Figure 8.3. Each of the 3 neurons in this layer receives 5 inputs and thus 5 associated weights/parameters in the edges/synapses. In the first neuron on the first (top) level, weight on the input feature X_1 is W_{P1} where "P" stands for "input". Weights from the prior layer of 3 neurons are W_{11}, W_{21}, and W_{31} respectively where subscripts pq in W_{pq} is such that p represents the level position of the previous output neuron and q represents the level position of the current neuron. Hence weights from prior layer of neurons to the second level neuron are W_{12}, W_{22}, and W_{32} respectively. Weights from prior layer of neurons to the third level neuron are W_{13}, W_{23}, and W_{33} respectively. Bias or weight on value 1 to the top-level neuron is b_1, to the second level neuron is b_2, and to the third level neuron is b_3. So far there are 3 input weights (W_{P1}, W_{P2}, and W_{P3}), 3×3 weights W_{pq} from prior layer of neurons, and 3 biases. In addition, there are the weights W_{o1}, W_{o2}, W_{o3} and b_0 representing weights and bias from last hidden layer to the output neuron. Hence in total there are 3+3×3+3+(3+1) or (3+2)×3+(3+1) or (3+2)×(3+1) –1 = 19 parameters to be fitted in this RNN. In general, when the hidden layer has m neurons, then the total number of parameters to be fitted is (m+2)(m+1) – 1. This number is far less than that in ANN when each hidden layer has a different set of parameters indexed by superscript [n] indicating n^{th} hidden layer.

As another example, consider the second hidden layer in the NN in Figure 8.3. 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. At the output layer, there are the bias parameter b_0, and the 3 weight parameters W_{O1}, W_{O2}, and W_{O3}.

In Figure 8.3, the output from the first neuron in the first hidden layer is

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

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. In subsequent iterative computations based on revised weights, these values (h_{01}, h_{02}, h_{03}) will take on values (h_{51}, h_{52}, h_{53}) of the last hidden layer neurons based on each case/subject.

The output as the hidden state h_{11} will be passed forward 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

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

The hidden states (or output from last hidden layer neurons) are h_{51}, h_{52}, and h_{53}. Therefore, the output is $\tanh(W_{O1} \times h_{51} + W_{O2}\, h_{52} + W_{O3}\, h_{53} + b_o)$ where the subscript "O" denotes "output".

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, h_{tj}, contains information of lagged exogenous input features X_{t-1}, X_{t-2}, X_{t-3}, and so on. This type of architecture is important if indeed lagged input features also have useful information in predicting the output. This type of neural network that allows for recurrence of influences of past features is thus called Recurrent Neural Network (RNN).

The RNN of Figure 8.3 may be represented in a concise manner as follows. Each box represents a hidden layer of neurons.

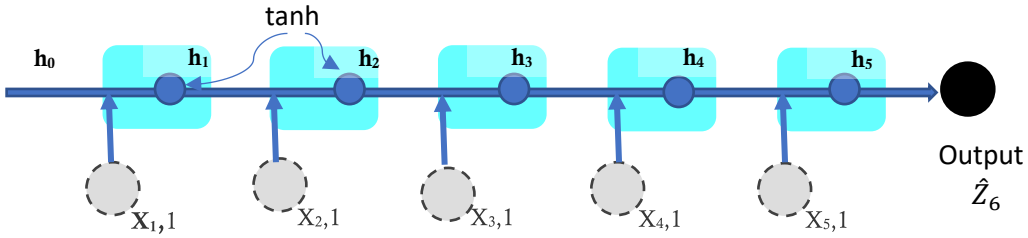


Figure 8.4

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 8.5 by compacting hidden layers and sequential inputs. In this diagram, the recurrence of the hidden layers of inputs is made more obvious.

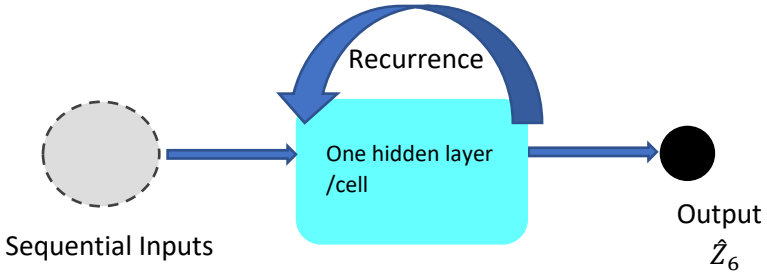


Figure 8.5

Figures 8.3, 8.4 are sometimes called the unfolded NN of the above diagram. In Figure 8.4, 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 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 t, viz., X_{t-1}, X_{t-2}, X_{t-3}, and so on, may not be independent. For example, the vector X_t may be autocorrelated with X_{t-1} and so on, e.g., 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_{t-j}, 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, besides current input X_t, more likely also affect predicted output $\hat{Y}_{t+1}$. 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, ChatGPT, and so on.

In the training of the RNN shown in Figures 8.3, 8.4, 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. In general, a training data set could have a time series or a sequence of T data points. The T sample points are divided into cases or packs each with R, e.g., 5, number of sequential features or recurrent cells/hidden layers. The number of cases or packs is then T/R.

A certain number of cases/packs are grouped into mini batches with batch size S or S number of cases in each mini batch. So, there are T/(RS) number of mini batches (each with size S). Each case in a mini batch yields one set of partial derivatives of the loss function with respect to the (m+2) (m+1) – 1 number of parameters. While the sequence within each case or pack clearly considers dependence and recurrence, the different cases or packs are treated as independent. Thus, S number of cases in each mini batch yield S sets of partial derivatives of the loss function. The aggregate loss function is the sum of losses of the S number of cases in each batch. Thus, over each mini batch, the forward propagation and then backward propagation in updating the parameters based on the sum of S partial derivatives is considered one iteration.

The training sample cases can be run in parallel for each iteration. Over E number of epochs and T/(RS) number of mini batches, there will be E × T/(RS) number of iterations to update/revise the parameters. Just as in MLP, when S = 1, it is similar to the stochastic gradient descent method. The Adam (with changing learning rate) can also be applied to adjust the parameters to minimize the loss function via iterations.

Consider the loss function of one case, e.g., $L(Y_{t+R}, \hat{Y}_{t+R}) = (\ Y_{t+R} - \hat{Y}_{t+R})^2$ where $\hat{Y}_{t+R} \equiv \hat{Z}_{t+R}$, R > 0, 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},…, W_{Om}, b_o, occur at the output layer.

The partial derivative with respect to each of the output layer parameter is, where m=3, R=5, and $\hat{Y}_{t+6} = \tanh(W_{O1} \times h_{51} + W_{O2}\, h_{52} + W_{O3}\, h_{53} + b_0)$, for j = 1,2, or 3:

$$\frac{\partial L(Y_{t+6}, \hat{Y}_{t+6})}{\partial W_{Oj}} = \frac{\partial L(Y_{t+6}, \hat{Y}_{t+6})}{\partial \hat{Y}_{t+6}} \times \left(1 - \hat{Y}_{t+6}{}^2\right) \times \frac{\partial (W_{O1} \times h_{51} + W_{O2} \times h_{52} + W_{O3} \times h_{53} + b_o)}{\partial W_{Oj}} = \frac{\partial L(Y_{t+6}, \hat{Y}_{t+6})}{\partial \hat{Y}_{t+6}} \times \left(1 - \hat{Y}_{t+6}{}^2\right) \times h_{5j}$$

Note that d tanh(z) / dz = 1 – (tanh(z))^2, so $\partial \tanh(W_{O1} \times h_{51} + W_{O2}\, h_{52} + W_{O3}\, h_{53} + b_0)/\partial(W_{O1} \times h_{51} + W_{O2}\, h_{52} + W_{O3}\, h_{53} + b_0) \approx 1 - [\tanh(W_{O1} \times h_{51} + W_{O2}\, h_{52} + W_{O3}\, h_{53} + b_0)]^2$.

$$\frac{\partial L(Y_{t+6}, \hat{Y}_{t+6})}{\partial b_O} = \frac{\partial L(Y_{t+6}, \hat{Y}_{t+6})}{\partial \hat{Y}_{t+6}} \times \left(1 - \hat{Y}_{t+6}{}^2\right) \times \frac{\partial (W_{O1} \times h_{51} + W_{O2} \times h_{52} + W_{O3} \times h_{53} + b_o)}{\partial b_O} = \frac{\partial L(Y_{t+6}, \hat{Y}_{t+6})}{\partial \hat{Y}_{t+6}} \times \left(1 - \hat{Y}_{t+6}{}^2\right)$$

Now,

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

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

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

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

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

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

…….

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

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

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

$h_{01} = h_{02} = h_{03} = 0$ or some small random numbers in (-1,+1).

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

$$\frac{\partial L(Y_{t+6}, \hat{Y}_{t+6})}{\partial W_{P1}} = \frac{\partial L(Y_{t+6}, \hat{Y}_{t+6})}{\partial \hat{Y}_{t+6}} \times \frac{\partial \hat{Y}_{t+6}}{\partial W_{P1}} = \frac{\partial L(Y_{t+6}, \hat{Y}_{t+6})}{\partial \hat{Y}_{t+6}} \times \left(1 - \hat{Y}_{t+6}{}^2\right) \times \frac{\partial (W_{O1} \times h_{51} + W_{O2} \times h_{52} + W_{O3} \times h_{53} + b_o)}{\partial W_{P1}} \tag{8.2}$$

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) $\frac{\partial L(Y_{t+6},\hat{Y}_{t+6})}{\partial W_{P1}}$ clearly involves summation of partial derivative terms across each time-step or each sequential recurrent unit or hidden layer, e.g., $\frac{\partial h_{5j}}{\partial W_{P1}}$, $\frac{\partial h_{4j}}{\partial W_{P1}}$, $\frac{\partial h_{3j}}{\partial W_{P1}}$, $\frac{\partial h_{2j}}{\partial W_{P1}}$, $\frac{\partial h_{1j}}{\partial W_{P1}}$ for every j, involving terms of inputs X_1, …., X_5 and previous parameter values $W_{ij}^{[t]}$, and so on. This is unlike the feedforward NN case where a partial derivative would involve summation only across paths originating from the parameter edge and not across all edges in each layer. Firstly, input features a long time in the past such as X_1 can have almost zero contributions to the gradient $\frac{\partial L(Y_{t+6},\hat{Y}_{t+6})}{\partial W_{P1}}$ when the W_{ij}'s are small, i.e., product of small W_{ij}'s vanishes. Secondly, for a many-layered deep RNN, the gradients $\frac{\partial L(Y_{t+6},\hat{Y}_{t+6})}{\partial W_{Pj}}$ lose components involving products of small W_{ij}'s that vanish and hence may not update well. This is the "vanishing gradient" problem in this version of RNN. The flip side of the problem is when W_{ij}'s are large, i.e., > 1. In this case, there is "exploding gradient" problem, and updates to weights are again problematic.

The above partial derivatives are for one case. They are summed across all cases in one mini batch size S. If loss across all cases in the mini batch of size S is $L = 1/S\ \Sigma^S\ L\left(Y_{t+R}, \hat{Y}_{t+R}\right)$, then $\partial L/\partial W_{ij}^{[t]} = 1/S\ \Sigma^S\ \partial L\left(Y_{t+R}, \hat{Y}_{t+R}\right) / \partial W_{ij}^{[t]}$ at the t^{th} iteration. After the updated partial derivatives are found, the usual adjustments of the parameter weights are done, viz.

$$W_{ij}^{[t+1]} = W_{ij}^{[t]} - \alpha(t+1)\frac{\partial L}{\partial W_{ij}^{[t]}}$$

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 backpropagation may in some situations progressively "collapse to zero" or else they may explode exponentially ("vanishing" or else "exploding" gradient with deeper layers); (2) the design does not allow it to consider future or forward input for training since the time-sequence uses information or input from the past or back till the current and is meant to predict the future or forward data. This limitation is real for problems such as predicting a picture by neighboring pixels or a word sequence as in a sentence. However, in general, limitation (2) is appropriate for financial time series where known future state cannot be used to predict an earlier state as this may introduce serious issues of spuriously high accuracies.

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/reduce loss of gradient. Problem (2) may be remedied by using features at current time but are other market-based forecasts of the future state, e.g., a forward contract price on a future spot price on the same underlying commodity.

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. See demonstration file Chapter8-1.ipynb.

[1]:
```
### this notebook is about RNN on predicting daily S&P500 VIX using non-overlapping
###  X-train data, epochs = 30
import pandas as pd
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
```

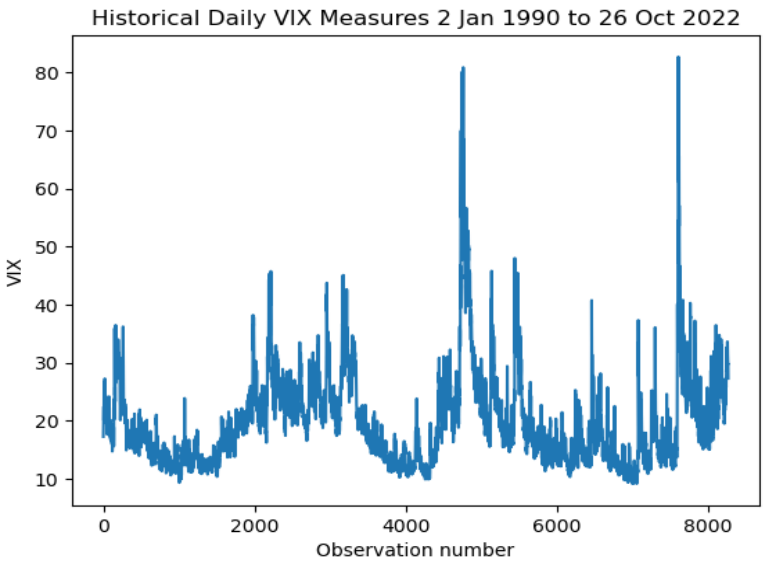
[2]:
```
Vx = pd.read_csv("VIX.csv")
print(Vx)
```

```
           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]
```

The adjusted close VIX prices or "fear gauge" are chosen. This should be similar to the VIX close (in the case of stock prices, they would be different).

[4]:
```
### view time series graph of data -- check if it is  stationary
###  perhaps by some unit root test
VIX.plot()
plt.xlabel('Observation number')
plt.ylabel('VIX')
plt.title('Historical Daily VIX Measures 2 Jan 1990 to 26 Oct 2022')
```

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



The total data set is split into 80% training data (6616 observations) and 20% test data (1654 observations) – see code line [5], [6]. Note that the split is done with the training data set occurring prior to the test data set. It is not a random selection. The financial time series has the property that under reasonable market efficiency, past information and not future information should be part of an investor's consideration for future forecast. Future observations could not have been perfectly known beforehand.

```
[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:]
```

```
[6]: train_data.shape, test_data.shape  ### 0.8 x 8270 = 6616
```

```
[6]: ((6616,), (1654,))
```

Snapshot of the data is shown as follows.

```
In [8]: train_data.values.reshape(-1, 1)
```

```
Out[8]: array([[17.24    ],
               [18.190001],
               [19.219999],
               ...,
               [13.95    ],
               [13.1     ],
               [14.12    ]])
```

Next, the data is scaled to between 0 and 1 using the MinMaxScaler in sklearn. Note that scaling is done separately for the training data and test data as they are separated in time, and there may be drifts over time. It is preferable not to mix them up in using just one scaling.

```
[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
     test_data = scaler.fit_transform(test_data.values.reshape(-1, 1))
     ### (-1,1) reshapes it to a 2D array
```

In codeline [13], the training data and the test data from total sample size T are arranged for the training and then for prediction as shown in Figure 8.6.

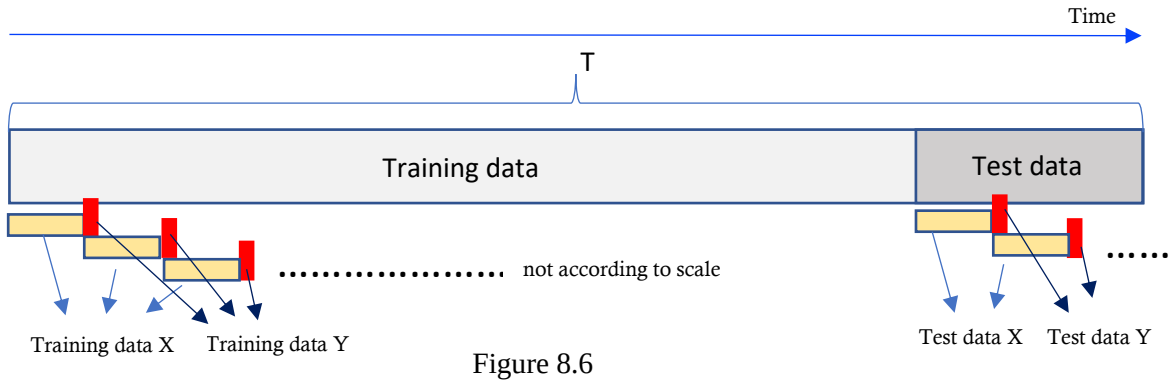


Figure 8.6

```
[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/10 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.

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

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

```
[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
```

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

```
[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
```

```
[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.

```
[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
```

```
[[0.11083158]
 [0.12410903]
 [0.13850453]
 [0.15094341]
 [0.15303983]
 [0.18015375]
 [0.18350805]
 [0.15010481]
 [0.21425575]
 [0.23801537]] [0.25073375]
[[0.20782669]
 [0.20754717]
 [0.21006289]
 [0.18434661]
 [0.24304683]
 [0.21537385]
 [0.22473793]
 [0.22809223]
 [0.23717681]
 [0.23941301]] [0.20195667]
[[0.25073375]

    .
    .
    .
    .
    .
    .
    .
```

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.

```
[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() defines no. of inputs, neurons in hidden & in output layer
    model.compile(loss='mean_squared_error', optimizer='adam')
### .compile in Sequential carries loss and optimizer options
    return model
```

```
[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
```

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.

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 (30) and batch size in one batch, 661 where 661 = (T/10), 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, test_predict is run after train_predict error is minimized or loss
###  is minimized after the training set trainX, trainY are used in .fit

### Dense unit 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
###  an animated progress bar like this: progress_bar. verbose=2 will be most detailed
```

After computing over 661 cases (each is a pack of 10 days) 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:

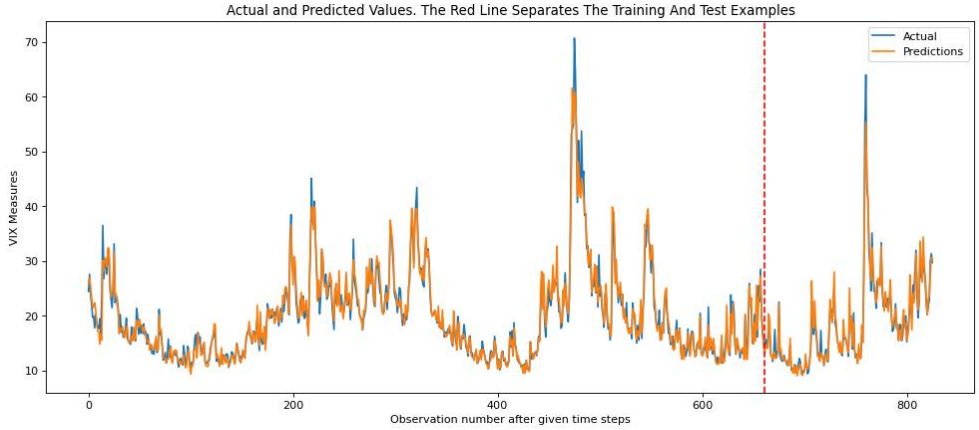
```
[21]: def print_error(trainY, testY, train_predict, test_predict):
          ### Error of predictions
          train_rmse = math.sqrt(mean_squared_error(trainY, train_predict))
          test_rmse = math.sqrt(mean_squared_error(testY, test_predict))
          ### Print RMSE
          print('Train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

      print_error(trainY, testY, train_predict, test_predict)

Train RMSE: 0.022 RMSE
Test RMSE: 0.025 RMSE
```

The RMSEs are about 2.2% for the training and 2.5% for the testing. This level of accuracy is not significantly improved in the RNN for this study even if we perform overlapping data using all 6606 number of cases for training and increasing 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.

```
[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)
```

```
[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='r', 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 the partial derivatives in the RNN back propagation collapse toward zero) – 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, a variant of RNN, can be used.

In the LSTM architecture, the recurrent cell in Figure 8.4 is redesigned as follows in Figure 8.7. There is an additional state (also not explicitly observed) called the 'cell state'. Like the hidden states h_t in a traditional RNN as in Figure 8.4 that stores 'short-term' memory of effects of past inputs, the cell state c_t stores 'long-term' memory of effects of past inputs. As in Figure 8.4, each "box" with each fresh input represents one hidden layer in the RNN. Each "box" can contain multiple neurons as width.

Four 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 a Hadamard product with output from the (3) Cell Update Gate, and then to make the final update on the cell state that is preliminarily updated by (1), (4) the Output Gate before they update the hidden state in conjunction with the cell state c_t.

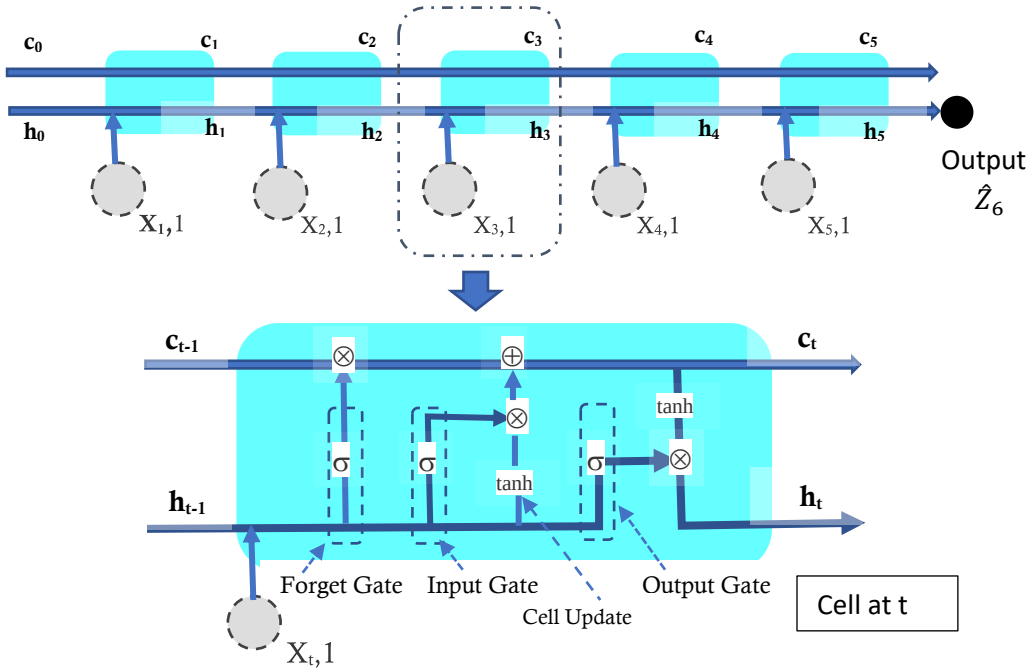


Figure 8.7

Suppose we create 50 neurons in the hidden layer or the cell. In the Forget Gate (1), concatenated inputs (X_t,1, 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.

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

where weights W_{FX} and W_{Fh} have appropriate dimensions to match the dimensions in X_t and h_{t-1}. For example, at each time of input, X_t could be 1×1 and 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 $\sigma(W_{FX}\,X_t + W_{Fh}\,h_{t-1} + b_F)$ 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 vector in 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 $\sigma(W_{FX}\,X_t + W_{Fh}\,h_{t-1} + b_F)$ is then multiplied by c_{t-1} (symbol ⊗ in Figure 8.7) using the Hadamard product or element to element multiplication. This provides the preliminarily updated cell state.

In Input Gate (2), concatenated inputs (X_t,1, h_{t-1}) after weighting $\sigma\,(W_{IX}\,X_t + W_{Ih}\,h_{t-1} + b_I\,)$ is used in Hadamard product with cell update candidate $\tanh\,(W_{CX}\,X_t + W_{Ch}\,h_{t-1} + b_C\,)$ to feed as addition to update the preliminarily updated cell state c_{t-1} from (1) to c_t. This Input Gate identifies only important impacts from the concatenated inputs to be added to the long-term memory cell c_t where

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

In Output Gate (3), the tanh activation on the final updated cell state c_t is then multiplied (Hadamard product ⊗) by $\sigma\,(W_{OX}\,X_t + W_{Oh}\,h_{t-1} + b_O\,)$ to yield the next hidden state h_t that flows into the next LSTM layer, viz.

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

In a many-to-one structure as in Figure 8.7, the final output could be a scalar predicted number provided by the output layer comprising 51 weights (dimension of output h_t and the bias), viz. $f(W_{Yh}\,h_t + b_Y)$ where W_{Yh} and b_Y denote final output weights ("Y" being the output state) from a RNN/LSTM and f(.) is the final activation function.

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 $4 \times [(50 + 1) \times 50 + 50]$ 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 $4 \times [(N + J) \times N + N]$ 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) in one layer of LSTM, there are $4 \times [(50 + 1) \times 50 + 50] = 10{,}400$ parameters. All adjacent layers of 50 neurons in same LSTM share the same $4 \times [(50 + 1) \times 50 + 50] = 10{,}400$ parameters.

However, single LSTM (even with a large number of neurons per layer or width, or a large number of adjacent layers) and many epochs may not be able to produce 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 LSTM with two stacked hidden layers at each time-sequence is shown below in Figure 8.8. 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 hidden state $h_t^{[2]}$, i.e. $f(W_{Yh}\, h_t^{[2]} + b_Y)$ where $h_t^{[2]} = \sigma\,(W_{OX}^{[2]}\, h_t^{[1]} + W_{oh}^{[2]}\, h_{t-1}^{[2]} + b_O^{[2]}) \otimes \tanh\,(c_t^{[2]})$.

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 8.8.

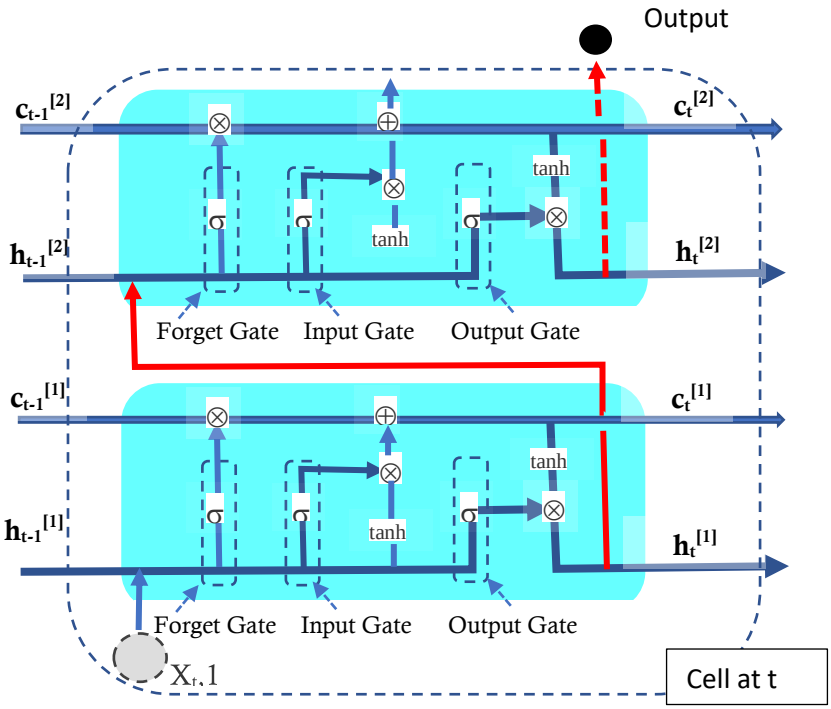


Figure 8.8

The Output Gate from first level cell provides 50 hidden states at level 1, viz.

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

where the superscripts to the parameters denote association with the cell level. Together with $h_{t-1}^{[2]}$, these are inputs to the Forget Gate, Input Gate, Candidate for Cell State Update, and the Output Gate at

level 2. Since N = 50, and dimension of $h_{t-1}^{[2]}$ (same dimension as $h_{t-1}^{[1]}$) and of $h_t^{[1]}$ are both N, then (N + N) inputs 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. The Forget Gate output fraction vector $\sigma(W_{FX}^{[2]} h_t^{[1]} + W_{Fh}^{[2]} h_{t-1}^{[2]} + b_F^{[2]})$ is then multiplied by $c_{t-1}^{[2]}$ 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 × [(N + N) × N + N] parameters at the second level LSTM cell/stacked layer. If N = 50, then there is a total of 4 × [(50 +50) × 50 +50] = 20,200 parameters at the second level LSTM cell.

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.

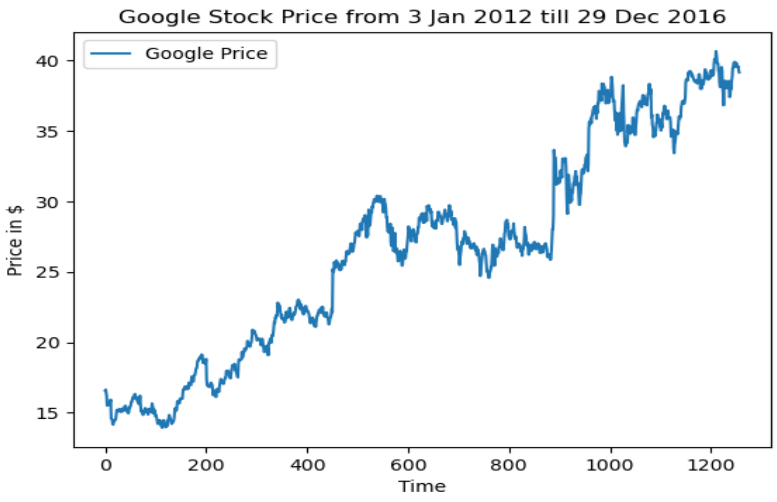
```
[2]: ### import training set
     dataset=pd.read_csv('GOOG.csv')
     dataset.head()
```

[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.

```
[3]: p=dataset.iloc[0:1259,5:6] ### reads in 'Adj Close'
     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 same MinMaxScaler.

```
[4]: training_set=dataset.iloc[0:1100,5:6].values
     trgsset=pd.DataFrame(training_set)

     test_set=dataset.iloc[1100:1259,5:6].values
     tstset=pd.DataFrame(test_set)

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

     (1257, 7) (1100, 1) (157, 1)
```

```
[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)

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

Next, the data structure is created to feed the inputs to the NN. As in Figure 8.6, T = 1100 for the training data set. The first 60 data points are used as a case in training data X (X_train) while the next data point (point 61) is used as the corresponding label 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.

```
[7]: ### creating data structure with 1040 cases, each 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)
     X_train=np.reshape(X_train, (X_train.shape[0], X_train.shape[1],1))
     ### This last step converts X_train to 3D from (1040,60) to (1040,60,1)
     ###  for input to the keras app
     print(X_train.shape, y_train.shape)

     (1040, 60, 1) (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 dropout rate is a regularization tool applied to reduce overfitting. In general, it is used in deep learning NN with many neurons and layers. If the dropout rate is set to say 0.20 at a particular layer (including input layer), then at that layer, 20% of the neurons will be randomly selected to “disappear” in that iteration of forward and backward propagation through that layer. In that iteration, there is no forward pass from the temporarily dropped out neurons, and there is also no updates or revisions in weights connected to those neurons during the backward propagation. In the next iteration, other neurons will be dropped out and revisions of weights continue as before. When dropout rate is added to the training, it is sometimes suggested that the width of the layer be scaled up by 1/(1 – dropout rate) so that the sum of all effective

neurons in each iteration remains the same. Dropout can prevent overfitting and allow better prediction in generalized data.

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

[10]:
```
### Initializing RNN
model = Sequential()
```

[11]:
```
### Add first LSTM layer and add Dropout Regularization

model.add(LSTM(units=50,return_sequences=True,input_shape=(X_train.shape[1],1)))

### Sequential reads input as 3D. Add return_sequences=True for all LSTM layers except
###  the last one. Setting this flag to True lets Keras know that LSTM output should
###  be 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 does not feed to another LSTM layer.

model.add(Dropout(0.2))
### This is a hyperparameter that prevents overfit by dropping 20% of nodal values
```

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

[13]:
```
### Add third LSTM layer and Dropout
model.add(LSTM(units=50,return_sequences=True))
model.add(Dropout(0.2))
```

[14]:
```
### 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))
```

[15]:
```
### Add output layer
model.add(Dense(units=1))  ### not capital "U"nit
```

[16]:
```
### Compiling the RNN
model.compile(optimizer='adam',loss='mean_squared_error')
```

[17]:
```
### Run the training set with the LSTM (specialized RNN here)
model.fit(X_train,y_train,epochs=100,batch_size=10)
```

In codeline [16], 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 [17] 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.

[18]:
```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 60, 50)	10,400
dropout (Dropout)	(None, 60, 50)	0
lstm_1 (LSTM)	(None, 60, 50)	20,200
dropout_1 (Dropout)	(None, 60, 50)	0
lstm_2 (LSTM)	(None, 60, 50)	20,200
dropout_2 (Dropout)	(None, 60, 50)	0
lstm_3 (LSTM)	(None, 50)	20,200
dropout_3 (Dropout)	(None, 50)	0
dense (Dense)	(None, 1)	51

Next we perform the prediction (getting the LSTM output layer output) by feeding in the feature data in X_train. See code line [19]. As the output 'predict_train' is multiple array format, we use code line [21] to retrieve the column containing the 1040 predicted outputs with each case.

[19]:
```
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 ━━━━━━━━━━━━━━━━━━━━ 2s 43ms/step
(1040, 1)
```

[20]:
```
print(predict_train)
```

```
[[0.08865437]
 [0.08858573]
 [0.08465725]
 ...
 [0.86496216]
 [0.86246455]
 [0.87109554]]
```

[21]:
```
predict_train=predict_train[:, 0]
### select first column or first day of 60 days of each day prediction from 1 to 1040
```

[22]:
```
print(predict_train)
```

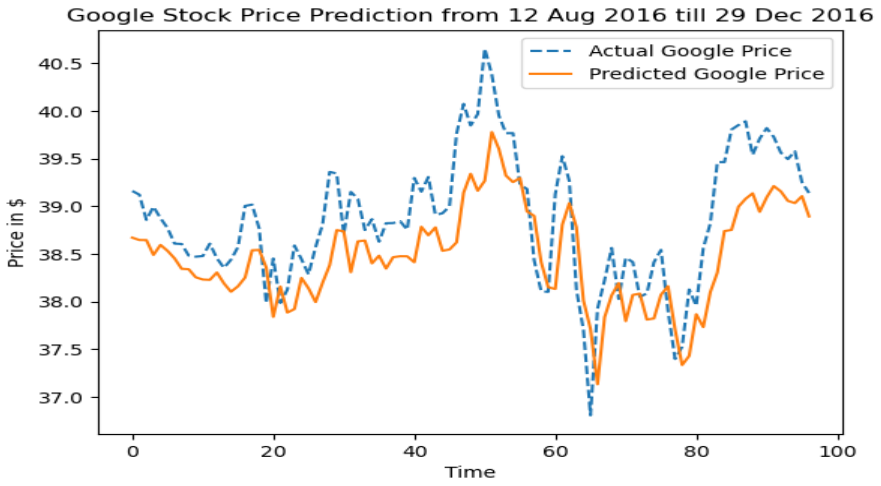
```
[0.08865437 0.08858573 0.08465725 ... 0.86496216 0.86246455 0.87109554]
```

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.

[23]:
```
### 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))
```

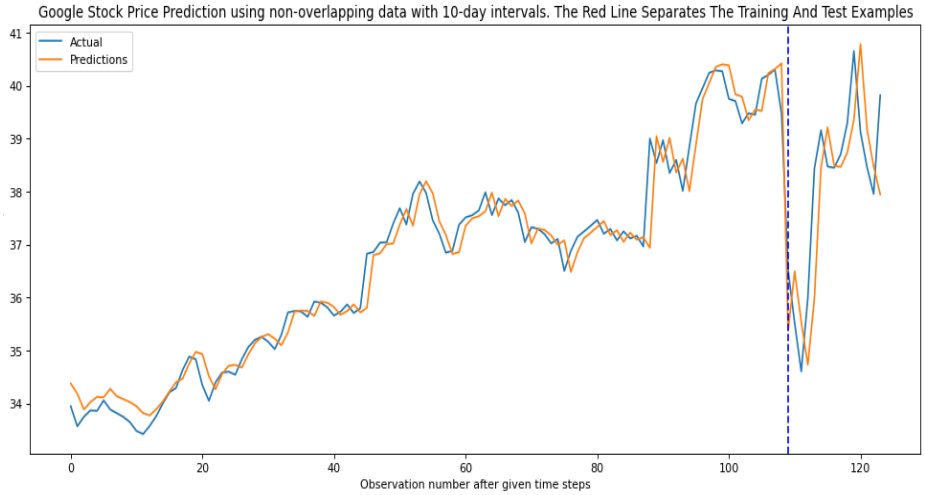
```
[24]: y_test.shape ### now (97.) is 1 Dim -- convert to two dim

[24]: (97,)

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

[26]: ### 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

      4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step

[28]: 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: 0.021 RMSE
      Test RMSE: 0.022 RMSE
```

The RMSE for the training set fitting and then the test set prediction are about 2.1 % and 2.2% respectively. Recall that a different run will produce slightly different RMSEs due to the random seeds in the NN. The 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 5.3% and 16.3% respectively. The 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. The usual forward hidden state updates are h_t^f based on inputs $(X_t,1)$ and past hidden state h_{t-1}^f.

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

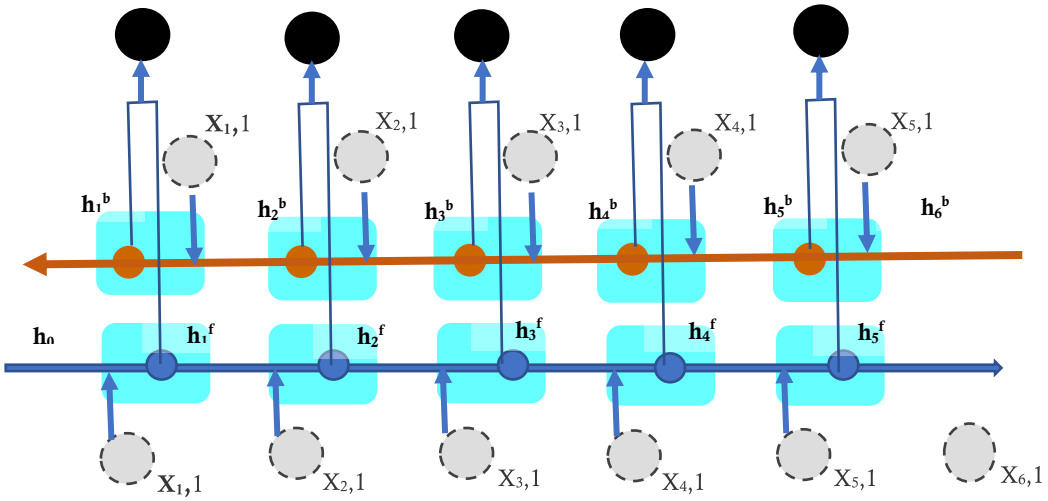


Figure 8.9

The additional backward hidden state updates are now h_t^b based on inputs $(X_t,1)$ and future hidden state h_{t+1}^b. Their outputs are concatenated for final output. The diagram 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.

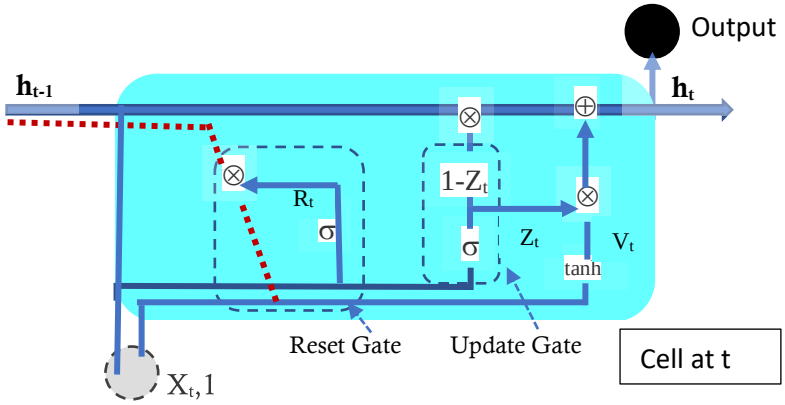


Figure 8.10

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

The Reset Gate decides how much of the past information in 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 $Z_t = \sigma\ (W_{UX}\ X_t + W_Z\ h_{t-1} + b_Z\ )$. The output hidden state $h_t = Z_t \otimes V_t + (1– Z_t) \otimes h_{t-1}$. The Update Gate determines a weighted average of V_t and past h_{t-1}. A detailed explanation of an alternative variation – the Gated Recurrent Unit (GRU) can be found in https://towardsdatascience.com/ understanding-gru-networks-2ef37df6c9be.

Convolutional Neural Network (CNN)

Another type of deep-learning neural network is convolutional neural network (CNN) that is frequently applied in image identification and classification. The CNN approach can also be applied to time series data for prediction and classification. In image recognition or classification, the input is typically a 2-dimensional matrix of pixel values. A pixel value could be the representation of shade of a pixel (smallest unit or building block of a digital image) at a location. A rugby ball could perhaps be represented as follows on the left and the figure 5 could be represented by 5 x 5 matrix on the right.

0	0	1	0	0
0	2	3	2	0
0	3	5	3	0
0	2	3	2	0
0	0	1	0	0

0	1	1	1	0
1	1	0	0	0
0	2	1	2	0
0	0	0	3	0
0	3	2	0	0

A colored picture could be represented by 3 sets of matrices, each corresponding to a color (RGB). In the color context, the 3 sets of filters can also be termed 3 channels. An input picture with pixel values in a huge, e.g., 10,000 × 10,000 matrix, could be reduced to a 5 × 5 matrix using several operations such as convolution filters, activation, and pooling. As an example, a 7 × 7 input matrix on the left when convolved with a 3 × 3 filter based on element by element multiplication and then summing all elements in the matrix produces 0 and is entered in the output matrix.

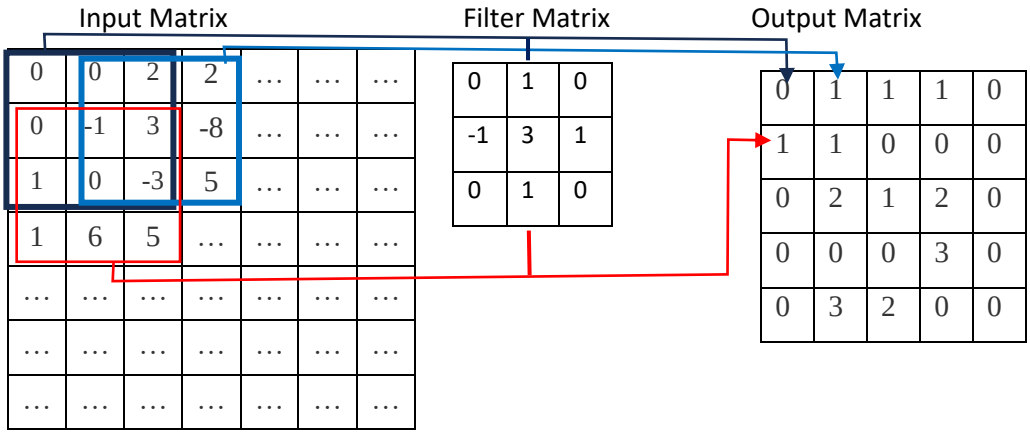


Figure 8.11

This operation is a convolution. The output matrix is also called a feature map. Often, an activation function such as ReLU is put on this summing operation to ensure the output is a positive number.

The 3 × 3 filter works across the input matrix on each 3 × 3 sub-matrix from left to right, then top to bottom like a magnifying glass. There are two hyperparameters in the convolution operation: padding and stride. A padding of size p ($p \geq 0$) means adding an extra p first and p last rows of zeros and also an extra p first and p last columns of zeros to the matrix. Thus a padding of p=2 would mean an output matrix of 7 × 7 without shrinking the original image size. A stride of 1 means that the shift of the filter is one column right or one row down at a time. This is the case shown in Figure 8.11. A stride of 2 means that the shift of the filter is two columns right or two rows down at a time, and so on.

The above makes up a convolution and is a convolution layer in the CNN. It serves to sharpen the image. Often many filters (randomly initiated) are put onto one input image to derive many feature maps for use later. One could be to detect line shapes, another to detect curve shapes, and so on. Together they aggregate to form more features to identify or classify the input image or input time series.

Next the feature map is typically subject to a Pooling layer in which the feature map or output matrix in the convolution layer is reduced (reducing the number of feature inputs) via a k × k filter. The pooling filter could be a Max Pooling operation whereby the output is the maximum of all elements in the k × k sub-matrix. Or it could be a Sum Pooling or else Averaging Pooling operation.

Each filter in a convolution layer typically produces one output feature map. The outputs or feature maps could be passed through several alternating convolution and pooling layers before they are then flattened into a single dimension array of feature inputs into a dense (fully connected) neural network (layer(s) of neurons). The flattening concatenates all the final feature maps. So, if each final feature map is of dimension k × k, then m feature maps mean a flattening to an input feature array of $k^2 \times m$. In a single time series prediction, the original image could be a one-dimensional array of numbers such as stock prices. The filter could be also a k × 1 vector that shifts down the array of numbers to produce the feature map. This is shown in Figure 8.12.

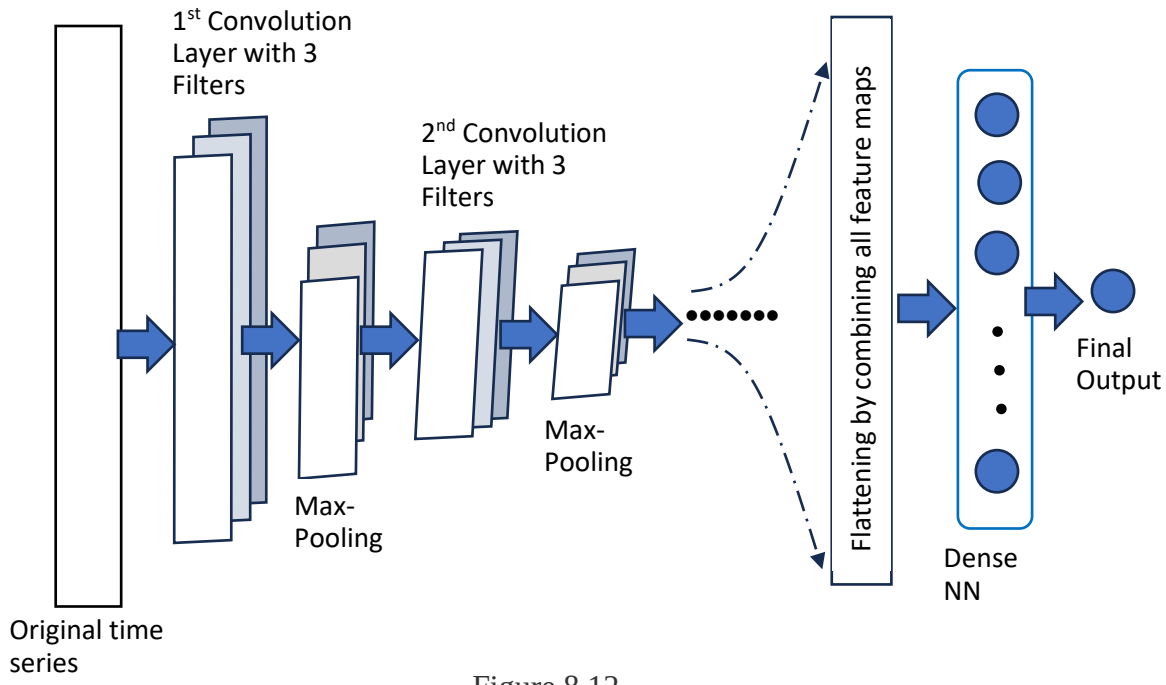


Figure 8.12

The final output is in the output layer which is either a classification or a regression result. The CNN can be adapted to work with more complicated NN such as LSTM by coupling the convolutional and pooling layers with the dense LSTM. 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.

Using the same data set of Google prices in Worked Example II (demonstration file Chapter8-2.ipynb), the CNN is applied to predict the following day stock price. The algorithm uses 12 1-D filters with zero padding, kernel size of 3, and activation function ReLU. MaxPooling is done with a pool size equal to 2. These are fed into a dense 100 neurons NN. Execution involves 1,000 epochs and batch size of 10. The code lines are shown as follows.

```
[39]: # Define model architecture
      model = Sequential()
      model.add(Conv1D(filters=12,kernel_size=3,activation='relu',input_shape=(60,1),padding='valid'))
      model.add(MaxPooling1D(pool_size=2))
      model.add(Flatten())
      model.add(Dense(100,activation='relu'))
      model.add(Dense(1))
      model.compile(optimizer='adam',loss='mse')
```

```
[40]: # fit CNN model
      history = model.fit(X_train,y_train,epochs=1000,batch_size=10,verbose=1)
      print(history)
```

Prediction is shown in code lines [41], [42], and the RMSE results are reported in code line [44].

[42]:
```
CNN_y_train = model.predict(X_train, verbose=0)
print(CNN_y_train)
```

```
[[0.08787928]
 [0.08102182]
 [0.08562031]
 ...
 [0.85767883]
 [0.8520642 ]
 [0.8435601 ]]
```

[43]:
```
# Testing the model
CNN_y_pred = model.predict(X_test, verbose=0)
print(CNN_y_pred)
```

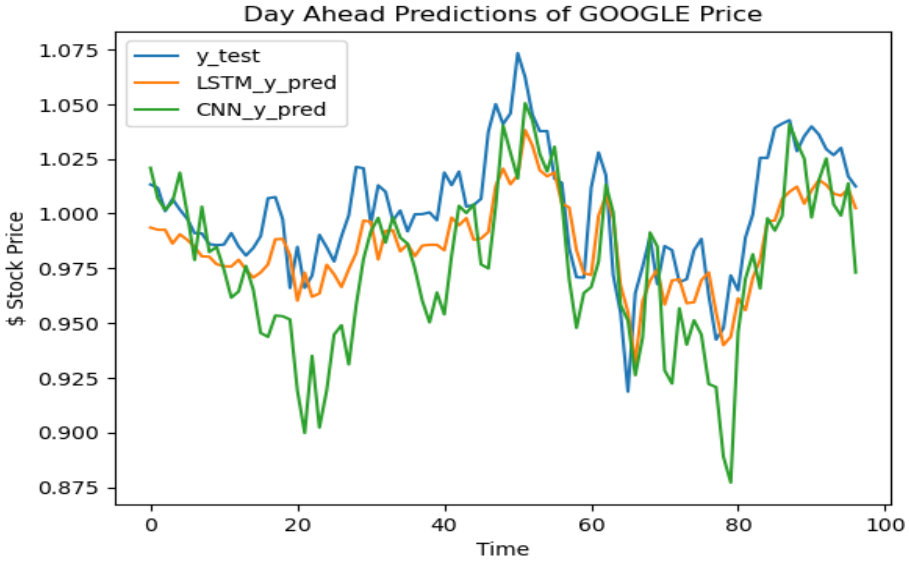
```
[[1.0208956 ]
 [1.0069052 ]
 [1.0016748 ]
 [1.0052946 ]
```

.....
.....
.....

[44]:
```
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, CNN_y_train, CNN_y_pred)
```

```
Train RMSE: 0.011 RMSE
Test RMSE: 0.036 RMSE
```

The CNN training error of RMSE 1.1% is smaller than that of LSTM 2.1%. However, for the testing data set, CNN RMSE of 3.6% is larger than the 2.2% of LSTM. The predicted CNN prices in the test set are compared with the LSTM predicted price. The prediction results shown in the following graph are comparable.



8.5 Concluding Thoughts

In this chapter we see some of the deep Neural Networks and also see examples of using them to predict VIX and also a stock price. The standard ML process is used involving training the model (or fitting the model on the training data) and then testing the trained model. Hyperparameters that are involved in the model fitting are assumed to be optimized in the training process. Otherwise, a validation data set would have to be used to do the hyperparameter fine-tuning. Doing it all on the training data set and not cutting off a piece of the data for validation has the advantage of using enough data on a time series. However, it has the disadvantage of losing some robustness as validation and also a testing provide a double check before the model is rolled out for real action in generalized data and on real problems that may yield costly decisions if the model is not performing.

In the training and testing for a next day stock price prediction, we show that it is suitable to deploy a rolling window with a sequence of training data prices followed by a test data point. Thus, it is not the usual deployment of a sequential training data set and a sequential test data set. The one-period ahead training obviously uses the most recent information and is in line with the idea of market informational efficiency, in whatever degree.

For stock price prediction for trading purposes, there are two major caveats. One is transactions costs. Any apparent trading profits can be decimated or vanquished by transaction fees/costs, impact costs/widening bid-ask spread if more want to buy (ask goes up) or 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, and 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 not as fast as competitors.

Two is the technical aspect of the prediction by minimizing mean-square errors. The results for Google indicate that the NNs are able to provide forward predictions with a small MSE. However, a closer look may reveal that most of the time, the algorithm is trend chasing. Hence the predicted curve appears to be a forward shift of the past price curve. Trading profit may be possible if the trend persists and that there is no major price reversals. Otherwise, a buy on an upward trend (or a sell on a downward trend) in the face of a major market fall (rise) would face huge losses (unless one has the capital to hold on and suffer marked-to-market losses) that can easily wipe out all the previous trend-chasing gains.

More meaningful ML models for trading 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 such as average price movements in the last few intervals, bid-ask spread, volume, a market-sentiment measure, firm size, price-earnings ratios, price-to-book ratios, profit news, competitor prices, industry price, market movements, interest rates, and so on. And importantly, the model is effective only when back-testing on a trading strategy, e.g., buy low and sell high or sell high and buy low, can yield a significant profit.

8.6 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