

**QF634 APPLIED QUANTITATIVE RESEARCH
METHODS LECTURE 8**

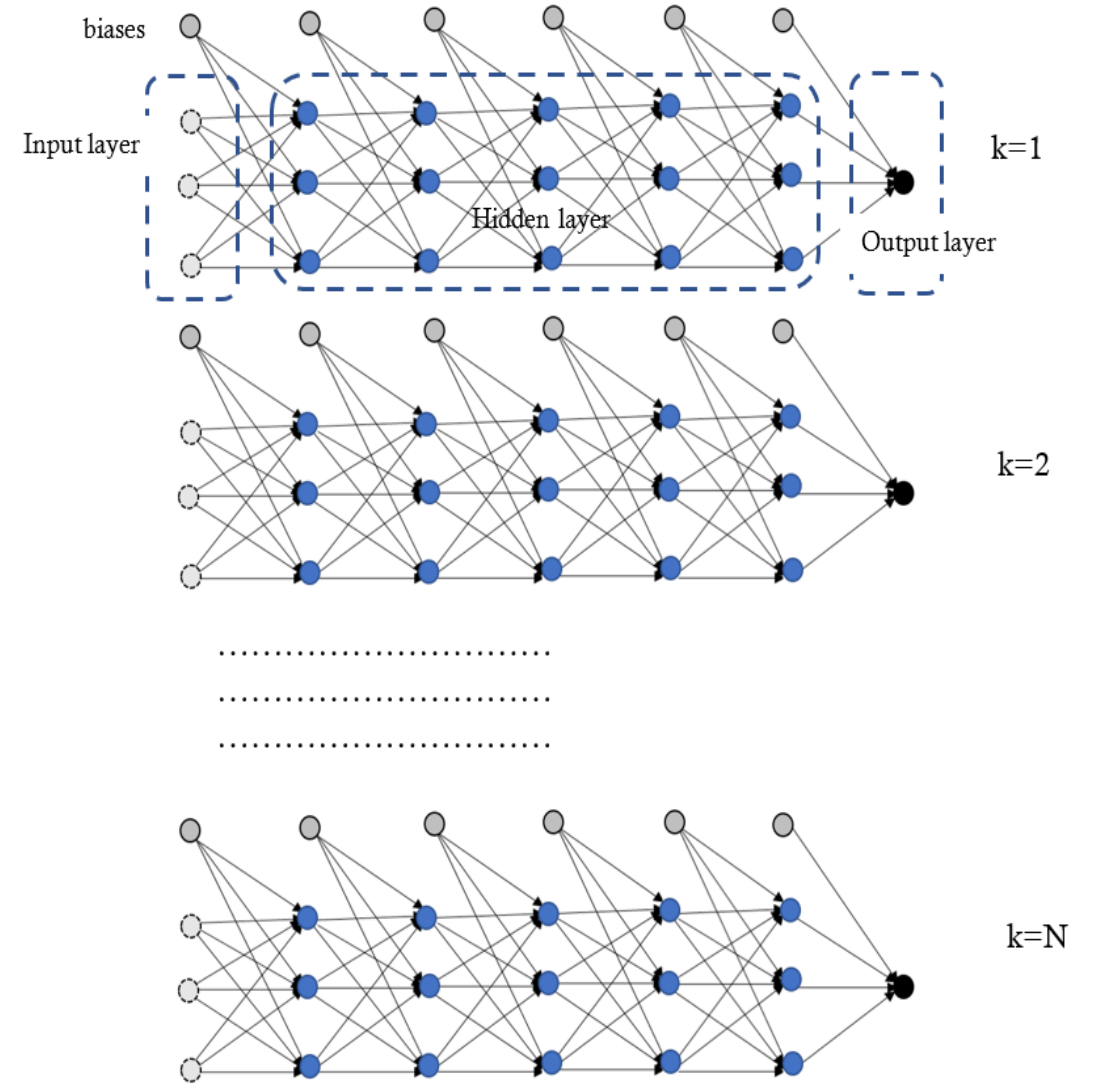
Lecturer: Prof Lim Kian Guan

ARTIFICIAL INTELLIGENCE

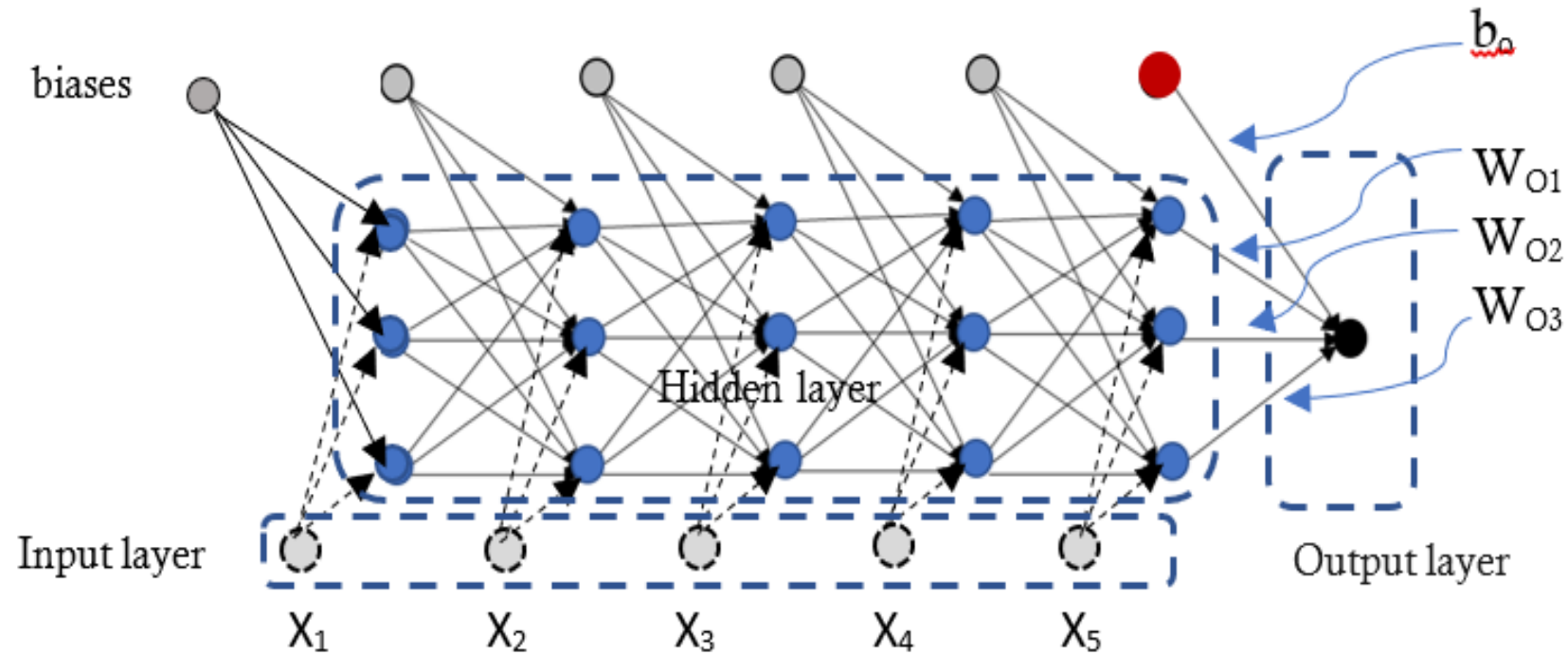
Artificial Neural Network II: Deep Learning NN

N multilayer perceptrons (MLP), an identical one for each case/subject k. For each k, the inputs $\{X_{i,k}\}$ are features ($i=1,2,\dots,3$) related to subject k. The features/inputs are marked as dotted circles in the input layer.

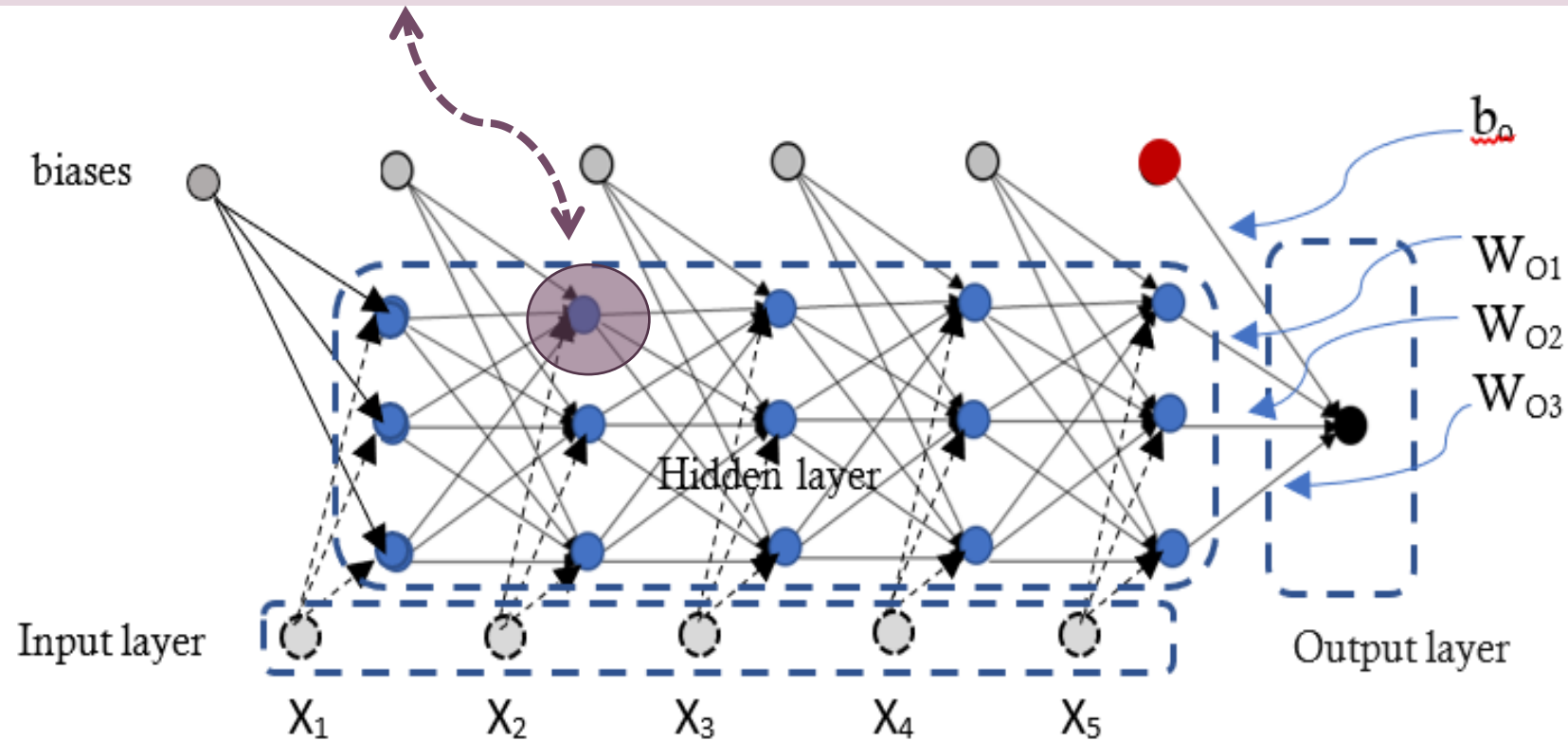
When cases k are independent of one another, i.e., their inputs and output do not affect other cases, then the common (across k) input weights and biases, and weights and biases in hidden layer(s) are computed via backpropagation to minimize the error/loss function $L(.)$ that combines all cases $\sum_{k=1}^N \frac{1}{N} L(Y_k, \hat{Z}_k)$ 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.



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.



For example, the parameters to the first (uppermost) neuron in the 2nd 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.

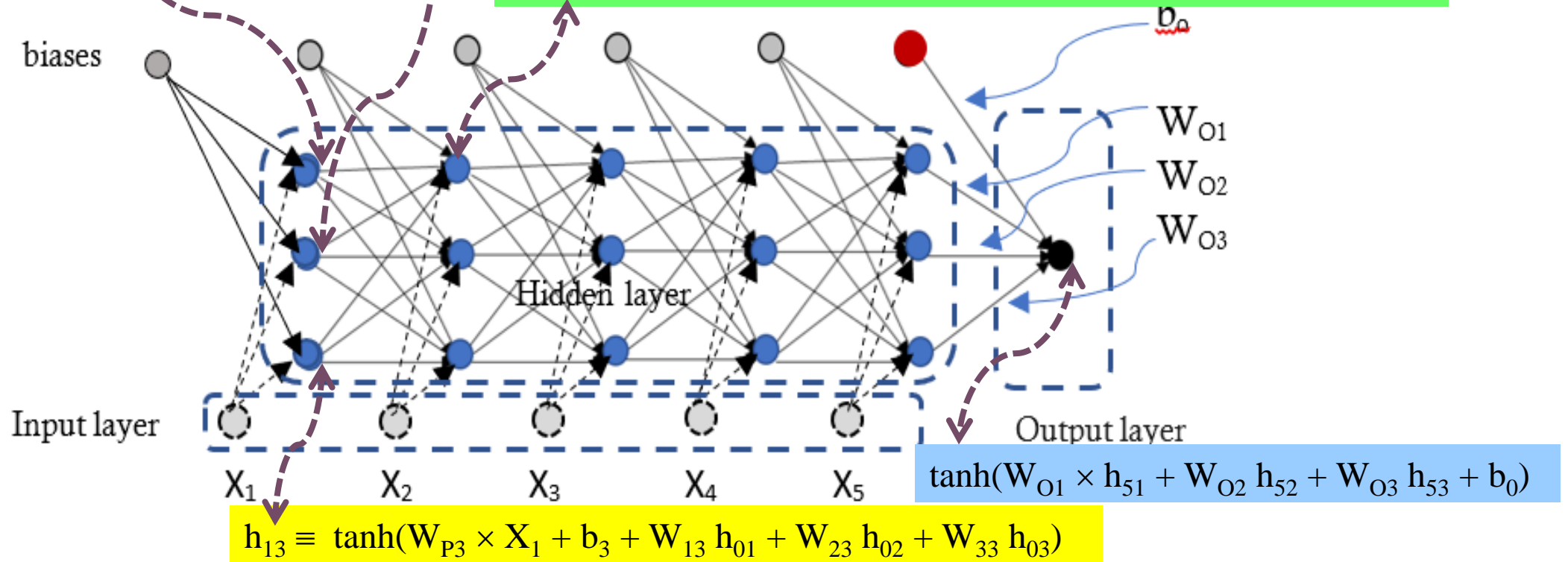


Output from the first neuron in the first hidden layer is

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

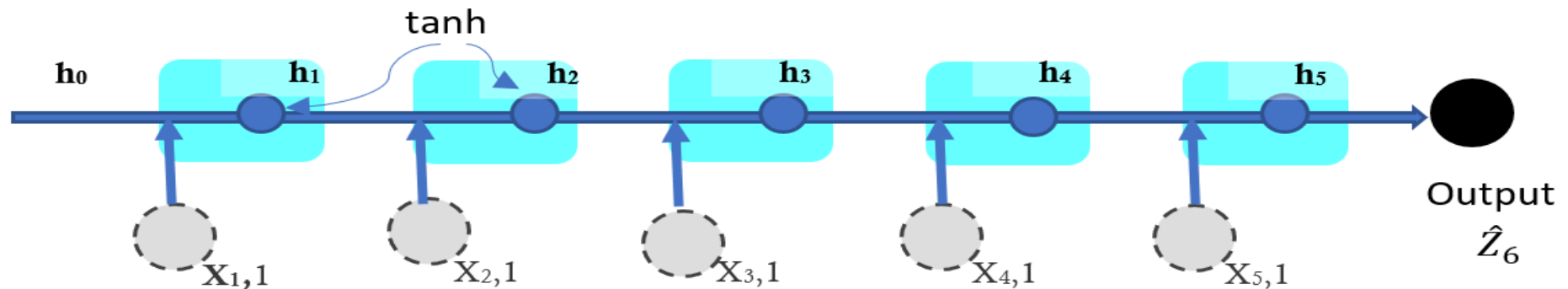
$$h_{12} \equiv \tanh(W_{P2} \times X_1 + b_2 + W_{12} h_{01} + W_{22} h_{02} + W_{32} h_{03})$$

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



A concise representation of Recurrent Neural Network

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.



In the training of the RNN, we treat each of the R ($=5$ here) recurrent units like R 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. A total sample could have a time series or a sequence of T units. The units are then divided into cases or packs each with R recurrent units/hidden layers. The number of cases or packs is T/R . 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.

Backward Propagation in RNN

$$\frac{\partial L(Y_{t+5}, \hat{Y}_{t+5})}{\partial W_{P1}} = \frac{\partial L(Y_{t+5}, \hat{Y}_{t+5})}{\partial \hat{Y}_{t+5}} \times \frac{\partial \hat{Y}_{t+5}}{\partial W_{P1}} = \frac{\partial L(Y_{t+5}, \hat{Y}_{t+5})}{\partial \hat{Y}_{t+5}} \times \left(\frac{\partial \hat{Y}_{t+5}}{\partial h_{51}} \times \frac{\partial h_{51}}{\partial W_{P1}} + \frac{\partial \hat{Y}_{t+5}}{\partial h_{52}} \times \frac{\partial h_{52}}{\partial W_{P1}} + \frac{\partial \hat{Y}_{t+5}}{\partial h_{53}} \times \frac{\partial h_{53}}{\partial W_{P1}} \right)$$

where

$$\frac{\partial h_{51}}{\partial W_{P1}} = \left(\frac{\partial h_{51}}{\partial W_{P1}} + \frac{\partial h_{51}}{\partial h_{41}} \times \frac{\partial h_{41}}{\partial W_{P1}} + \frac{\partial h_{51}}{\partial h_{42}} \times \frac{\partial h_{42}}{\partial W_{P1}} + \frac{\partial h_{51}}{\partial h_{43}} \times \frac{\partial h_{43}}{\partial W_{P1}} \right)$$

$$\frac{\partial h_{52}}{\partial W_{P1}} = \left(\frac{\partial h_{52}}{\partial h_{41}} \times \frac{\partial h_{41}}{\partial W_{P1}} + \frac{\partial h_{52}}{\partial h_{42}} \times \frac{\partial h_{42}}{\partial W_{P1}} + \frac{\partial h_{52}}{\partial h_{43}} \times \frac{\partial h_{43}}{\partial W_{P1}} \right)$$

$$\frac{\partial h_{53}}{\partial W_{P1}} = \left(\frac{\partial h_{53}}{\partial h_{41}} \times \frac{\partial h_{41}}{\partial W_{P1}} + \frac{\partial h_{53}}{\partial h_{42}} \times \frac{\partial h_{42}}{\partial W_{P1}} + \frac{\partial h_{53}}{\partial h_{43}} \times \frac{\partial h_{43}}{\partial W_{P1}} \right)$$

and so on. This clearly involves summation of partial derivative terms and their products across each time-step or each sequential recurrent unit or hidden layer, involving terms of inputs X_1, \dots, X_5 . This is unlike the feedforward NN case where a partial derivative would involve summation across paths originating from the parameter edge.

Advantages and Disadvantages of RNN

- 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 unit), 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 the latest actual output just after the training sequence for the training; this is unlike an ANN where inputs can include estimated future variables;

Worked Example I -- Data

Please upload Chapter8-1.ipynb and follow the computing steps in Jupyter Notebook

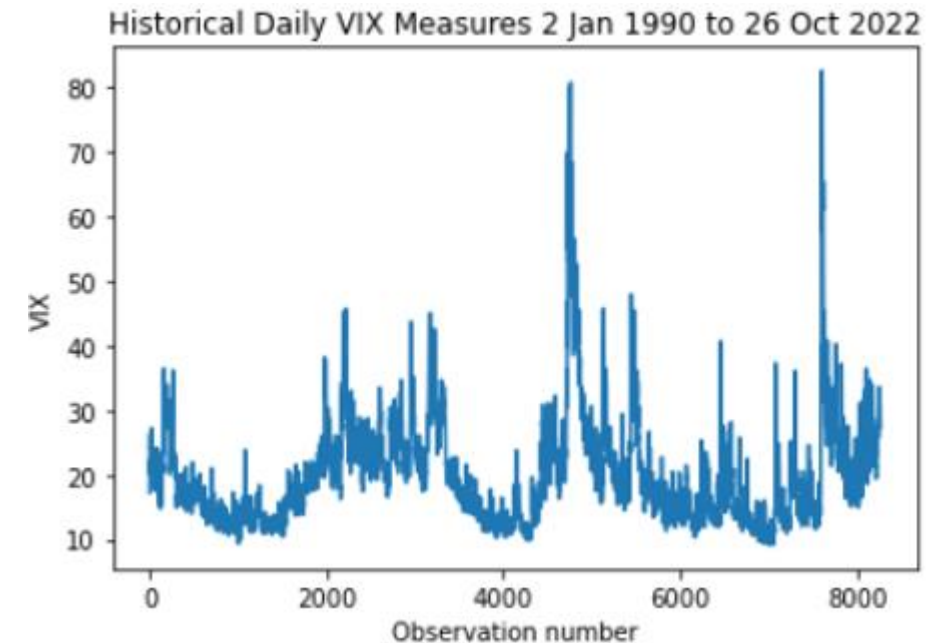
- The CBOE Volatility Index (VIX) is a real-time index derived from the prices of SPX (S&P 500) index options with near-term (approximately 1 month) expiration dates. It is market expectation of the SPX risk-neutral volatility over the short term. VIX has been called a ‘fear gauge’ and is often seen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

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]

Please upload Chapter8-1.ipynb and follow the computing steps in Jupyter Notebook

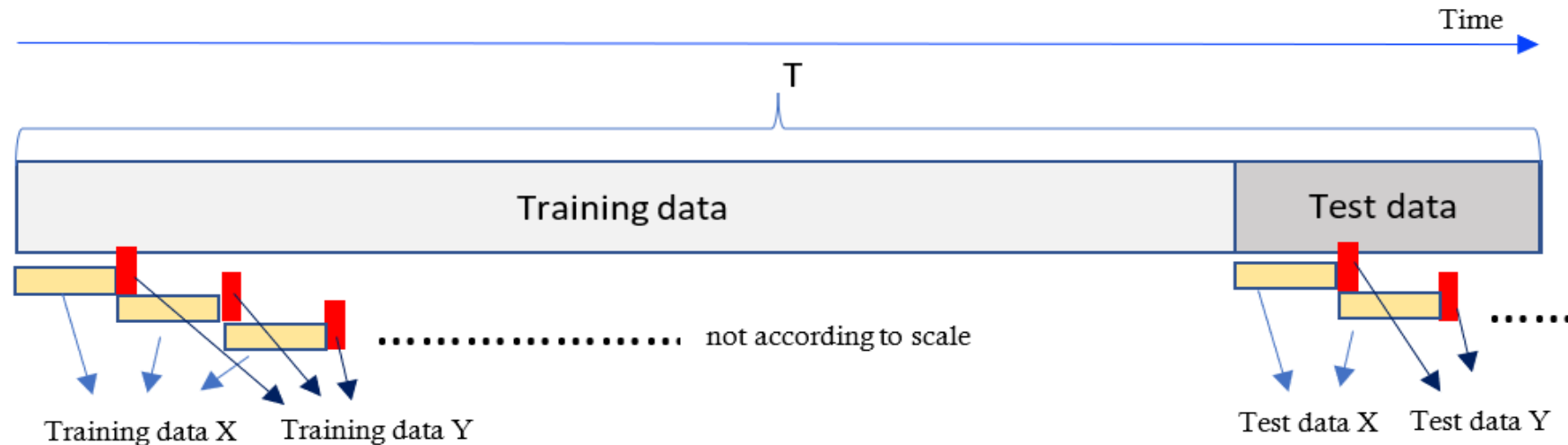


Transforming the Data

Please upload Chapter8-1.ipynb and follow the computing steps in Jupyter Notebook

```
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
```

Arranging Data for Training and Testing



Arranging Data for Training and Testing

Please upload Chapter8-1.ipynb and follow the computing steps in Jupyter Notebook

```
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)
    X = dat[range(time_steps*rows_x)]
    X = np.reshape(X, (rows_x, time_steps, 1))
    return X, C

    ### here len(C) is 661
    ### range(L) is 0,1,2,..., L-1. L is 10 x 661 = 6610. X is array 1,2,...,6610
    ### X reshaped as (661,10,1)

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)
```

We use a sequence of 10 steps (approximately two weeks of 10 trading days) to form one pack of 10 recurrent units. 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.

RNN Structure

Please upload Chapter8-1.ipynb and follow the computing steps in Jupyter Notebook

```
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

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
```

Fitting and Prediction

Please upload Chapter8-1.ipynb and follow the computing steps in Jupyter Notebook

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 $(m+2)(m+1) - 1 = 10 \times 9 - 1 = 89$ 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,)
```

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
```

Minimized Loss

Please upload Chapter8-1.ipynb and follow the computing steps in Jupyter Notebook

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:

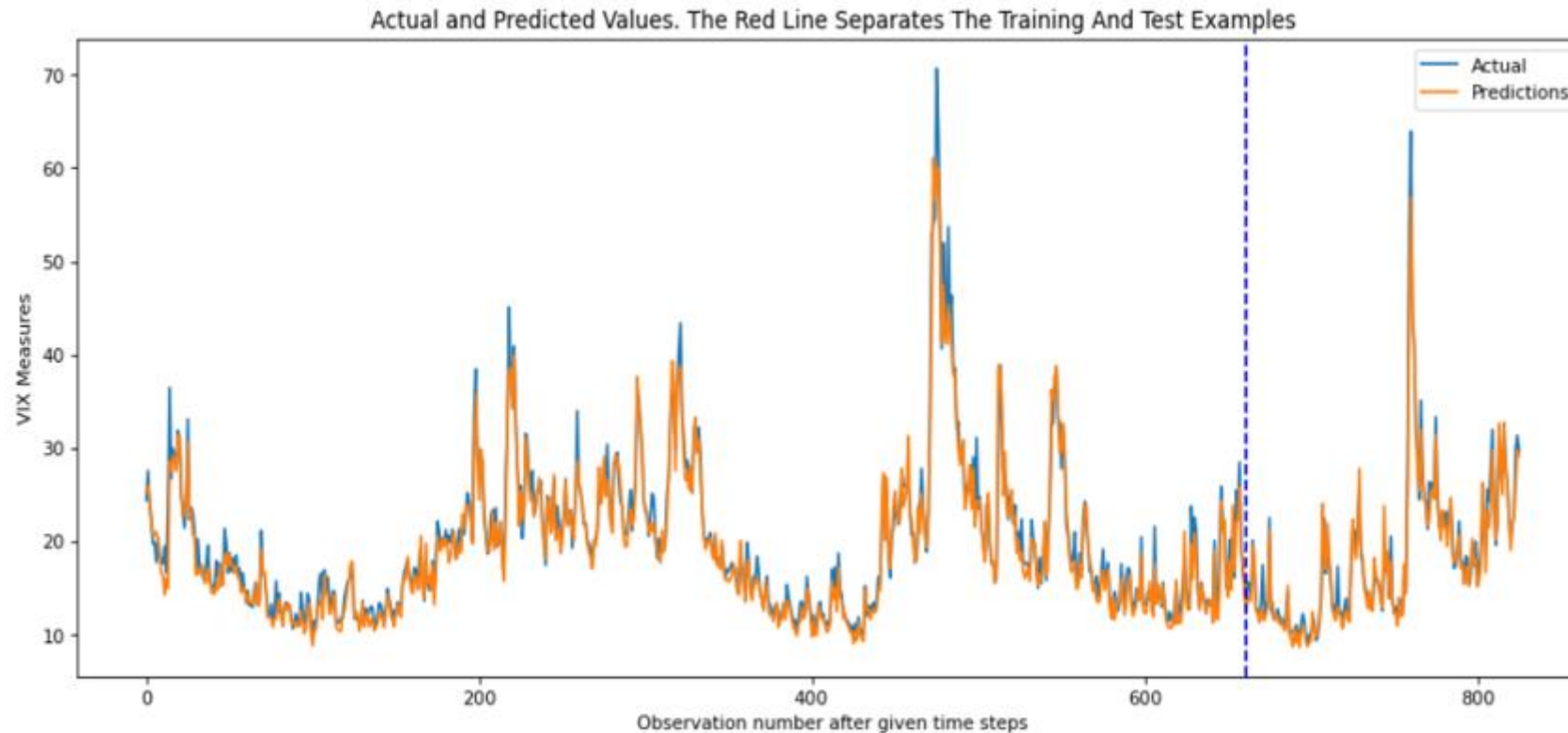
```
In [21]: def print_error(trainY, testY, train_predict, test_predict):  
    ### Error of predictions  
    train_rmse = math.sqrt(mean_squared_error(trainY, train_predict))  
    test_rmse = math.sqrt(mean_squared_error(testY, test_predict))  
    ### Print RMSE  
    print('Train RMSE: %.3f RMSE'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

Prediction versus Actual

Please upload Chapter8-1.ipynb and follow the computing steps in Jupyter Notebook

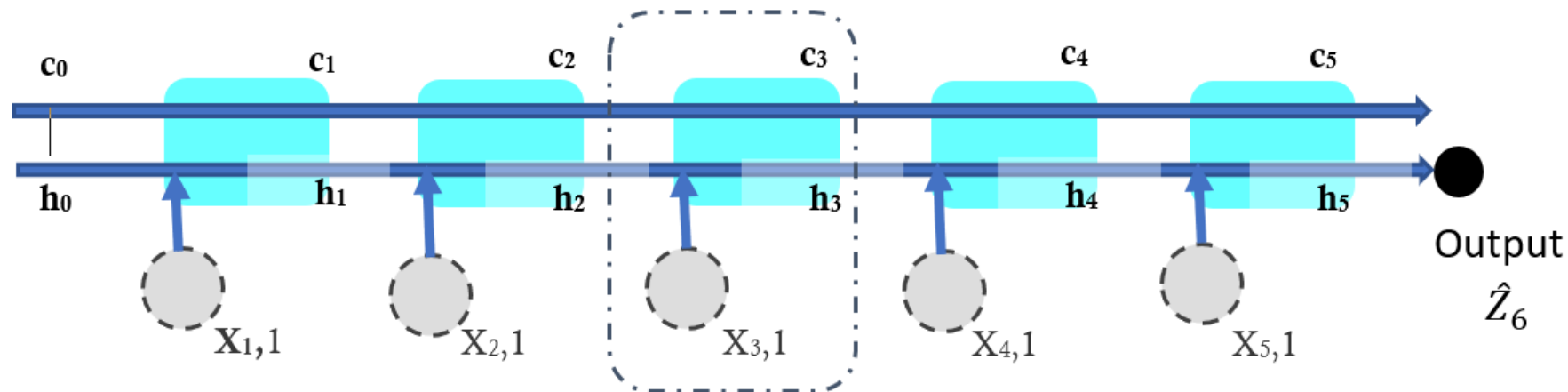
The RMSEs are about 2.3% for the training and 2.4% for the testing. The plot of the rescaled (inverse of scaling in [9]) actual versus predicted outputs in both the training set and the test set are shown below.



Variants of RNN

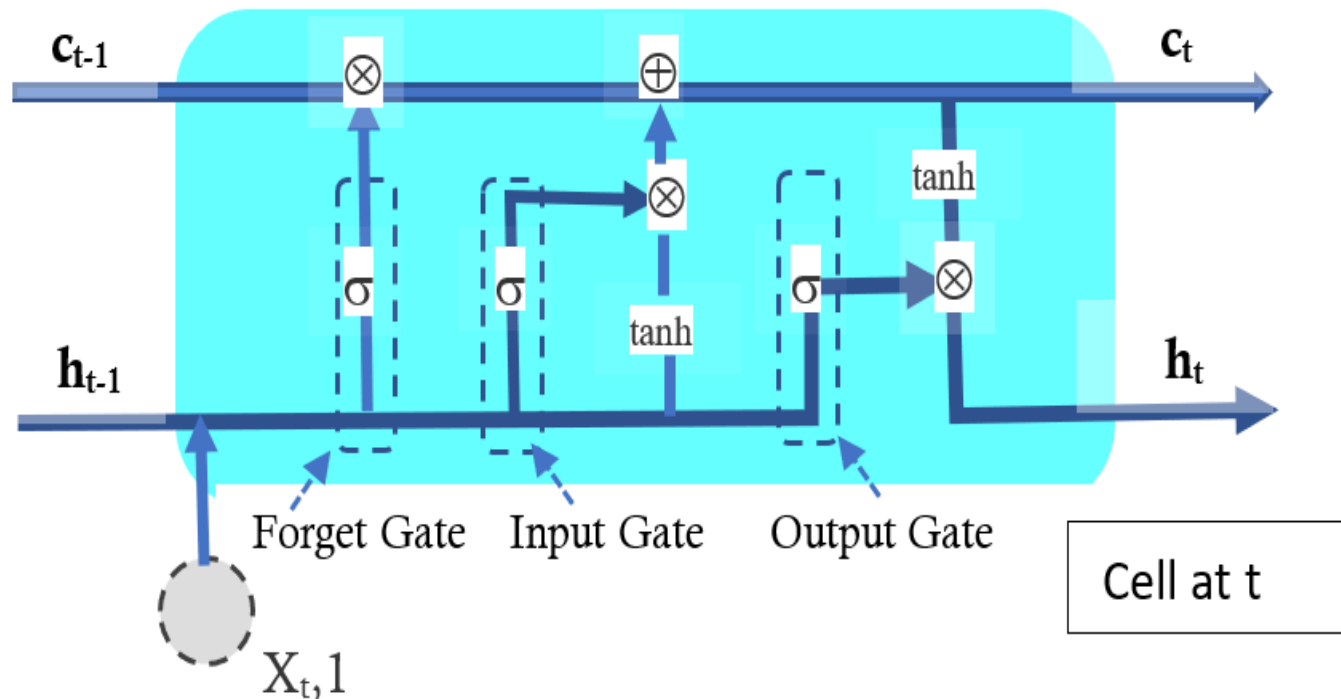
Mitigating lack of long term memory with disappearing gradients

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.



Variants of RNN

Long Short Term Memory NN



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.

Variants of RNN

Long Short Term Memory NN

- 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 σ -function, viz.

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

- In (2), input gate is multiplied by cell update candidate

$$\sigma (W_{IX} X_t + W_{Ih} \mathbf{h}_{t-1} + b_I) \otimes \tanh (W_{CX} X_t + W_{Ch} \mathbf{h}_{t-1} + b_C)$$

This Hadamard product is used to update cell state

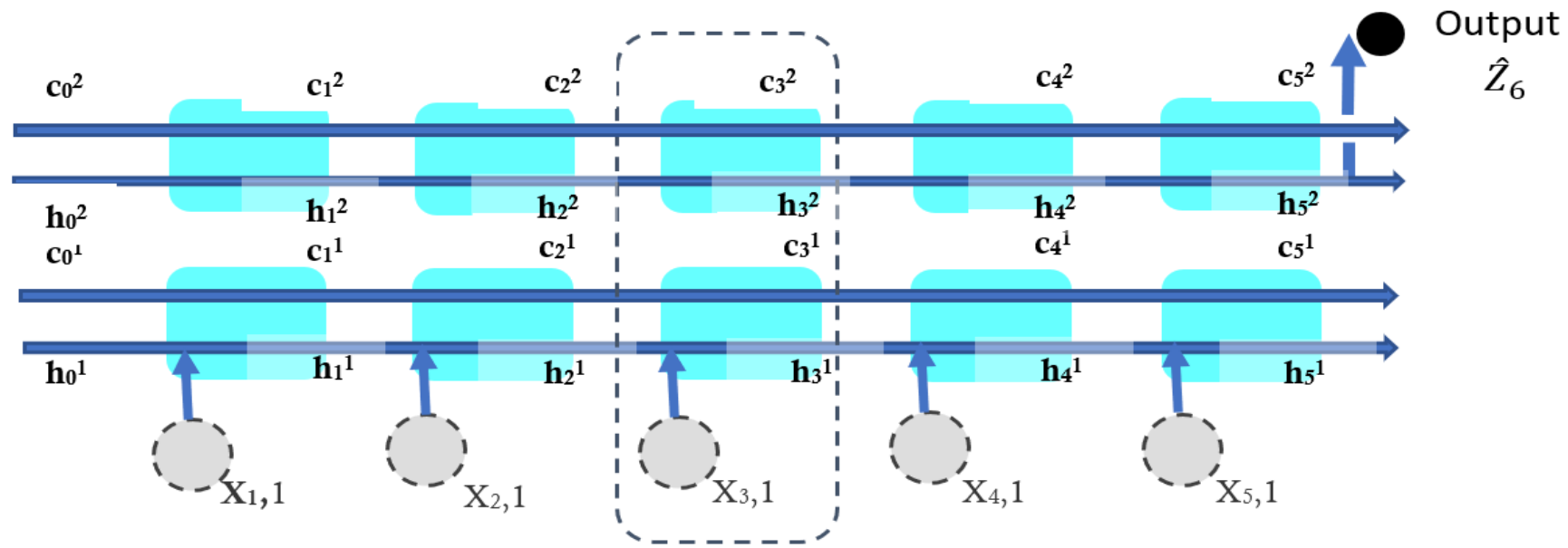
$$\begin{aligned} \mathbf{c}_t = & \sigma (W_{FX} X_t + W_{Fh} \mathbf{h}_{t-1} + b_F) \otimes \mathbf{c}_{t-1} \\ & + \sigma (W_{IX} X_t + W_{Ih} \mathbf{h}_{t-1} + b_I) \otimes \tanh (W_{CX} X_t + W_{Ch} \mathbf{h}_{t-1} + b_C) \end{aligned}$$

- In (3), $\mathbf{h}_t = \sigma (W_{OX} X_t + W_{Oh} \mathbf{h}_{t-1} + b_O) \otimes \tanh (\mathbf{c}_t)$

Variants of RNN

Stacked Long Short Term Memory NN

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).



Variants of RNN

Stacked Long Short Term Memory NN

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

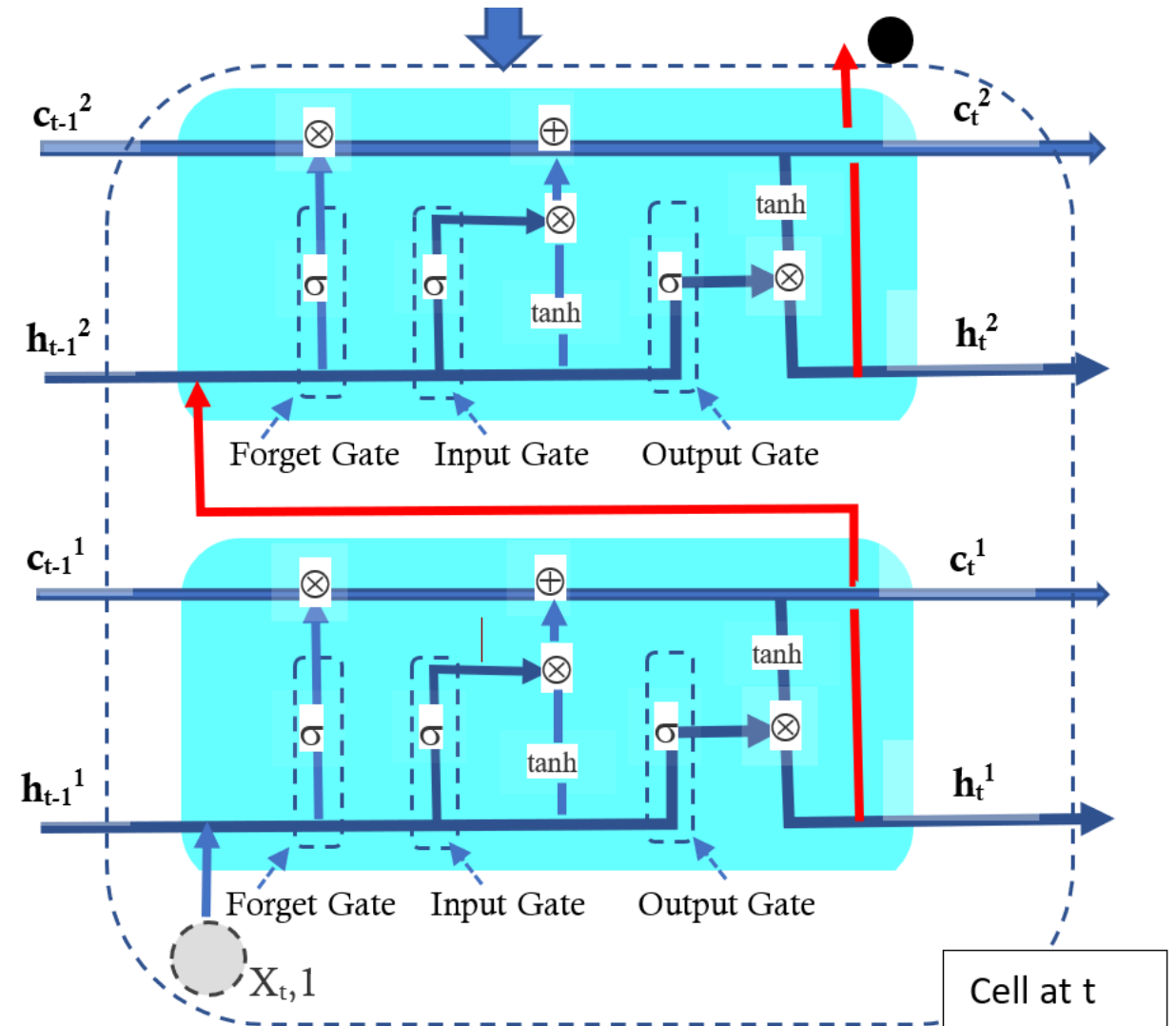
$$\mathbf{h}_t^1 = \sigma(W_{OX}^1 X_t + W_{Oh}^1 \mathbf{h}_{t-1}^1 + b_O^1) \otimes \tanh(\mathbf{c}_t^1).$$

Forget Gate output fraction vector

$$\sigma(W_{FX}^2 \mathbf{h}_t^1 + W_{Fh}^2 \mathbf{h}_{t-1}^2 + b_F^2)$$

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.

$$\mathbf{h}_t^2 = \sigma(W_{OX}^2 \mathbf{h}_t^1 + W_{Oh}^2 \mathbf{h}_{t-1}^2 + b_O^2) \otimes \tanh(\mathbf{c}_t^2)$$



Worked Example II -- Data

Please upload Chapter8-2.ipynb and follow the computing steps in Jupyter Notebook

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.

```
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



Please upload Chapter8-2.ipynb and follow the computing steps in Jupyter Notebook

The data structure is created to feed the inputs to the NN. $T = 1100$ 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 (overlapping cases) 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,)
```

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.

Stacked LSTM

Please upload Chapter8-2.ipynb and follow the computing steps in Jupyter Notebook

```
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))
```

```
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')
```


Fitting and Prediction

Please upload Chapter8-2.ipynb and follow the computing steps in Jupyter Notebook

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

```
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
```

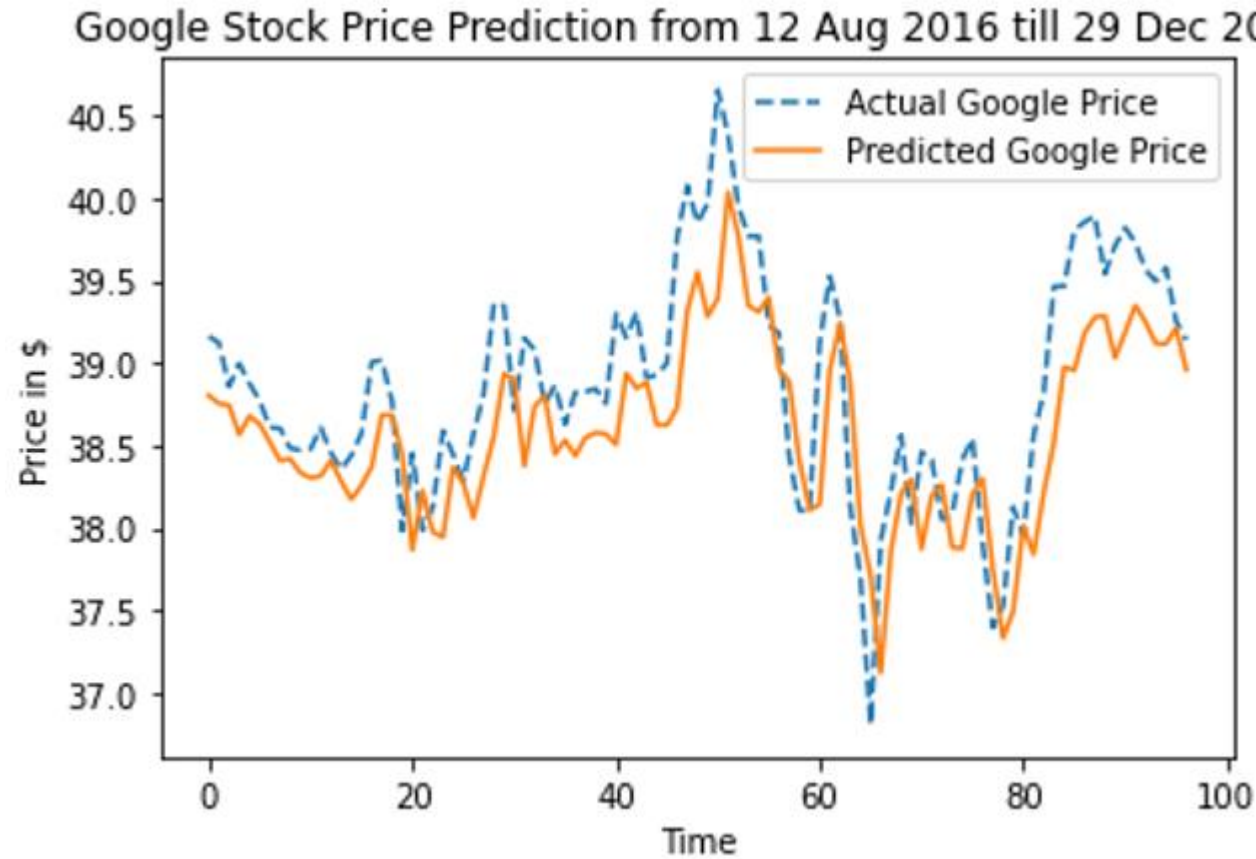
```
In [25]: ### Prediction  
predicted_stock_price=model.predict(X_test)  
predicted_stock_price=predicted_stock_price[:,0]  
#predicted_stock_price1=sc.inverse_transform(predicted_stock_price)  
#predicted_stock_price.shape
```

```
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
```

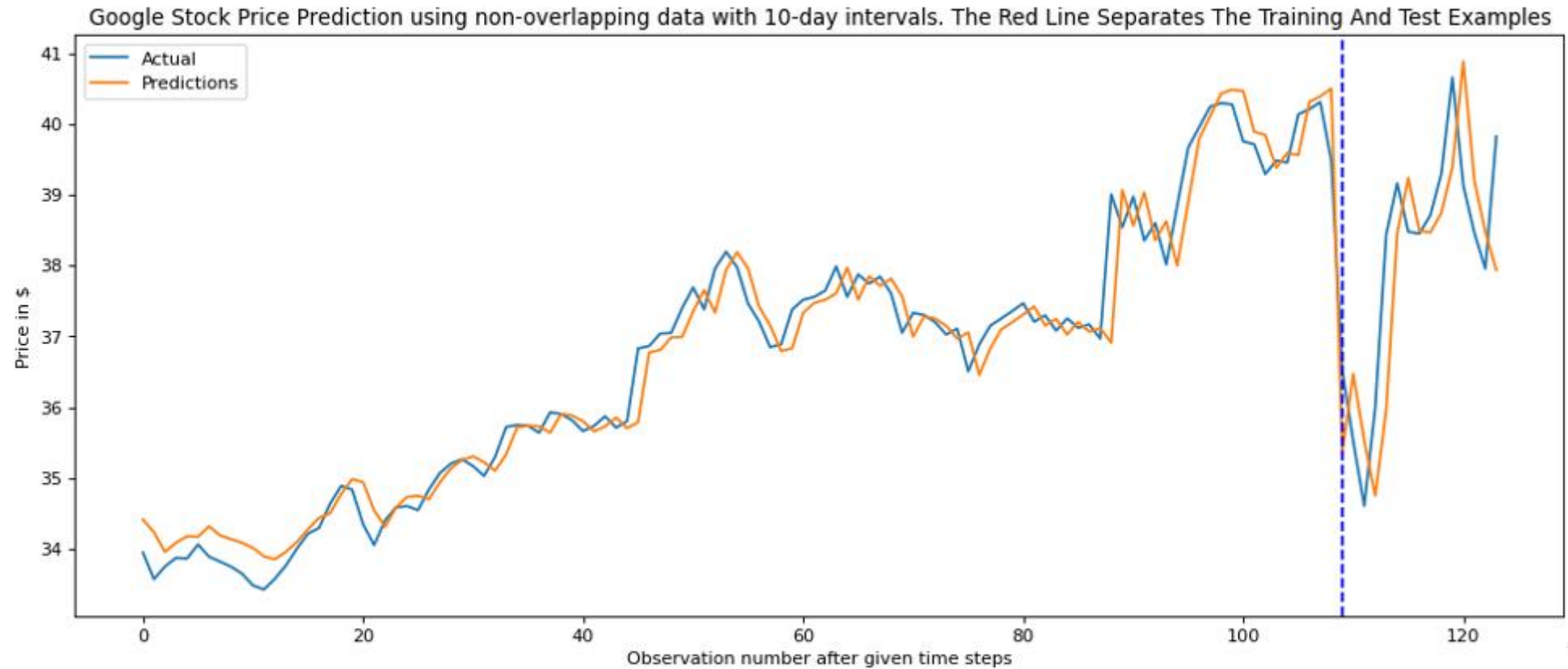
Overlapping Cases

Please upload Chapter8-2.ipynb and follow the computing steps in Jupyter Notebook



Non-overlapping Cases

Please upload Chapter8-3.ipynb and follow the computing steps in Jupyter Notebook



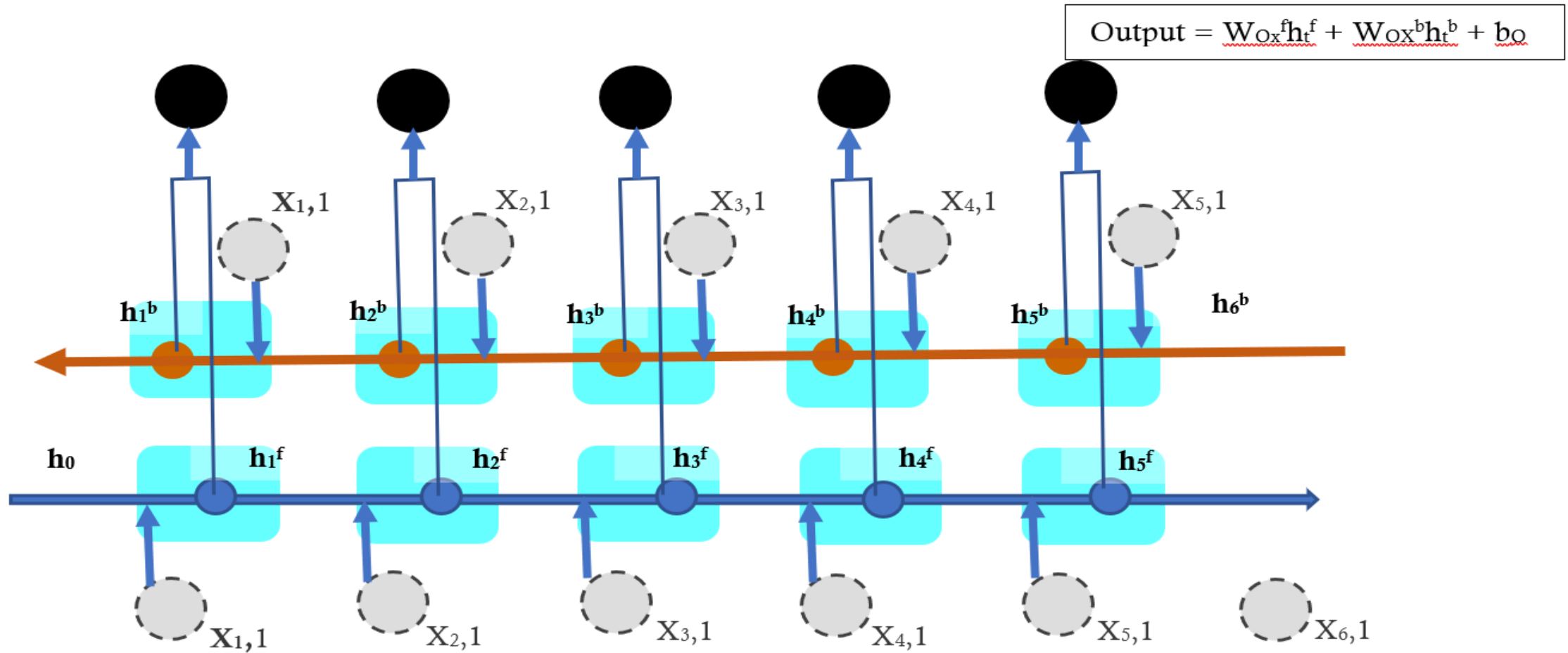
CAVEATS

- 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 prediction to something economic/downright market finance such as trading – can you make a steady/consistent profit?
 - 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)

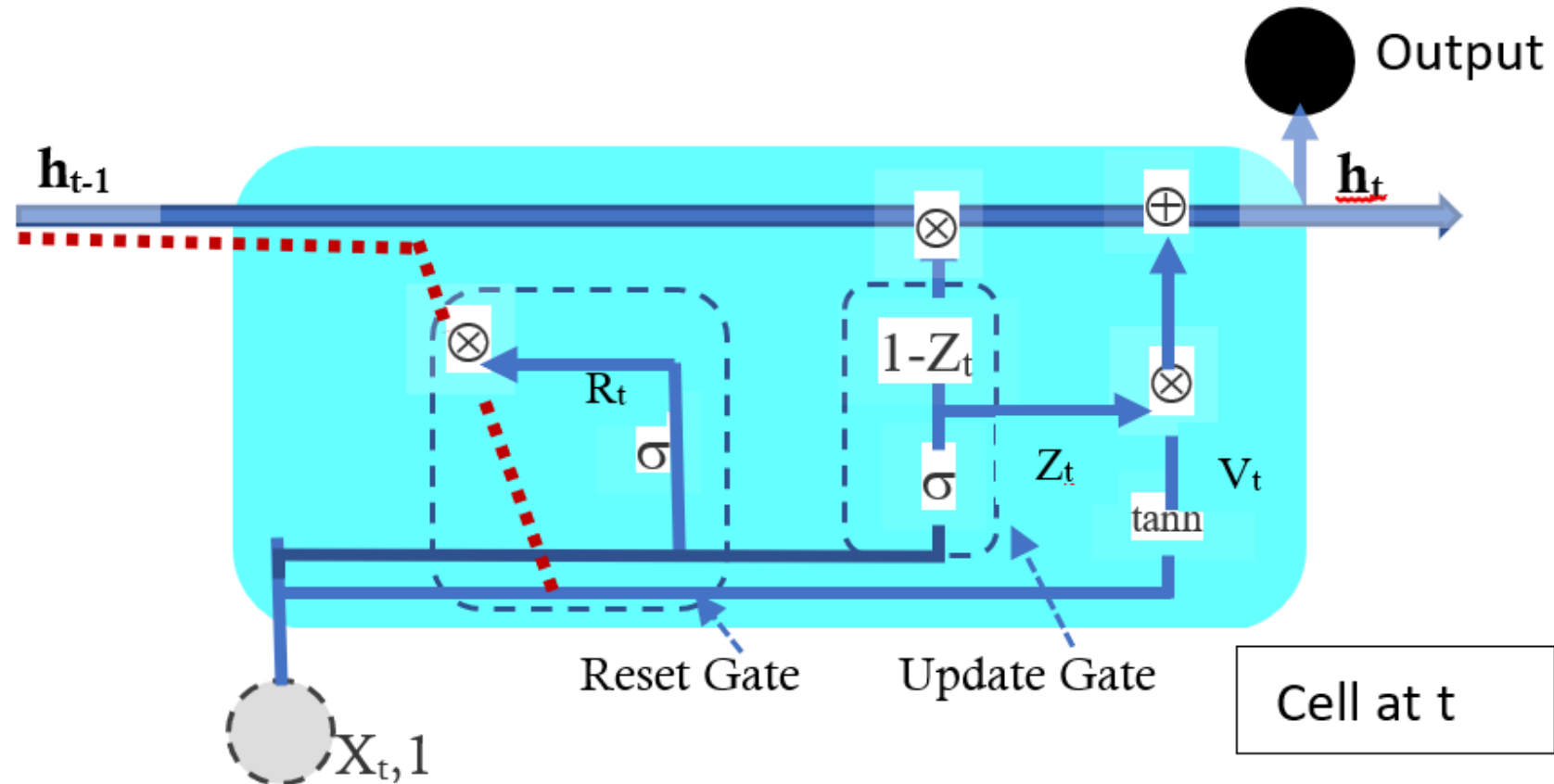
CAVEATS

- 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.

Bidirectional RNN/LSTM



Gated Recurrent Unit (GRU)



In-Class Practice Exercise (not graded):

Chapter8-4.ipynb

Use the VIX price data for prediction – VIX.csv

Construct a stacked LSTM NN with 50 neurons in each stacked layer, and 4 layers, plus an output layer. Use batch size = 10 and # epochs = 100. Here time-steps = 60. This is an input in the `input_shape=(X_train.shape[1],1)` of Keras LSTM App.

Use Adam optimizer, and mean-squared-error loss.

Use Overlapping training cases.

Find the training and the test prediction RMSEs.

End of Class