**Experiment 5**

**Name: SAP ID:**

|  |  |
| --- | --- |
| **Date:** |  |
| Aim | **Implementing Basic RNN: Develop an RNN model for stock price prediction using historical data.** |
| Software | Colab |
| Pre-requisite | Internet and required dataset |
| Theory | RNN:  Recurrent neural networks (RNN) are a class of neural networks that is powerful for modeling sequence data such as time series or natural language. We have used it in predicting stock prices. The logic behind this is that it will remember the price after a particular sequence and the model will gain experience based on that pattern. Schematically, RNN layer uses a for loop to iterate over the timesteps of a sequence, while maintaining an internal state that encodes information about the timesteps it has seen so far. RNN can retain sequence patterns only for a short time thus, we move to LSTM that can remember patterns in Long and Short Term memory.  To get a detailed view of RNN and LSTM, click on the image below:  [[https://miro.medium.com/v2/resize:fit:516/1*HZTEsWqk4b4XUDI0FDnv_Q.png](https://purnasaigudikandula.medium.com/recurrent-neural-networks-and-lstm-explained-7f51c7f6bbb9)](https://purnasaigudikandula.medium.com/recurrent-neural-networks-and-lstm-explained-7f51c7f6bbb9)  **Data Collection & Preprocessing:**  The first thing that we need to do is install yfinance. If you haven’t already installed it yet.  pip install yfinance  Import the yfinance module for collecting data of a particular stock. For our case, we will use Escorts Ltd.  #importing yfinance import yfinance as yf#Collecting data data = yf.download('ESCORTS.NS',period='5y',interval='1d')  Parameters required for yfinance.download() are ticker which is the symbol for your stock; period: total duration for which you want data to be extracted, and interval: which refers to the consecutive records e.g. for 1 day put ‘1d’. This will extract the stock’s data of 5 years for each day excluding the holidays.  Import the following libraries:  import pandas as pd import numpy as np import tensorflow as tf import matplotlib.pyplot as plt %matplotlib inline  Now that we have collected the data, we need to select the column which is required. Data contains stock’s historical data on Open, Close, Low, High, Volume, and Adjusted Close. We will be using adjusted close for pattern detection and prediction. Also, split the data into train and test set so that we can evaluate our model later.  data\_target = data.iloc[:1182,4] data\_test = data.iloc[1132:,4] steps = 7#return numpy representation of data data = data.loc[:,["Adj Close"]].values test = data[len(data) - len(data\_test) - steps:]#4 the column is Adj Close  Let’s check the trend by visualizing it.  plot = data\_target.plot()  efore, proceeding further we need to define some functions for Scaling Down data and converting data into a set of patterns followed for a particular price.  For scaling down, we will use MinMaxScaler available under ScikitLearn.  #Scaling Dataset def scaledata(data\_target): #Import scaler and initialise it  from sklearn.preprocessing import MinMaxScaler  scaler = MinMaxScaler(feature\_range=(0,1))  #transform by converting it to array and shape of (-1,1)  data\_target\_scaled = scaler.fit\_transform(np.array(data\_target).reshape(-1,1))  #plot the scaled version of data  plot\_scaled = pd.DataFrame(data\_target\_scaled).plot()  print(data\_target.shape) #returns scaled data  return data\_target\_scaled, scaler  Before proceeding to the next function, let’s understand why is it required. By now, we know that RNN retains the pattern for example if you wear red on Sunday, blue on Monday, and green on Tuesday and then repeats it, RNN can retain that pattern for a short time. It can predict that you will wear blue tomorrow if you wear a red t-shirt today. Thus, data must have a pattern to be recognized.  Now, we will build a function that will convert the data into patterns of prices, and the target price achieved after that pattern follows. This way, our model can learn the response of the price patterns.  #Create pattern and end price set def createPatternSet(data\_target\_scaled,steps=7):   x\_patern = [] #Independent Variable  y\_price = [] #Dependent Variable for day in range(steps,data\_target\_scaled.shape[0]):  row = data\_target\_scaled[day-steps:day,0]  #print(len(row))  x\_patern.append(row)  y = data\_target\_scaled[day,0]  #print(y)  y\_price.append(y)    x\_patern,y\_price = np.array(x\_patern),np.array(y\_price)  #RNN and LSTM takes 3D inputs, we need to change the shape of array to 3 dimensional. x\_patern = x\_patern.reshape(x\_patern.shape[0],x\_patern.shape[1],1) #returns independent and dependent variable sets  return x\_patern,y\_price  The above function takes data to be converted and the number of steps as steps. By default, we set steps equal to 7 which means that the 7-day pattern and the Price after that will be recorded as independent and dependent variables respectively.  #Scale Down Target data\_target\_scaled = scaledata(data\_target)[0] scaler = scaledata(data\_target)[1] #prepare test data test = data[len(data) - len(data\_test) - steps:] test = scaler.transform(test)  https://miro.medium.com/v2/resize:fit:375/1*ff2PoRG2N46G02TTMyznEQ.png  efore, proceeding further we need to define some functions for Scaling Down data and converting data into a set of patterns followed for a particular price.  For scaling down, we will use MinMaxScaler available under ScikitLearn.  #Scaling Dataset def scaledata(data\_target): #Import scaler and initialise it  from sklearn.preprocessing import MinMaxScaler  scaler = MinMaxScaler(feature\_range=(0,1))  #transform by converting it to array and shape of (-1,1)  data\_target\_scaled = scaler.fit\_transform(np.array(data\_target).reshape(-1,1))  #plot the scaled version of data  plot\_scaled = pd.DataFrame(data\_target\_scaled).plot()  print(data\_target.shape) #returns scaled data  return data\_target\_scaled, scaler  Before proceeding to the next function, let’s understand why is it required. By now, we know that RNN retains the pattern for example if you wear red on Sunday, blue on Monday, and green on Tuesday and then repeats it, RNN can retain that pattern for a short time. It can predict that you will wear blue tomorrow if you wear a red t-shirt today. Thus, data must have a pattern to be recognized.  Now, we will build a function that will convert the data into patterns of prices, and the target price achieved after that pattern follows. This way, our model can learn the response of the price patterns.  #Create pattern and end price set def createPatternSet(data\_target\_scaled,steps=7):   x\_patern = [] #Independent Variable  y\_price = [] #Dependent Variable for day in range(steps,data\_target\_scaled.shape[0]):  row = data\_target\_scaled[day-steps:day,0]  #print(len(row))  x\_patern.append(row)  y = data\_target\_scaled[day,0]  #print(y)  y\_price.append(y)    x\_patern,y\_price = np.array(x\_patern),np.array(y\_price)  #RNN and LSTM takes 3D inputs, we need to change the shape of array to 3 dimensional. x\_patern = x\_patern.reshape(x\_patern.shape[0],x\_patern.shape[1],1) #returns independent and dependent variable sets  return x\_patern,y\_price  The above function takes data to be converted and the number of steps as steps. By default, we set steps equal to 7 which means that the 7-day pattern and the Price after that will be recorded as independent and dependent variables respectively.  #Scale Down Target data\_target\_scaled = scaledata(data\_target)[0] scaler = scaledata(data\_target)[1] #prepare test data test = data[len(data) - len(data\_test) - steps:] test = scaler.transform(test)  **Train and Test Set:**  We will use our function to process and build x\_train and y\_train.  #Overwrite steps to 50. it doesnt really matter here because we will be doing a lot of iterations with it (Take anyhthing less than 100). train\_pattern = createPatternSet(data\_target\_scaled,steps=50) x\_train = train\_pattern[0] y\_train = train\_pattern[1]#Input Shape needs to be 3D. x\_train.shape >>> (1132, 50, 1)  We finished building our train set and we will build the test set.  #create pattern and price for test set. test\_pattern = createPatternSet(test,steps=50) x\_test = test\_pattern[0] y\_test = test\_pattern[1]#Dont forget to check the shape of x\_test (3D reuired). x\_test.shape  Test data will be used for the model’s evaluation. We will predict the values based on the x\_test and then compare them to the original y\_test value.  So, far we have completed data preprocessing, and now the interesting part**.**  **Model Architecture:**  We will define a class with methods that can build the architecture, compile it, and fit it on given data. The class will also have methods to set parameters like the number of neurons, batch\_size, and epoch. The reason to build this class is that we can run a for loop with the different parameters passed and analyze the result. This will help the readers in trying other combinations of hyperparameters.  Later on, we can also inherit from this class for LSTM and overwrite the architecture building method. There are other options available for hyper tuning the model but we stick to this for the most basic understanding.  Build the Class for RNN:   |  | | --- | | def buildArchitecture(self,rnn=2,dense=1): | |  | StocksPriceRNN.model = tf.keras.Sequential() | |  | StocksPriceRNN.model.add(tf.keras.layers.SimpleRNN(StocksPriceRNN.neurons, | |  | activation='tanh', | |  | return\_sequences = True, | |  | input\_shape = (self.x\_train.shape[1],1))) | |  | StocksPriceRNN.model.add(tf.keras.layers.Dropout(0.2)) | |  | for i in range(rnn): | |  | StocksPriceRNN.model.add(tf.keras.layers.SimpleRNN(StocksPriceRNN.neurons, | |  | activation='tanh', | |  | return\_sequences = True)) | |  | StocksPriceRNN.model.add(tf.keras.layers.Dropout(0.2)) | |  |  | |  | #return sequense changed to false | |  | StocksPriceRNN.model.add(tf.keras.layers.SimpleRNN(StocksPriceRNN.neurons, | |  | activation='tanh', | |  | return\_sequences = False)) | |  | StocksPriceRNN.model.add(tf.keras.layers.Dropout(0.2)) | |  |  | |  | for i in range(dense): | |  | StocksPriceRNN.model.add(tf.keras.layers.Dense(units=StocksPriceRNN.neurons, | |  | activation='tanh')) | |  |  | |  | #Output | |  | StocksPriceRNN.model.add(tf.keras.layers.Dense(units=1)) | |  | return StocksPriceRNN.model.summary() | |  |  | |  | def compiler(self): | |  | opt= tf.keras.optimizers.Adam() | |  | StocksPriceRNN.model.compile(optimizer = opt, | |  | loss = StocksPriceRNN.loss) | |  | return StocksPriceRNN.model.summary() | |  |  | |  | def modelfit(self): | |  | history = StocksPriceRNN.model.fit(self.x\_train,self.y\_train, | |  | epochs=self.epoch,batch\_size=StocksPriceRNN.batch\_size,validation\_split=0.2, | |  | ) | |  | return history | |  |  | |  | def changeBatchSize(self,size): | |  | StocksPriceRNN.batch\_size = size | |  | print("Changed!") | |  | def changeNeurons(self,size): | |  | StocksPriceRNN.neurons = size | |  | print("Changed!") | |  | def changeEpoch(self,size): | |  | self.epoch = size | |  | print("Changed!") |   By default the model’s architecture contains the following layers:  Layer (type) Output Shape Param #  ================================================================= simple\_rnn\_44 (SimpleRNN) (None, 50, 50) 2600  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ dropout\_32 (Dropout) (None, 50, 50) 0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ simple\_rnn\_45 (SimpleRNN) (None, 50, 50) 5050  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ dropout\_33 (Dropout) (None, 50, 50) 0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ simple\_rnn\_46 (SimpleRNN) (None, 50, 50) 5050  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ dropout\_34 (Dropout) (None, 50, 50) 0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ simple\_rnn\_47 (SimpleRNN) (None, 50) 5050  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ dropout\_35 (Dropout) (None, 50) 0  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ dense\_19 (Dense) (None, 1) 51  ================================================================= Total params: 17,801 Trainable params: 17,801 Non-trainable params: 0  **Iterations & Evaluations:**  First, we will iterate on RNN and then on LSTM. The best output of both will be compared for the evaluation.  Build a For-Loop statement for passing different epochs and batch sizes. Run the model on the data and visualize the output:  for steps in [7,30,90]:  for epoch in [20,30,50]:  #prepare train data  train\_pattern = createPatternSet(data\_target\_scaled,steps=steps)  #prepare test data  test = data[len(data) - len(data\_test) - steps:]  test = scaler.transform(test) test\_pattern = createPatternSet(inputs,steps=steps)  x\_test = test\_pattern[0]  y\_test = test\_pattern[1]  #Build Model  RNN1 = StocksPriceRNN(x\_train,y\_train,epoch)  RNN1.buildArchitecture(2,0)  RNN1.compiler()  #fit model  history = RNN1.modelfit()  #Predict Values  pred = RNN1.model.predict(x=x\_test)  output = scaler.inverse\_transform(pred)  org\_vals = scaler.inverse\_transform(y\_test)  #visualise  print("Plotting for Steps {} and Epoch {}".format(steps,epoch))  plotting(org\_vals,output)  This will output in 9 iterations. After comparing the iterations, I found that RNN gives sloppy but comparatively best results on a combination of 90–30 and 90–50 (steps-epoch).  Here is the Output:  https://miro.medium.com/v2/resize:fit:700/1*Bweld_eLgriNMYtoDICW6Q.png  Not much difference between 30–50 epoch but accuracy can increase on the increase of epochs.  Similarly, we run iterations for LSTM: Please note that there will be some warnings related to “out of calls” and this is natural as we have split the test data with 100 records only. To avoid this, Make sure that experimental\_relax\_shapes=True, an option that relaxes argument shapes that can avoid unnecessary retracing (not necessarily required).  # for different epochs, batch size, and neurons/units.  for epch in [60,100,200]:  for batch in [2,4,6]:  for neurons in [8,10,12]:  LSTM2 = LstmModel(x\_train,y\_train,epoch=epch)  LSTM2.changeBatchSize(batch)  LSTM2.changeNeurons(neurons) LSTM2.buildArchitecture()  LSTM2.compiler()  history = LSTM2.modelfit() pred = LSTM2.model.predict(x\_test)  pred = scaler.inverse\_transform(pred)  #org = scaler.inverse\_transform(y\_test) print("For epch {}, neurons {} and batch {}".format(epch,neurons,batch))  plotting(org,pred)  The output will contain 27 iterations.  Analyzing the output of LSTM, it is clear that LSTM performs better on the dataset than the RNN. LSTM gave better results at batch\_size = 2, units = 10 and epoch = 200.  https://miro.medium.com/v2/resize:fit:700/1*LyATHGlVodC5V8f1JEPO1g.png |
| Result | Link: <https://www.kaggle.com/rishirajak/share-price-prediciton-using-lstm-and-rnn> |
| Conclusion | Write your detail understanding about the experiment. |