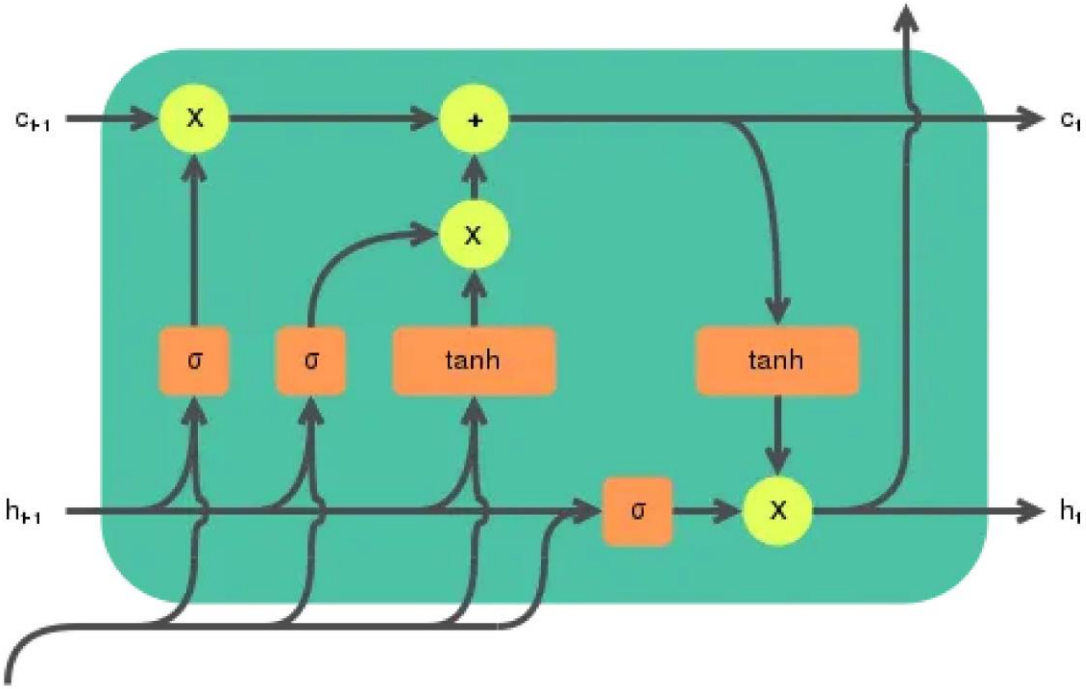
Share Price Prediction

In this project we are going to build a deep learning model using techniques like RNN model and LSTM to predict share price trends. We will also check which model is more suitable for prediction of share price trends.

RNN : Recurrent Neural Networks is a class of Neural Network. It is powerful Neural Network used for prediction of Time Series Analysis and Natural Language Processing. In this project RNN is used for predicting share prices of stock. Here model will remember the price after particular sequence and gain experience. RNN iterates over the timesteps of sequence. Limitation of RNN is it retains the sequence only for short time.

LSTM : Long Short Term Memory is an artificial RNN. It is preferred over Feed Forward Neural Network because it can remember data points over longer period of time. LSTM consists of a cell, input gate, output gate and forget gate.



1. **Data Collection and Preprocessing**

   In this project we will be using yahoo finance data of ESCORTS Ltd, for that we need to install yfinance first.

```
pip install yfinance
```

Import yfinance module and download ESCORT Ltd data from yfinance for 5 years daily iterval. Parameters for download are ticker = ESCORTS.NS, period=5y, and interval=1d

```
#importing yfinance
import yfinance as yf

#Collecting data
data = yf.download('ESCORTS.NS',period='5y',interval='1d')
```

Then import required libararies

```
import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
%matplotlib inline
```

Our data has following headers

```
[ ]  data.head(3)
```

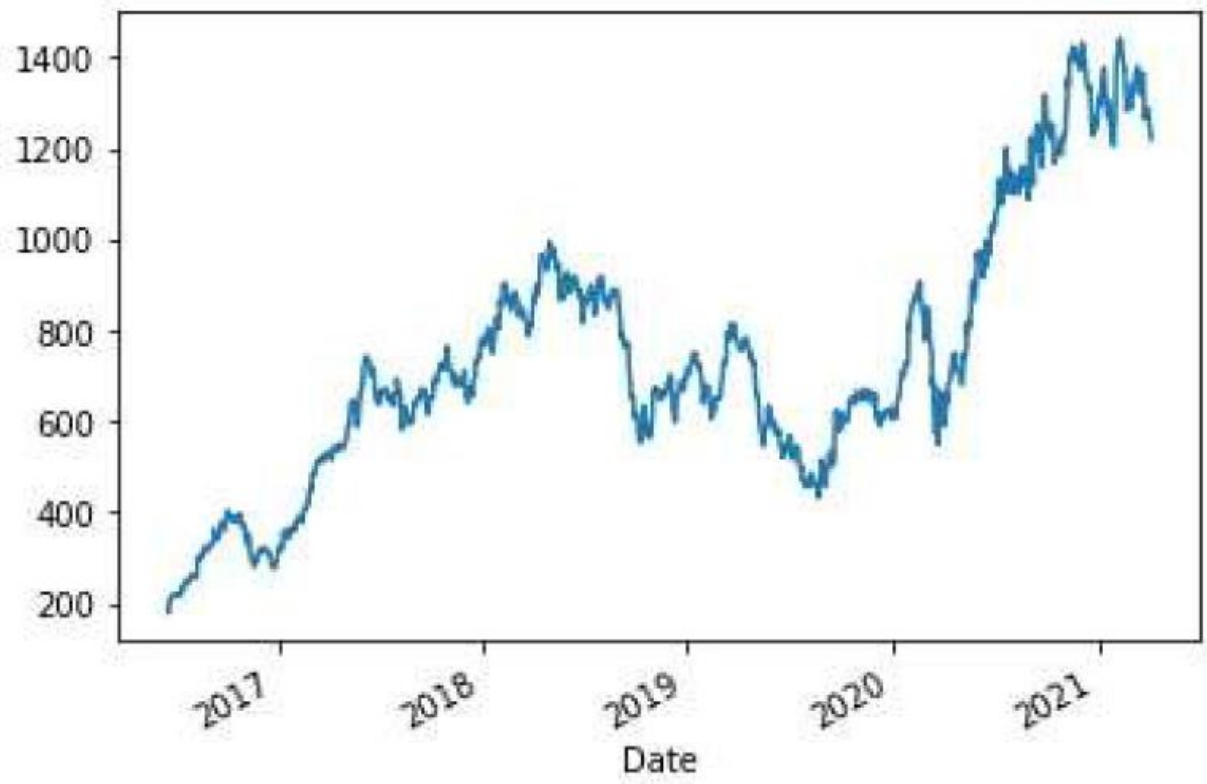
Date	Open	High	Low	Close	Adj Close	Volume
2018-10-17	645.000000	649.599976	615.250000	617.950012	607.538269	2445573
2018-10-19	611.099976	616.000000	583.500000	604.349976	594.167297	1719936
2018-10-22	607.000000	613.799988	570.450012	573.349976	563.689697	1593082

Now split the data into Training and Test Sets

```
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:]
```

Visualizing test set data to check the trends

```
plot = data_target.plot()
```



.

Now we will scale down data and convert data for a particular stock into set of patterns. We will define a function scaling down the data.

```
#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
```

Function for converting data into patterns of prices and Target price achieved after that pattern follows. In this wat 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)

     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
```

As RNN and LSTM takes 3D inputs, we need to change the shape of array to 3D.

Here we have set steps = 7 it means 7 day pattern and price after that is 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)
```

We can build train and test set by changing steps = 50.

```
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)
```

```
#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
```

2. **Model Archietecture**

   RNN

   We will build the class for RNN and run a for loop with different parameters (no of neurons, batch_size, epochs). This will help us to run the model with different parameters and experiment it with.

```
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,
                                      )
    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!")
```

Model architecture contains 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
```

In this model we have used Adm optimizers with mean_sqaured_error loss function

3. **Model Archietecture**

LSTM

```
class LstmModel(StocksPriceRNN):
    StocksPriceRNN.model = tf.keras.Sequential()
    def __init__(self,x_train,y_train,epoch):
        super().__init__(x_train,y_train,epoch)

    def buildArchitecture(self,dense=1):
        StocksPriceRNN.model = tf.keras.Sequential()
        StocksPriceRNN.model.add(tf.keras.layers.LSTM(
                                 StocksPriceRNN.neurons,
                                 input_shape=(None,1)))
        #Output
        StocksPriceRNN.model.add(tf.keras.layers.Dense(units=1))
        return StocksPriceRNN.model.summary()
```

4. **Visualizing Results**

```
def plotting(org_vals,output):
    plt.figure(figsize=(10,5), dpi=80, facecolor='w', edgecolor='k')
    plt.plot(org_vals,color="Green",label="Org value")
    plt.plot(output,color="Yellow",label="Predicted")
    plt.legend()
    plt.xlabel("Days")
    plt.ylabel("Price")
    plt.grid(True)
    plt.show()
```

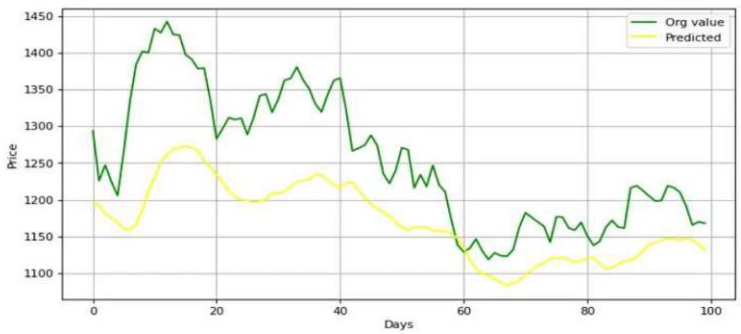
5. **Model Evaluation**

RNN

Now we passing different parameters to our RNN and LSTM model and iterate using for loop. The output from both the models will be compared and evaluated.

```
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)
```

By experimenting it was found that RNN gives best result for steps = 90 and epoch = 30. Here is the output visualization

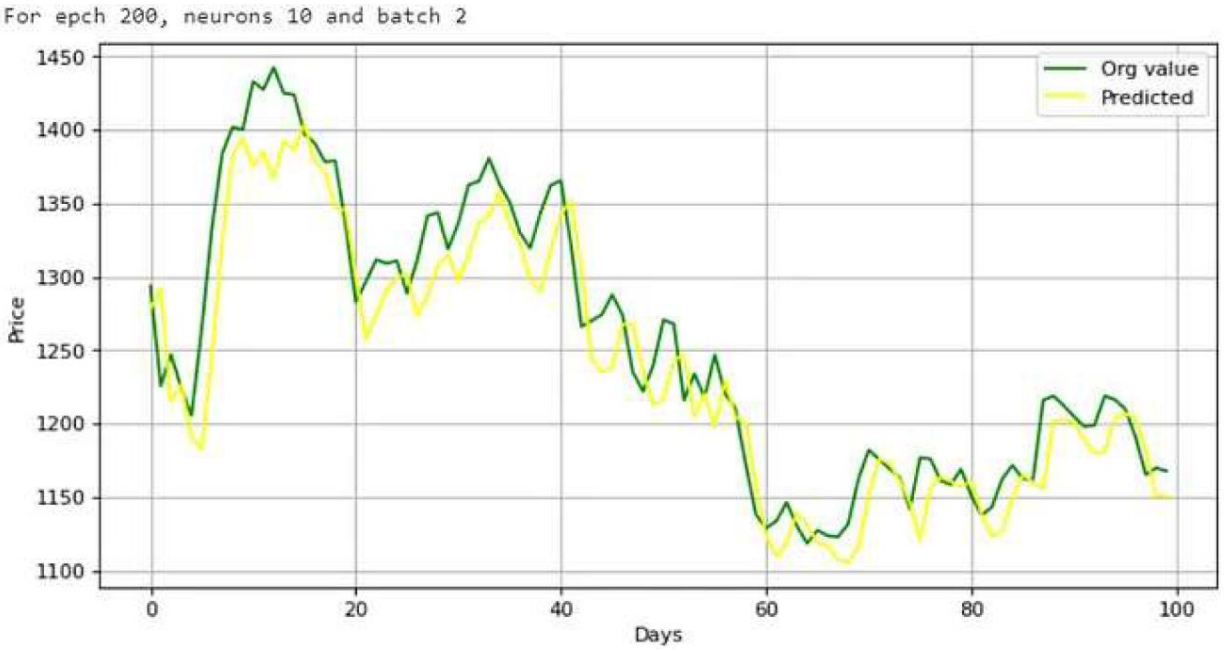


6. **Model Evaluation**

LSTM

```
# 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)
```

Analyzing result, it is clear that LSTM performs better on dataset than RNN at batch_size = 2, units = 10 and epoch = 200.



7. **Conclusion**

The model is capable of lot more iterations. We can try out with different hyper parameters and get closure to the original price curve.