**PACKAGES AND LIBRARIES**

import pandas as pd

import numpy as np

import math

import matplotlib.pyplot as plt

import seaborn as sns

sns.set\_theme(style="whitegrid")

plt.style.use('ggplot')

import tensorflow as tf

import warnings

import os

import datetime as dt

import tensorflow

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM

from tensorflow.keras.layers import Dense

from tensorflow.keras.layers import Dropout

from tensorflow import keras

from tensorflow.keras import optimizers

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error, r2\_score

import numpy as np

import time

import math

from keras.optimizers import Adam,Adagrad

import mealpy

mealpy.\_\_version\_\_

from mealpy import FloatVar, ARO, StringVar, IntegerVar

**DATA PREPROCESSING**

data = pd.read\_csv("D:\Final Year Project\Dataset\Proposed Paper Dataset\S&P 500\gspc\_final.csv")

data.head()

data.drop(data.index[:49], inplace=True)

data.reset\_index(drop=True, inplace=True) # Reset index after dropping rows

data['Date'] = pd.to\_datetime(data['Date'], format='%d-%m-%Y')

data['Date'] = data['Date'].dt.strftime('%Y-%m-%d')

data.set\_index("Date",inplace=True)

data = data.fillna(data.mean())

data.isna().sum()

len(data) # 4397

**#Correlation heatmap**

fig = plt.figure(figsize= (10,5))

sns.heatmap(data.corr(), annot=True)

sns.set\_style("whitegrid")

plt.show()

data = data.iloc[:, 1:]

fig = plt.figure(figsize= (10,5))

sns.heatmap(data.corr(), annot=True)

sns.set\_style("whitegrid")

plt.show()

**#Denoising code**

from skimage.restoration import (denoise\_wavelet, estimate\_sigma)

data['Close']= denoise\_wavelet(data.iloc[:, 0], wavelet='haar',

#method='BayesShrink',

method='VisuShrink',

mode='soft', rescale\_sigma = True)

data.head()

**EVALUATION METRIC**

def mean\_absolute\_percentage\_error(y\_true, y\_pred):

return (np.mean(np.abs((y\_true - y\_pred)/(y\_true))\*100))

def calculate\_scores(y\_true, y\_pred):

mae = mean\_absolute\_error(y\_true, y\_pred)

mape = mean\_absolute\_percentage\_error(y\_true, y\_pred)

mse = mean\_squared\_error(y\_true, y\_pred)

rmse = np.sqrt(mse)

R = np.corrcoef(y\_true, y\_pred)

r2 = r2\_score(y\_true, y\_pred)

print(f"Mean Absolute Error (MAE): {mae:.4f}")

print(f"Mean Absolute Percentage Error (MAPE): {mape:.4f}")

print(f"Mean Squared Error (MSE): {mse:.4f}")

print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")

print(f"R-Pearson Correlation Coefficient (R): {R[0,1]:.4f}")

print(f"R-squared (R2): {r2:.4f}")

dic = {'rmse':rmse, 'R': R[0,1], 'mape': mape}

return (dic)

**NECESSARY FUNCTIONS**

def DatasetCreation(dataset, time\_step = 1):

DataX, DataY = [], []

for i in range(len(dataset)- time\_step -1):

a = dataset[i:(i+ time\_step), ]

DataX.append(a)

DataY.append(dataset[i + time\_step, 0])

return np.array(DataX), np.array(DataY)

def data\_split(data, split = 0.2):

l1 = int(len(data) \* (1- split))

l2 = len(data) - l1

data1 = data.iloc[0:l1,:]

data2 = data.iloc[l1:len(data),:]

return data1, data2

def min\_max\_transform(data, feature\_range=(0, 1)):

scaler = MinMaxScaler(feature\_range)

return scaler.fit\_transform(data)

def min\_max\_inverse\_transform(data\_scaled, min\_original, max\_original):

return min\_original + data\_scaled\*(max\_original - min\_original)

**VISUALIZATION**

**#Scatter Plot**

def true\_pred\_plot(model\_output):

y\_train = model\_output['datasets']['y\_train']

y\_test = model\_output['datasets']['y\_test']

train\_pred = model\_output['best\_model']['train\_predictions']

test\_pred = model\_output['best\_model']['test\_predictions']

fig = plt.figure(figsize= (14,5))

plt.subplot(121)

plt.scatter(y\_train, train\_pred, marker= "+", color = 'mediumblue')

identity\_line = np.linspace(max(min(y\_train), min(train\_pred)), min(max(y\_train), max(train\_pred)))

plt.plot(identity\_line, identity\_line, color="red", linestyle="dashed", linewidth= 2.5)

plt.xlabel("True")

plt.ylabel("Predicted")

plt.title("Training data (a)")

plt.subplot(122)

plt.scatter(y\_test, test\_pred, marker = "+", color = 'mediumblue')

identity\_line = np.linspace(max(min(y\_test), min(test\_pred)), min(max(y\_test), max(test\_pred)))

plt.plot(identity\_line, identity\_line, color="red", linestyle="dashed", linewidth= 2.5)

plt.xlabel("True")

plt.ylabel("Predicted")

plt.title("Test data (b)")

plt.suptitle("Scatter plot of True vs Predicted Close prices")

plt.show()

**#Line plot**

def best\_model\_prediction\_plot(model\_output):

time\_step = model\_output['hyper\_parameters']['time\_step']

data = model\_output['datasets']['data']

train\_predict\_plot\_data = np.empty\_like(data.values[:,0])# extracting closing price

train\_predict\_plot\_data[:] = np.nan

test\_predict\_plot\_data = np.empty\_like(data.values[:,0])

test\_predict\_plot\_data[:] = np.nan

fig = plt.figure(figsize = (14,5))

plt.subplot(121)

train\_pred = model\_output['best\_model']['train\_predictions']

test\_pred = model\_output['best\_model']['test\_predictions']

train\_predict\_plot\_data[time\_step:len(train\_pred)+ time\_step] = train\_pred

test\_predict\_plot\_data[len(train\_pred)+(time\_step\*2)+1:len(data.values)-1] = test\_pred

plt.plot(data.values[:,0],'k', linewidth = 1.5)

plt.plot(train\_predict\_plot\_data,'mediumblue',linewidth = 1.5)

plt.plot(test\_predict\_plot\_data,'darkgreen',linewidth = 1.5)

plt.xlabel('')

plt.ylabel('Close price')

plt.title("(a)")

plt.legend(['True value', 'Predicted value in train set', 'Predicted value in test set'], loc = 'upper left')

plt.subplot(122)

plt.plot(data.values[len(train\_pred)+(time\_step\*2)+1:-1, 0],'k',linewidth = 1.5)

plt.plot(test\_pred,'darkgreen',linewidth = 1.5)

plt.xlabel('')

plt.ylabel('Close price')

plt.title("(b)")

plt.legend(['True value', 'Predicted value'], loc='upper left')

def create\_visualization(model\_output):

true\_pred\_plot(model\_output)

best\_model\_prediction\_plot(model\_output)

**ARO (Artifical Rabbit Optimization)**

train\_data ,test\_data = data\_split(data, 0.2)

print("Total train data : ")

print(len(train\_data))

print("No of data used for hyper-parametr tuning")

train\_data ,val\_data = data\_split(train\_data, 0.2)

print("training data : ")

print(len(train\_data))

print("validation data : ")

print(len(val\_data))

# Scaling data

min\_train, max\_train = train\_data["Close"].min(), train\_data["Close"].max()

min\_test, max\_test = val\_data["Close"].min(), val\_data["Close"].max()

train\_data\_scaled = min\_max\_transform(train\_data)

test\_data\_scaled = min\_max\_transform(val\_data)

X\_train, y\_train = DatasetCreation(train\_data\_scaled, 5)

X\_test, y\_test = DatasetCreation(test\_data\_scaled, 5)

num\_features = train\_data.shape[1]

y\_train\_original = min\_max\_inverse\_transform(y\_train, min\_train, max\_train)

y\_test\_original = min\_max\_inverse\_transform(y\_test, min\_test, max\_test)

**# Defining objective function for ARO**

def objective\_function(solution):

neurons = int(solution[0]) # Number of neurons in LSTM layer

optimizer = solution[1]

learning\_rate = solution[2]

batch\_size = int(solution[3])

print("============================================= Running For =============================================")

print("Values used are : (neurons, optimizer, learning rate, batch size)")

print(neurons)

print(optimizer)

print(learning\_rate)

print(batch\_size)

lstm\_model = build\_single\_layer\_LSTM(layers=[neurons], time\_step= 5, num\_features=num\_features, optimizer=optimizer, learning\_rate=learning\_rate,verbose=0)

callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience= 5)

# Train the model using X\_train and y\_train

lstm\_model.fit(X\_train, y\_train, epochs=50, batch\_size=batch\_size, callbacks=[callback],verbose=0)

ypred = min\_max\_inverse\_transform(lstm\_model.predict(X\_test).ravel(), min\_test, max\_test)

ytest = min\_max\_inverse\_transform(y\_test, min\_test, max\_test)

mse = np.mean((ytest - ypred)\*\*2)

rmse = np.sqrt(mse)

print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")

print("=============================================================================================")

return rmse

**# Defining bounds for neurons, optimizer, learning rates and dropout rate**

neurons\_bounds = (10,200) # Assuming a range for neurons (you can adjust this)

optimizers = ('Adam','Adagrad','Nadam')

learning\_rates = (0.001,0.1)

batches = (8,32)

bound = (FloatVar(lb=neurons\_bounds[0], ub=neurons\_bounds[1], name="neurons"),

StringVar(valid\_sets=optimizers, name="optimizer"),

FloatVar(lb=learning\_rates[0], ub=learning\_rates[1], name="learning\_rate"),

FloatVar(lb=batches[0], ub=batches[1], name="batch\_size"))

problem\_dict = {

"bounds": bound,

"obj\_func": objective\_function,

"minmax": "min",

}

model = ARO.IARO(epoch=2 pop\_size=50)

best\_hyperparameters = model.solve(problem\_dict)

print("Best Hyperparameters (Neurons, Optimizer, Learning Rate, Batch):", best\_hyperparameters.solution)

print(best\_hyperparameters.solution)

neurons = int(best\_hyperparameters.solution[0])

optimizer = int(best\_hyperparameters.solution[1])

learning\_rate = round(best\_hyperparameters.solution[2],4)

batch\_size = int(best\_hyperparameters.solution[3])

print("=======================================================")

print("Best Hyper-parameters")

print("=======================================================")

print("LSTM Layer neurons : "+str(neurons))

print("Optimizer : "+str(optimizer))

print("Learning Rate : "+str(learning\_rate))

print("Batch Size : "+str(batch\_size))

**# Defining objective function for ARO (Multi Layer LSTM)**

def objective\_function(solution):

neurons1 = int(solution[0])

neurons2 = int(solution[1]) # Number of neurons in LSTM layer

neurons3 = int(solution[2]) # Number of neurons in LSTM layer

optimizer = solution[3]

learning\_rate = solution[4]

batch\_size = int(solution[5])

print("============================================= Running For =============================================")

print("Values used are : (neurons1, neurons2, neurons3, optimizer, learning rate, batch size)")

print(neurons1)

print(neurons2)

print(neurons3)

print(optimizer)

print(learning\_rate)

print(batch\_size)

lstm\_model = build\_single\_layer\_LSTM(layers=[neurons1,neurons2,neurons3], time\_step= 5, num\_features=num\_features, optimizer=optimizer, learning\_rate=learning\_rate,verbose=0)

callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience= 5)

# Train the model using X\_train and y\_train

lstm\_model.fit(X\_train, y\_train, epochs=40, batch\_size=batch\_size, callbacks=[callback],verbose=0)

ypred = min\_max\_inverse\_transform(lstm\_model.predict(X\_test).ravel(), min\_test, max\_test)

ytest = min\_max\_inverse\_transform(y\_test, min\_test, max\_test)

mse = np.mean((ytest - ypred)\*\*2)

rmse = np.sqrt(mse)

print(f"Root Mean Squared Error (RMSE): {rmse:.4f}")

print("=============================================================================================")

return rmse

**# Defining bounds for neurons, learning rate, batch szie and dropout rate**

neurons1\_bounds = (10,200) # Assuming a range for neurons (you can adjust this)

neurons2\_bounds = (10,200) # Assuming a range for neurons (you can adjust this)

neurons3\_bounds = (10,200) # Assuming a range for neurons (you can adjust this)

optimizers = ('Adam','Adagrad','Nadam')

learning\_rates = (0.001,0.1)

batches = (8,32)

bound = (FloatVar(lb=neurons1\_bounds[0], ub=neurons1\_bounds[1], name="neurons1"),

FloatVar(lb=neurons2\_bounds[0], ub=neurons2\_bounds[1], name="neurons2"),

FloatVar(lb=neurons3\_bounds[0], ub=neurons3\_bounds[1], name="neurons3"),

StringVar(valid\_sets=optimizers, name="optimizer"),

FloatVar(lb=learning\_rates[0], ub=learning\_rates[1], name="learning\_rate"),

FloatVar(lb=batches[0], ub=batches[1], name="batch\_size"))

problem\_dict = {

"bounds": bound,

"obj\_func": objective\_function,

"minmax": "min",

}

model = ARO.IARO(epoch=1, pop\_size=5)

best\_hyperparameters = model.solve(problem\_dict)

print("Best Hyperparameters (Neurons-1, Neurons-2, Neurons-3, Optimizer, Learning Rate, Batch):", best\_hyperparameters.solution)

print(best\_hyperparameters.solution)

print("----------------------------------------------------")

print("Min RMSE : ",best\_hyperparameters.target.fitness)

print("----------------------------------------------------")

neurons1 = int(best\_hyperparameters.solution[0])

neurons2 = int(best\_hyperparameters.solution[1])

neurons3 = int(best\_hyperparameters.solution[2])

optimizer = int(best\_hyperparameters.solution[3])

learning\_rate = round(best\_hyperparameters.solution[4],4)

batch\_size = int(best\_hyperparameters.solution[5])

if optimizer == 0:

opt = 'Adam'

elif optimizer == 1:

opt = 'Adagrad'

elif optimizer == 2:

opt = 'Nadam'

elif optimizer == 3:

opt = 'RMProps'

else:

print("No optimizer found in the list(['Adam', 'Adagrad','Nadam'])!")

print("=======================================================")

print("Best Hyper-parameters")

print("=======================================================")

print("LSTM Layer-1 neurons : "+str(neurons1))

print("LSTM Layer-2 neurons : "+str(neurons2))

print("LSTM Layer-3 neurons : "+str(neurons3))

print("Optimizer : " + str(opt))

print("Learning Rate : "+str(learning\_rate))

print("Batch Size : "+str(batch\_size))

**Building Single Layer Optimized Model**

from keras.optimizers import Adam,Adagrad,Nadam

def build\_single\_layer\_LSTM(layers, time\_step, num\_features, optimizer = 0, learning\_rate = 0.001, verbose = 1):

model = Sequential()

model.add(LSTM(int(layers[0]), input\_shape = (time\_step, num\_features)))

model.add(Dense(1, activation = 'linear'))

optimizer = int(optimizer)

if optimizer == 0:

opt = Adam(learning\_rate = learning\_rate)

elif optimizer == 1:

opt = Adagrad(learning\_rate = learning\_rate)

elif optimizer == 2:

opt = Nadam(learning\_rate = learning\_rate)

else:

print("No optimizer found in the list(['Adam', 'Adagrad','Nadam'])!")

model.compile(loss='mean\_squared\_error', optimizer= opt)

if verbose == 1:

print(model.summary())

return model

**# Single Layer LSTM**

def single\_layer\_LSTM(neurons, hyper\_parameters, data, time\_step = 5, test\_split = 0.2, epochs = 20, num\_replicates = 2):

print("Progress: Performing data preparation steps.......\n")

train\_data, test\_data = data\_split(data, test\_split)

num\_features = train\_data.shape[1]

min\_train, max\_train = train\_data["Close"].min(), train\_data["Close"].max()

min\_test, max\_test = test\_data["Close"].min(), test\_data["Close"].max()

train\_data\_scaled = min\_max\_transform(train\_data)

test\_data\_scaled = min\_max\_transform(test\_data)

X\_train, y\_train = DatasetCreation(train\_data\_scaled, time\_step)

X\_test, y\_test = DatasetCreation(test\_data\_scaled, time\_step)

y\_train\_original = min\_max\_inverse\_transform(y\_train, min\_train, max\_train)

y\_test\_original = min\_max\_inverse\_transform(y\_test, min\_test, max\_test)

print("Progress: Building and training models.......\n")

neurons = np.array(neurons)

rmse\_array = np.zeros((len(neurons), num\_replicates))

mape\_array = np.zeros((len(neurons), num\_replicates))

R\_array = np.zeros((len(neurons), num\_replicates))

elapsed\_time\_array = np.zeros((len(neurons), num\_replicates))

models\_history = []

train\_predictions = []

test\_predictions = []

for i in range(len(neurons)):

print("Model hyperparameters used: \n ", hyper\_parameters[i])

model\_history\_per\_replicate = []

train\_predictions\_per\_replicate = []

test\_predictions\_per\_replicate = []

hidden\_nodes = int(neurons[i])

for k in range(num\_replicates):

print("Program is running for %d neurons and %d replicate ----->\n" %(hidden\_nodes, k))

layers = [hidden\_nodes]

model = build\_model(layers, time\_step, num\_features, optimizer = hyper\_parameters[i][0], learning\_rate = hyper\_parameters[i][1], verbose = 0)

callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience= 5)

start = time.time()

history = model.fit(X\_train, y\_train, batch\_size = hyper\_parameters[i][2], epochs= epochs, callbacks=[callback], verbose = 1)

model.save("optimized\_single\_layer\_model\_"+str(k)+".h5")

plt.figure(figsize=(8, 5))

plt.plot(history.history['loss'])

plt.title('Epoch vs. Loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.grid(True)

plt.show()

end = time.time()

elapsed\_time = end - start

model\_history\_per\_replicate.append(history)

train\_pred = min\_max\_inverse\_transform(model.predict(X\_train).ravel(), min\_train, max\_train)

test\_pred = min\_max\_inverse\_transform(model.predict(X\_test).ravel(), min\_test, max\_test)

train\_predictions\_per\_replicate.append(train\_pred)

test\_predictions\_per\_replicate.append(test\_pred)

scores = calculate\_scores(min\_max\_inverse\_transform(y\_test, min\_test, max\_test),test\_pred)

rmse\_array[i][k] = scores['rmse']

mape\_array[i][k] = scores['mape']

R\_array[i][k] = scores['R']

elapsed\_time\_array[i][k] = elapsed\_time

models\_history.append(model\_history\_per\_replicate)

train\_predictions.append(train\_predictions\_per\_replicate)

test\_predictions.append(test\_predictions\_per\_replicate)

print("Progress: Collecting outputs.......\n")

neurons\_df = pd.DataFrame(neurons)

rmse\_df = pd.DataFrame(rmse\_array)

mape\_df = pd.DataFrame(mape\_array)

R\_df = pd.DataFrame(R\_array)

elapsed\_time\_df = pd.DataFrame(elapsed\_time\_array)

train\_predictions = np.array(train\_predictions)

test\_predictions = np.array(test\_predictions)

min\_index = pd.DataFrame(rmse\_df.min(axis = 1)).idxmin()[0]

min\_col = pd.DataFrame(rmse\_df.min(axis = 0)).idxmin()[0]

num\_neurons\_with\_best\_rmse = neurons\_df.loc[min\_index,0]

best\_rmse = rmse\_df.loc[min\_index, min\_col]

mape\_with\_best\_rmse = mape\_df.loc[min\_index, min\_col]

R\_with\_best\_rmse = R\_df.loc[min\_index, min\_col]

elapsed\_time\_with\_best\_rmse = elapsed\_time\_df.loc[min\_index, min\_col]

train\_predictions\_with\_best\_rmse = train\_predictions[min\_index][min\_col]

test\_predictions\_with\_best\_rmse = test\_predictions[min\_index][min\_col]

loss\_with\_best\_rmse = models\_history[min\_index][min\_col].history['loss']

hyper\_parameters = { 'neurons': neurons,

'model\_specific\_hyper\_parameters': hyper\_parameters,#additional best\_hyper\_parmeters for each models

'epochs': epochs,

'time\_step':time\_step,

'num\_replicates': num\_replicates,

'test\_split':test\_split

}

scores = {'neurons': neurons\_df, 'rmse': rmse\_df, 'mape': mape\_df, 'R': R\_df, 'elapsed\_time': elapsed\_time\_df}

avg\_scores = pd.DataFrame({'neurons': neurons,

'rmse': rmse\_df.mean(axis = 1),

'mape': mape\_df.mean(axis = 1),

'R': R\_df.mean(axis = 1),

'elapsed\_time': elapsed\_time\_df.mean(axis = 1)})

all\_stds = pd.DataFrame({'neurons': neurons,

'rmse': rmse\_df.std(axis = 1),

'mape': mape\_df.std(axis = 1),

'R': R\_df.std(axis = 1),

'elapsed\_time': elapsed\_time\_df.std(axis = 1)})

all\_minimums = pd.DataFrame({'neurons': neurons,

'rmse': rmse\_df.min(axis = 1),

'mape': mape\_df.min(axis = 1),

'R': R\_df.min(axis = 1),

'elapsed\_time': elapsed\_time\_df.min(axis = 1)})

all\_maximums = pd.DataFrame({'neurons': neurons,

'rmse': rmse\_df.max(axis = 1),

'mape': mape\_df.max(axis = 1),

'R': R\_df.max(axis = 1),

'elapsed\_time': elapsed\_time\_df.max(axis = 1)})

model\_with\_best\_rmse = { 'neurons': num\_neurons\_with\_best\_rmse,

'replicate': min\_col,

'rmse': best\_rmse,

'mape': mape\_with\_best\_rmse,

'R': R\_with\_best\_rmse,

'elapsed\_time': elapsed\_time\_with\_best\_rmse,

'train\_predictions':train\_predictions\_with\_best\_rmse,

'test\_predictions': test\_predictions\_with\_best\_rmse,

'loss':loss\_with\_best\_rmse,

}

datasets = {'data': data,

'X\_train': X\_train,

'X\_test': X\_test,

'y\_train': y\_train\_original,

'y\_test': y\_test\_original

}

print("\nProgress: All works are done successfully, congratulations!!\n")

print("\nBest model (neurons, replicate, rmse): ", num\_neurons\_with\_best\_rmse, min\_col, best\_rmse)

print('\nAverage scores:\n', avg\_scores)

print('\nStandard\_deviations:\n', all\_stds)

print('\nMinimums:\n', all\_minimums)

print('\nMaximums:\n', all\_maximums)

output\_dictionary = { 'hyper\_parameters': hyper\_parameters,

'best\_model': model\_with\_best\_rmse,

'scores': scores,

'avg\_scores': avg\_scores,

'all\_stds': all\_stds,

'all\_minimums': all\_minimums,

'all\_maximums': all\_maximums,

'train\_predictions': train\_predictions,

'test\_predictions': test\_predictions,

'models\_history': models\_history,

'datasets': datasets

}

return (output\_dictionary)

**#Calling Single Layer Optimized Model**

neurons = 86

optimizer = "Nadam"

learning\_rate = 0.001

batch\_size = 8

neurons = np.array([neurons])

best\_hyper\_parameters = [[optimizer, learning\_rate, batch\_size]]

sl\_model\_output = single\_layer\_LSTM(neurons, best\_hyper\_parameters, data, time\_step = 5, test\_split = 0.2,

epochs = 100, num\_replicates = 15)

create\_visualization(sl\_model\_output)

**# Multi Layer LSTM**

def multi\_layer\_LSTM\_Model(layers, hyper\_parameters, data, time\_step = 5, test\_split = 0.2, epochs = 5, num\_replicates = 2):

print("Progress: Performing data preparation steps.......\n")

train\_data, test\_data = data\_split(data, test\_split)

num\_features = train\_data.shape[1]

min\_train, max\_train = train\_data["Close"].min(), train\_data["Close"].max()

min\_test, max\_test = test\_data["Close"].min(), test\_data["Close"].max()

train\_data\_scaled = min\_max\_transform(train\_data)

test\_data\_scaled = min\_max\_transform(test\_data)

X\_train, y\_train = DatasetCreation(train\_data\_scaled, time\_step)

X\_test, y\_test = DatasetCreation(test\_data\_scaled, time\_step)

y\_train\_original = min\_max\_inverse\_transform(y\_train, min\_train, max\_train) #in original scale

y\_test\_original = min\_max\_inverse\_transform(y\_test, min\_test, max\_test) #in original scale

rmse\_array = np.zeros(num\_replicates)

mape\_array = np.zeros(num\_replicates)

R\_array = np.zeros(num\_replicates)

elapsed\_time\_array = np.zeros(num\_replicates)

models\_history = []

train\_predictions = []

test\_predictions = []

for i in range(num\_replicates):

print("Program is running for %d replicate ----->\n" %i)

model = build\_model(layers, time\_step, num\_features, optimizer = hyper\_parameters[0], learning\_rate = hyper\_parameters[1], verbose = 0)

callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience= 5)

start = time.time()

history = model.fit(X\_train, y\_train, batch\_size = hyper\_parameters[2], epochs= epochs, callbacks=[callback], verbose = 1)

end = time.time()

elapsed\_time = end - start

models\_history.append(history)

train\_pred = min\_max\_inverse\_transform(model.predict(X\_train).ravel(), min\_train, max\_train) #in original scale

test\_pred = min\_max\_inverse\_transform(model.predict(X\_test).ravel(), min\_test, max\_test)

train\_predictions.append(train\_pred)

test\_predictions.append(test\_pred)

scores = calculate\_scores(min\_max\_inverse\_transform(y\_test, min\_test, max\_test),test\_pred)

rmse\_array[i] = scores['rmse']

mape\_array[i] = scores['mape']

R\_array[i] = scores['R']

elapsed\_time\_array[i] = elapsed\_time

min\_index = rmse\_array.argmin()

best\_rmse = rmse\_array[min\_index]

mape\_with\_best\_rmse = mape\_array[min\_index]

R\_with\_best\_rmse = R\_array[min\_index]

elapsed\_time\_with\_best\_rmse = elapsed\_time\_array[min\_index]

train\_predictions\_with\_best\_rmse = train\_predictions[min\_index]

test\_predictions\_with\_best\_rmse = test\_predictions[min\_index]

loss\_with\_best\_rmse = models\_history[min\_index].history['loss']

all\_scores = {'rmse': rmse\_array, 'mape': mape\_array, 'R': R\_array, 'elapsed\_time': elapsed\_time\_array}

avg\_scores = {'rmse': np.mean(rmse\_array),

'mape': np.mean(mape\_array),

'R': np.mean(R\_array),

'elapsed\_time': np.mean(elapsed\_time\_array)}

stds = {'rmse': np.std(rmse\_array),

'mape': np.std(mape\_array),

'R': np.std(R\_array),

'elapsed\_time': np.std(elapsed\_time\_array)}

minimums = {'rmse': np.min(rmse\_array),

'mape': np.min(mape\_array),

'R': np.min(R\_array),

'elapsed\_time': np.min(elapsed\_time\_array)}

maximums = {'rmse': np.max(rmse\_array),

'mape': np.max(mape\_array),

'R': np.max(R\_array),

'elapsed\_time': np.max(elapsed\_time\_array)}

model\_with\_best\_rmse = {

'replicate': min\_index,

'rmse': best\_rmse,

'mape': mape\_with\_best\_rmse,

'R': R\_with\_best\_rmse,

'elapsed\_time': elapsed\_time\_with\_best\_rmse,

'train\_predictions':train\_predictions\_with\_best\_rmse,

'test\_predictions': test\_predictions\_with\_best\_rmse,

'loss':loss\_with\_best\_rmse,

}

hyper\_parameters = {'layers': layers,

'model\_specific\_hyper\_parameters': hyper\_parameters,#additional best\_hyper\_parmeters for each models

'epochs': epochs,

'time\_step':time\_step,

'num\_replicates': num\_replicates,

'test\_split':test\_split

}

datasets = {'data': data,

'X\_train': X\_train,

'X\_test': X\_test,

'y\_train': y\_train\_original,

'y\_test': y\_test\_original

}

output\_dictionary = {'hyper\_parameters': hyper\_parameters,

'best\_model': model\_with\_best\_rmse,

'all\_scores': all\_scores,

'avg\_scores': avg\_scores,

'standard deviations': stds,

'minimums': minimums,

'maximums': maximums,

'train\_predictions': train\_predictions,

'test\_predictions': test\_predictions,

'datasets': datasets

}

return output\_dictionary

def run\_multi\_layer\_LSTM\_Model(hidden\_layers, hyper\_parameters, data, time\_step = 5, test\_split = 0.2, epochs = 5, num\_replicates = 2):

num\_models = len(hidden\_layers)

rmse = []

mape = []

R = []

elapsed\_time = []

avg\_rmse = []

avg\_mape = []

avg\_R = []

avg\_elapsed\_time = []

best\_avg\_rmse = 99999999999

best\_rmse = 99999999999

best\_model\_hidden\_layers = None

best\_model\_output = None

for i in range(num\_models):

print("Running model with hidden neurons: ", hidden\_layers[i])

print("\n")

print("Best Hyper\_parameters used: ", hyper\_parameters[i])

print("\n")

output = multi\_layer\_LSTM\_Model(hidden\_layers[i], hyper\_parameters[i], data, time\_step, test\_split, epochs, num\_replicates)

rmse.append(output['all\_scores']['rmse'])

mape.append(output['all\_scores']['mape'])

R.append(output['all\_scores']['R'])

elapsed\_time.append(output['all\_scores']['elapsed\_time'])

avg\_rmse.append(output['avg\_scores']['rmse'])

avg\_mape.append(output['avg\_scores']['mape'])

avg\_R.append(output['avg\_scores']['R'])

avg\_elapsed\_time.append(output['avg\_scores']['elapsed\_time'])

if avg\_rmse[i] < best\_avg\_rmse:

best\_avg\_rmse = avg\_rmse[i]

best\_rmse = output['best\_model']['rmse']

best\_model\_hidden\_layers = hidden\_layers[i]

best\_model\_output = output

rmse = np.array(rmse)

mape = np.array(mape)

R = np.array(R)

scores = {'layers': hidden\_layers, 'rmse': rmse, 'mape': mape, 'R':R, 'elapsed\_time': elapsed\_time}

avg\_scores = pd.DataFrame({'layers': hidden\_layers, 'rmse': np.array(avg\_rmse), 'mape': np.array(avg\_mape), 'R':np.array(avg\_R), 'elapsed\_time':np.array(avg\_elapsed\_time)})

stds = pd.DataFrame({'layers': hidden\_layers, 'rmse': np.std(rmse, axis = 1), 'mape': np.std(mape, axis = 1), 'R': np.std(R, axis = 1 ), 'elapsed\_time': np.std(elapsed\_time, axis = 1 )})

minimums = pd.DataFrame({'layers': hidden\_layers, 'rmse': np.min(rmse, axis =1 ), 'mape': np.min(mape, axis= 1), 'R': np.min(R, axis =1), 'elapsed\_time': np.min(elapsed\_time, axis =1)})

maximums = pd.DataFrame({'layers': hidden\_layers, 'rmse': np.max(rmse, axis =1), 'mape': np.max(mape, axis =1), 'R': np.max(R, axis =1), 'elapsed\_time': np.max(elapsed\_time,axis =1)})

output\_dictionary = {

'hyper\_parameters': hyper\_parameters[i],

'scores': scores,

'avg\_scores': avg\_scores,

'stds':stds,

'minimums': minimums,

'maximums': maximums,

'best\_avg\_rmse': best\_avg\_rmse,

'best\_rmse': best\_rmse,

'best\_model\_hidden\_layers': best\_model\_hidden\_layers,

'best\_model\_output': best\_model\_output

}

print("\nBest model and its avg rmse and minimum rmse):\n", best\_model\_hidden\_layers, best\_avg\_rmse, best\_rmse)

print('\nAverage scores:\n', avg\_scores)

print('\nStandard\_deviations:\n', stds)

print('\nMinimums:\n', minimums)

print('\nMaximums:\n', maximums)

return output\_dictionary

**#Calling Multi Layer Optimized Model**

hidden\_layers = [[40, 10], [10, 10, 45]]

best\_hyper\_parameters\_multilayers = [['Adam', 0.0201, 8],

['Adam', 0.0115, 9]]

ml\_model\_output = run\_multi\_layer\_LSTM\_Model(hidden\_layers, best\_hyper\_parameters\_multilayers, data, time\_step = 5, test\_split = 0.2, epochs = 100, num\_replicates = 15)