DES-DA-ED-Bi-GRU-BO

import numpy as np

import pandas as pd

from sklearn.preprocessing import MinMaxScaler

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

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Bidirectional, GRU, Dense, RepeatVector, TimeDistributed

from bayes\_opt import BayesianOptimization

import matplotlib.pyplot as plt

import yfinance as yf

# Define a function to calculate Theil's U-statistic

def calculate\_theils\_u(y\_true, y\_pred):

diff\_squared = np.square(y\_true - y\_pred)

y\_true\_squared = np.square(y\_true)

u\_statistic = np.sqrt(np.sum(diff\_squared) / np.sum(y\_true\_squared))

return u\_statistic

# Function for Double Exponential Smoothing

def double\_exponential\_smoothing(series, alpha, beta):

result = [series[0]]

for t in range(1, len(series)):

if t == 1:

level, trend = series[0], series[1] - series[0]

else:

value = series[t]

last\_level, level = level, alpha \* value + (1 - alpha) \* (level + trend)

trend = beta \* (level - last\_level) + (1 - beta) \* trend

result.append(level + trend)

return result

# Download and preprocess GE stock data

df = yf.download('GE', start='2010-01-01', end='2023-01-01')

df = df[['Close']]

df['DES'] = double\_exponential\_smoothing(df['Close'], alpha=0.3, beta=0.1)

df1 = df[['Close', 'DES']]

# Normalize the data

scaler = MinMaxScaler(feature\_range=(0, 1))

scaled\_data = scaler.fit\_transform(df1)

# Function to create input sequences and corresponding outputs for encoder-decoder

def create\_sequences(dataset, look\_back\_enc, look\_back\_dec):

dataX, dataY = [], []

for i in range(len(dataset) - look\_back\_enc - look\_back\_dec + 1):

input\_seq = dataset[i:(i + look\_back\_enc), :]

output\_seq = dataset[(i + look\_back\_enc):(i + look\_back\_dec + look\_back\_enc), :]

dataX.append(input\_seq)

dataY.append(output\_seq)

return np.array(dataX), np.array(dataY)

# Set the look-back windows for encoder and decoder

look\_back\_encoder = 3

look\_back\_decoder = 2

# Create input sequences and corresponding outputs for encoder-decoder

X, Y = create\_sequences(scaled\_data, look\_back\_encoder, look\_back\_decoder)

# Reshape input and output data to be [samples, time steps, features]

X = np.reshape(X, (X.shape[0], X.shape[1], df1.shape[1]))

Y = np.reshape(Y, (Y.shape[0], Y.shape[1], df1.shape[1]))

# Define the objective function for Bayesian optimization

def gru\_bayesian\_optimizer(units, epochs, batch\_size):

units = int(units)

epochs = int(epochs)

batch\_size = int(batch\_size)

# Define the encoder-decoder model with DA-Bi-GRU layers

model = Sequential()

model.add(Bidirectional(GRU(units, activation='relu'), input\_shape=(look\_back\_encoder, df1.shape[1])))

model.add(RepeatVector(look\_back\_decoder))

model.add(Bidirectional(GRU(units, activation='relu', return\_sequences=True)))

model.add(TimeDistributed(Dense(df1.shape[1])))

model.compile(loss='mse', optimizer='adam')

# Train the model

model.fit(X, Y, epochs=epochs, batch\_size=batch\_size, verbose=0)

# Predict on the training data

train\_predict = model.predict(X)

# Invert predictions and actuals to the original scale

train\_predict\_inverted = scaler.inverse\_transform(train\_predict.reshape(-1, df1.shape[1]))

Y\_original\_inverted = scaler.inverse\_transform(Y.reshape(-1, df1.shape[1]))

# Calculate root mean squared error (RMSE)

train\_score = np.sqrt(mean\_squared\_error(Y\_original\_inverted, train\_predict\_inverted))

return -train\_score # Return negative RMSE as we want to maximize the optimizer score

# Define the search space for hyperparameters

pbounds = {'units': (10, 100), 'epochs': (50, 100), 'batch\_size': (1, 10)}

# Initialize Bayesian optimization

optimizer = BayesianOptimization(f=gru\_bayesian\_optimizer, pbounds=pbounds, random\_state=42)

# Perform optimization

optimizer.maximize(init\_points=5, n\_iter=10)

# Retrieve the best hyperparameters

best\_params = optimizer.max['params']

best\_units = int(best\_params['units'])

best\_epochs = int(best\_params['epochs'])

best\_batch\_size = int(best\_params['batch\_size'])

# Display the best hyperparameters

print("Best Hyperparameters:")

print(f"Units: {best\_units}")

print(f"Epochs: {best\_epochs}")

print(f"Batch Size: {best\_batch\_size}")

# Create and compile the final model using the best hyperparameters

final\_model = Sequential()

final\_model.add(Bidirectional(GRU(best\_units, activation='relu'), input\_shape=(look\_back\_encoder, df1.shape[1])))

final\_model.add(RepeatVector(look\_back\_decoder))

final\_model.add(Bidirectional(GRU(best\_units, activation='relu', return\_sequences=True)))

final\_model.add(TimeDistributed(Dense(df1.shape[1])))

final\_model.compile(loss='mse', optimizer='adam')

# Train the final model

final\_model.fit(X, Y, epochs=best\_epochs, batch\_size=best\_batch\_size, verbose=2)

# Predict using the final model

train\_predict = final\_model.predict(X)

# Invert predictions and actuals to the original scale

train\_predict\_inverted = scaler.inverse\_transform(train\_predict.reshape(-1, df1.shape[1]))

Y\_original\_inverted = scaler.inverse\_transform(Y.reshape(-1, df1.shape[1]))

# Calculate evaluation metrics

mae = mean\_absolute\_error(Y\_original\_inverted, train\_predict\_inverted)

mse = mean\_squared\_error(Y\_original\_inverted, train\_predict\_inverted)

rmse = np.sqrt(mse)

theils\_u\_statistic = calculate\_theils\_u(Y\_original\_inverted, train\_predict\_inverted)

# Print the evaluation metrics

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

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

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

print(f"Theil's U-statistic: {theils\_u\_statistic:.4f}")

# Plot the results for one variable (adjust the index as needed)

plt.plot(Y\_original\_inverted[:, 0], label='Actual')

plt.plot(train\_predict\_inverted[:, 0], label='Predicted')

plt.xlabel('Time Step')

plt.ylabel('Stock Price')

plt.title('GE Stock Price Forecasting with DES-DA-ED-Bi-GRU-BO')

plt.legend()

plt.show()