COS30018 – Task 2 Report

Name: Abhinav Karn

Student ID: 104488053

Date: 23rd August 2024

Explanation of the modifications and additions in v0.1 code as per instructed in Task 2:

1. **Importing the Statements and Constants**

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import pandas_datareader as web
import datetime as dt
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, LSTM, InputLayer
import yfinance as yf
import os
```

These imports make it possible to use a number of very important libraries and functions, among them numpy and pandas for dealing with data, matplotlib for visualization of data, yfinance for fetching data, scikit-learn for scaling and splitting data, and finally, tensorflow for making a neural network. The 'os' module shall be used for operations dealing with files and directories, like checking for the existence of local data and for the creation of directories.

2. **Defining constants and configuration parameters:**

```python
# Define constants and configuration parameters
N_STEPS = 50
LOOKUP_STEP = 1
TEST_SIZE = 0.2
FEATURE_COLUMNS = ['Adj Close', 'Volume']
LOSS = 'mean_squared_error'
UNITS = 256
N_LAYERS = 2
DROPOUT = 0.2
OPTIMIZER = 'adam'
BIDIRECTIONAL = False
SCALE = True
SPLIT_BY_DATE = True
SHUFFLE = False
MODEL_NAME = 'model_v0.1'
SAVE_LOCAL = True
DATA_DIR = 'local_data'
NAN_STRATEGY = 'fill_mean'
```

These constants can be thought of as parameters defining the configuration for data preprocessing and model training.

N_STEPS and LOOKUP_STEP control how big our input sequences are, and how far ahead we are predicting.

FEATURE_COLUMNS indicate the columns from the dataset used in making the predictions.

LOSS, UNITS, N_LAYERS, DROPOUT, OPTIMIZER, BIDIRECTIONAL These variables configure neural network architecture and training parameters.

It is determined by SCALE and NAN_STRATEGY whether to scale data or how to deal with NaN values.

SAVE_LOCAL, DATA_DIR, and MODEL_NAME manage local storage and naming conventions.

3. **Loading the data and checking or the local storage**

```python
# Load data from Yahoo Finance
def load_data(ticker, start_date, end_date, feature_columns=FEATURE_COLUMNS):
    local_file_path = f"{DATA_DIR}/{ticker}_data.csv"
    if SAVE_LOCAL and os.path.exists(local_file_path):
        data = pd.read_csv(local_file_path, index_col=0, parse_dates=True)
        print(f"Loaded data from local storage: {local_file_path}")
    else:
        data = yf.download(ticker, start_date, end_date)
        if SAVE_LOCAL:
            os.makedirs(DATA_DIR, exist_ok=True)
            data.to_csv(local_file_path)
            print(f"Data saved locally to: {local_file_path}")
```

The code checks if the data for the given ticker is already in the local storage. If the data exists, then it directly reads it from a CSV file with pd.read_csv(). Otherwise, it downloads it with yfinance from Yahoo Finance. The os.makedirs() function makes sure to create the directory if it does not exist before saving the data, thus avoiding errors if the directory does not exist. This setup will give the best performance, reducing API calls as much as possible, especially during development.

4. Handling Missing Values Based on the Defined Strategy

```python
# Handle missing values
if NAN_STRATEGY == 'drop':
    data.dropna(inplace=True)
elif NAN_STRATEGY == 'fill_mean':
    data.fillna(data.mean(), inplace=True)
else:
    raise ValueError(f"Unknown NaN strategy: {NAN_STRATEGY}. Use 'drop' or 'fill_mean'.")
```

Explanation: This function handles missing values NAN. Here it applies the strategy defined in NAN_STRATEGY:

If it is set to 'drop', this removes all rows with missing values using data.dropna(inplace=True).

If 'fill_mean', fills missing values with columns' mean using data.fillna(data.mean(), inplace=True).

This flexibility will allow the model to process a dataset containing data of varying quality. The line raise ValueError ensures protection against strategy values that may be invalid.

5. Scaling Features Using MinMaxScaler

```python
# Scale features
if SCALE:
    scaler = MinMaxScaler(feature_range=(0, 1))
    data[feature_columns] = scaler.fit_transform(data[feature_columns])
else:
    scaler = None
```

It means that scaling of the features is important in a neural network, particularly when these features are of very different magnitude, such as stock prices compared to trading volumes. ‘MinMaxScaler’ will scale each feature onto a range of 0 through 1, normalizing the input data. As a result, it will make the LSTM model converge more efficiently because all features are treated with equal importance. If set to 'False', no scaling takes place.

6. **Splitting Data into Training and Testing Sets**

```
    # Split data
    if SPLIT_BY_DATE:
        train_data = data[:int(len(data) * (1 - TEST_SIZE))]
        test_data = data[int(len(data) * (1 - TEST_SIZE)):]
    else:
        train_data, test_data = train_test_split(data, test_size=TEST_SIZE, shuffle=SHUFFLE, random_state=42)

```

The dataset is split according to the SPLIT_BY_DATE flag into a training and a test set:

In the event that SPLIT_BY_DATE is True, this will create a chronological split with a percentage defined by TEST_SIZE reserved for testing. This is typical for most time-series data where random shuffling might disrupt temporal dependencies.

Unless False, it uses train_test_split() from scikit-learn to randomly split the data. It has shuffling controlled by the SHUFFLE flag and a fixed random_state for reproducibility.

7. **Creating Sequences for LSTM Input**

```
# Create X and y arrays from the data
def create_xy(data, n_steps, lookup_step, feature_columns):
    """
    Create X (features) and y (labels) arrays from the data.

    Parameters:
    data (pd.DataFrame): Data containing features
    n_steps (int): Number of time steps to use as input
    lookup_step (int): Number of days to look ahead for the label
    feature_columns (list): List of feature columns to use

    Returns:
    X (numpy array): Array of feature data
    y (numpy array): Array of labels
    """
    X, y = [], []
    data = data[feature_columns].values
    for i in range(len(data) - n_steps - lookup_step):
        X.append(data[i:i+n_steps])
        y.append(data[i+n_steps+lookup_step-1][0])  # Use the first column for the label (e.g., 'adjclose')
    return np.array(X), np.array(y)
```

Overview: The following function is used to prepare the sequences of input (X) and output (y) while training the LSTM model:

The function processes the input by creating sub-arrays of length n_steps, which are past observations, in order to predict the value at lookup_step ahead.

This method of generating a sequence is how an LSTM learns to predict future values from the patterns in historical data.

The 0 index ensures this will only make use of the single column of features—like 'Adj Close'—for the prediction target.

8. Constructing the LSTM Model

```python
# Construct the LSTM model
def create_model(n_steps, n_features, loss, units, n_layers, dropout, optimizer, bidirectional):
    """
    Create an LSTM model for time series prediction.

    Parameters:
    n_steps (int): Number of time steps in the input
    n_features (int): Number of features in the input
    loss (str): Loss function to use
    units (int): Number of LSTM units in each layer
    n_layers (int): Number of LSTM layers
    dropout (float): Dropout rate to prevent overfitting
    optimizer (str): Optimizer for training
    bidirectional (bool): Whether to use bidirectional LSTM layers

    Returns:
    model (Sequential): Compiled LSTM model
    """
    model = Sequential()
    for i in range(n_layers):
        if i == 0:
            if bidirectional:
                model.add(Bidirectional(LSTM(units, return_sequences=True), input_shape=(n_steps, n_features)))
            else:
                model.add(LSTM(units, return_sequences=True, input_shape=(n_steps, n_features)))
        elif i == n_layers - 1:
            if bidirectional:
                model.add(Bidirectional(LSTM(units)))
            else:
                model.add(LSTM(units))
        else:
            if bidirectional:
                model.add(Bidirectional(LSTM(units, return_sequences=True)))
            else:
                model.add(LSTM(units, return_sequences=True))
        model.add(Dropout(dropout))
    model.add(Dense(1))  # Output layer for predicting a single value
    model.compile(loss=loss, optimizer=optimizer)
    return model
```

Overview: This function will construct the LSTM model.

Sequential() This is a linear stack of layers.

The loop adds LSTM layers. First and last layers are handled separately: input shape for the first, no return_sequences for the last to return a scalar prediction.

Bidirectional wrapping is conditional on the bidirectional flag and hence enables the model to learn patterns temporal in both directions.

The dropout layers prevent overfitting.

The model is compiled with the definition of a loss function and an optimizer to set what the goal for training shall be.

9. Inverse Scaling Predictions to Original Values

```
# Inverse transform to get actual price values (if scaling was applied)
if scaler:
    predicted_prices = scaler.inverse_transform(np.concatenate((predicted_prices, np.zeros((predicted_prices.shape[0], len(FEATURE_COLUMNS) - 1))), axis=1))[:, 0]
    true_prices = scaler.inverse_transform(np.concatenate((y_test.reshape(-1, 1), np.zeros((y_test.shape[0], len(FEATURE_COLUMNS) - 1))), axis=1))[:, 0]
else:
    true_prices = y_test
```

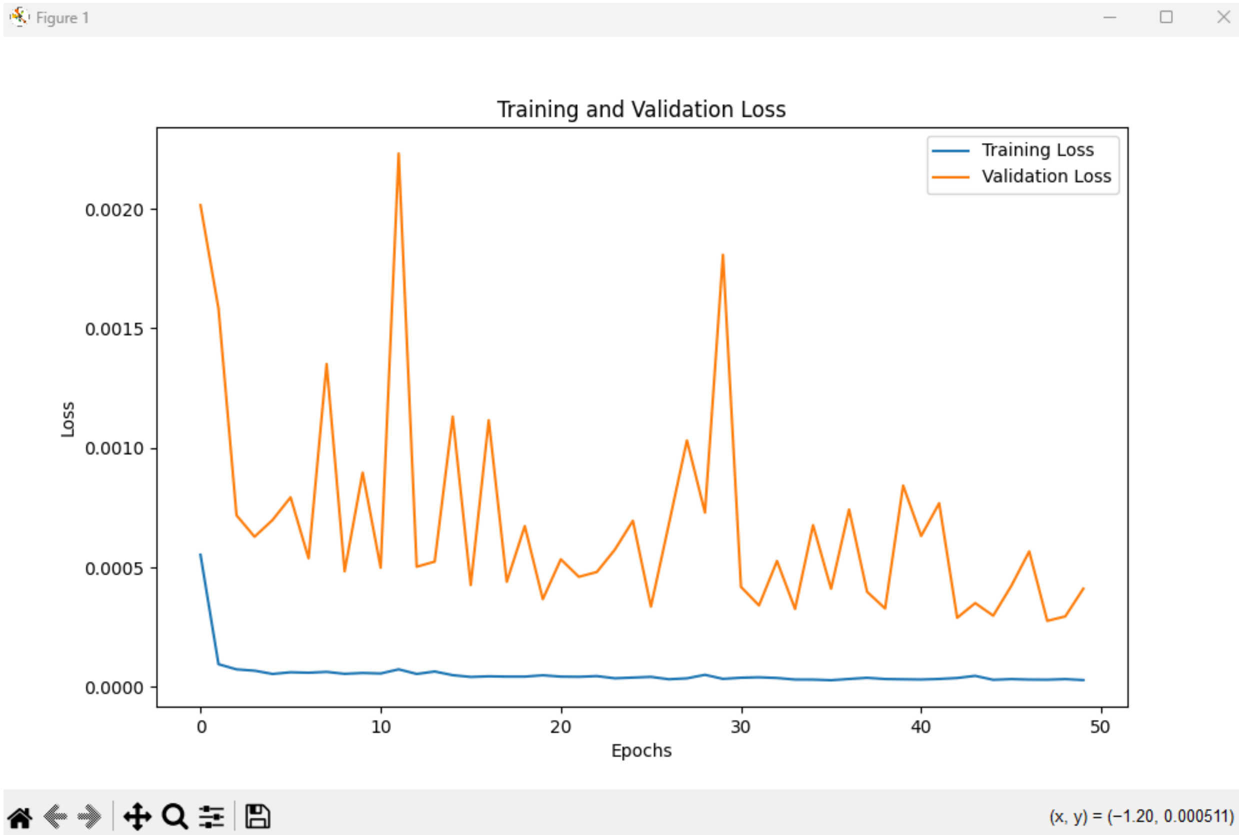
This block rescales the scaled predictions and true values back to the original scale. np.concatenate concatenates the prediction with zero-filled arrays to match the input shape expectation for a scaler.
scaler.inverse_transform This will inverse detach the scaled values back to the original stock prices so that the results are interpretable.
This is a crucial step because it ensures that the performance is evaluated and visualized accurately regarding real stock prices.

10. Plotting Actual vs. Predicted Prices

```
# Plot the true prices vs. predicted prices
plt.figure(figsize=(10, 6))
plt.plot(true_prices, color='black', label='Actual Prices')
plt.plot(predicted_prices, color='green', label='Predicted Prices')
plt.title('Actual vs. Predicted Prices')
plt.xlabel('Time')
plt.ylabel('Price')
plt.legend()
plt.show()
```

Explanation: This visualization plots actual stock prices against predicted prices:

- Helps visually assess the model's prediction performance.
- plt.plot() draws the lines for actual and predicted prices.
- The plot provides a clear picture of how well the model tracks real-world trends.



Summary:
In this detailed explanation, it shall cover all the important modifications and additions into my code.