Intelligent System

Assignment 1 - Option B

Week 4 Report

Xuan Tuan Minh Nguyen - 103819212

Environment Setup

1. Create a new environment using Conda CLI

There are several different procedures to create an environment, but the given procedure below will **encapsulate** all of the **required steps** to create **a safe and clean environment** using Conda.

- Navigate to the Github repository that contains the source code for v0.3.
- Once navigated, download the source code by clicking on **Code → Download ZIP** or use the following command in the **CLI (Terminal):**

git clone https://github.com/cobeo2004/cos30018.git

- Once the source code is successfully cloned (downloaded), navigate to the Week 3/v0.2 folder and execute the file conda-config.sh using the following command:

bash conda-config.sh

- The given file config.sh will execute the following procedure:
    - Generate an environment with a pre-defined name (you can change the name if you want to) in Python 3.10.9 by using the command: **conda create -n cos30018_env_w3_v0.2 python=3.10.9**
    - Activate the created environment using: **conda activate cos30018_env_w3_v0.2.**
    - Check and validate if the conda environment is successfully initialized by running **conda info –envs** for listing conda environments and see which environment that we are in and current Python version using **python –version.**

2. Installing required dependencies

Once the **environment** is **successfully initialized**, we can start **installing** the **dependencies (libraries)** that are **required by the program**. There are multiple pathways to install dependencies in Python, but the **most popular steps** are:

- Scan through the code to find out the required dependencies; for example, consider the file stock_prediction.py. We could see that there are quite a few required dependencies, such as: numpy matplotlib pandas tensorflow scikit-learn pandas-datareader yfinance TA-lib. However, there will be a new library called mplfinance that helps us to efficiently create a beautiful and easy to analyze **candlestick chart** without having to manually set up.

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import yfinance as yf
import talib as ta
import mplfinance as mpf
import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn import metrics

import tensorflow as tf
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, LSTM, Dropout, InputLayer, Input, Activation, Bidirectional, GRU, SimpleRNN
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.utils import plot_model

import pickle
import os
from typing_extensions import Annotated, Doc, TypeVar, Literal, Tuple, Literal, Required, TypedDict, Optional, List, NotRequired
```

Figure 1: Required Dependencies in stock_prediction.ipynb

- Once dependencies are scanned, use the following command to install the dependencies: pip install numpy matplotlib pandas tensorflow scikit-learn pandas-datareader yfinance TA-lib mplfinance.
- Another step is to list all required libraries into a requirements.txt file, and using the following command to install the required dependencies: pip install -U -r requirements.txt.

```
numpy
matplotlib
pandas
tensorflow
scikit-learn
pandas-datareader
yfinance
TA-lib
mplfinance

```

Figure 2: Example of requirements.txt

Understanding the Machine Learning 1

1. Define configuration dictionary

Codebase

```python
class ModelConfig(TypedDict):
    type: Required[Literal["LSTM", "GRU", "RNN"]]
    isBidirectional: Required[bool]
    units: Required[int]
    return_sequences: Required[bool]
    dropout: Required[float]
    activation: NotRequired[Literal["tanh", "relu", "sigmoid", "softmax", "linear"]]
```

Figure 3: ModelConfig typing

Attributes

- type: Required[Literal[“LSTM”, “GRU”, “RNN”]]: The layer type, which allows three values of LSTM, GRU and RNN and must be chosen.
- isBidirectional: Required[bool]: Determine if the layer is bidirectional or not, and the value is required.
- units: Required[int]: The number of units for each layer, which is also a required parameter for the dictionaries.
- return_sequence: Required[bool]: The return sequence of each layer, which is also a required parameter for the dictionaries.
- dropout: Required[float]: The dropout rate of each layer, which is required for each layer.
- activation: NotRequired[Literal[“tanh”, “relu”, “sigmoid”, “softmax”, “linear”]]: The activation layer, which allows 5 values of tanh, relu, sigmoid, softmax and linear and is not required, which means we could let it blank.

2. Function to generate dynamic model

Codebase

```python
def make_dynamic_model(input_shape: tuple[int, int], config: List[ModelConfig], output_units: Optional[int] = 1):
    model = Sequential()

    first_layer = config[0]
    first_layer_type = first_layer['type']
    if first_layer['isBidirectional']:
        match first_layer_type:
            case "LSTM":
                model.add(Bidirectional(LSTM(units=first_layer['units'], return_sequences=first_layer['return_sequences']), input_shape=input_shape))
            case "GRU":
                model.add(Bidirectional(GRU(units=first_layer['units'], return_sequences=first_layer['return_sequences']), input_shape=input_shape))
            case "RNN":
                model.add(Bidirectional(SimpleRNN(units=first_layer['units'], return_sequences=first_layer['return_sequences']), input_shape=input_shape))
    else:
        match first_layer_type:
            case "LSTM":
                model.add(LSTM(units=first_layer['units'], return_sequences=first_layer['return_sequences'], input_shape=input_shape))
            case "GRU":
                model.add(GRU(units=first_layer['units'], return_sequences=first_layer['return_sequences'], input_shape=input_shape))
            case "RNN":
                model.add(SimpleRNN(units=first_layer['units'], return_sequences=first_layer['return_sequences'], input_shape=input_shape))

    if 'activation' in first_layer:
        model.add(Activation(first_layer['activation']))

    model.add(Dropout(first_layer['dropout']))

    for layer in config[1:]:
        multi_layer_type = layer['type']
        if layer['isBidirectional']:
            match multi_layer_type:
                case "LSTM":
                    model.add(Bidirectional(LSTM(units=layer['units'], return_sequences=layer['return_sequences']), input_shape=input_shape))
                case "GRU":
                    model.add(Bidirectional(GRU(units=layer['units'], return_sequences=layer['return_sequences']), input_shape=input_shape))
                case "RNN":
                    model.add(Bidirectional(SimpleRNN(units=layer['units'], return_sequences=layer['return_sequences']), input_shape=input_shape))
        else:
            match multi_layer_type:
                case "LSTM":
                    model.add(LSTM(units=layer['units'], return_sequences=layer['return_sequences'], input_shape=input_shape))
                case "GRU":
                    model.add(GRU(units=layer['units'], return_sequences=layer['return_sequences'], input_shape=input_shape))
                case "RNN":
                    model.add(SimpleRNN(units=layer['units'], return_sequences=layer['return_sequences'], input_shape=input_shape))

        if 'activation' in layer:
            print(layer['activation'])
            model.add(Activation(layer['activation']))

        model.add(Dropout(layer['dropout']))

    model.add(Dense(units=output_units))

    return model
```

*Figure 4: Codebase for generate **dynamic model***

Parameters

- input_shape: Tuple[int, int]: The tuple of the **input data shape** that will be used for **training the neural network**.
- config: List[ModelConfig]: A list of predefined dictionaries (Refer to section ***Define configuration dictionary***) that represents the configuration of each layer, which includes type of layer (**LSTM, GRU and RNN**), the number of **units**, is that layer **bi-directional**, is that layer must **return a sequence**, predefined **activation function** and the **rate of dropout**.
- output_units: Optional[int] = 1: Indicate the **number of units** in the **output layer** of the neural network, by default it is equal to 1.

Functionalities

- The make_dynamic_model()function will execute the following procedures:
  + Create a Sequential Model: Firstly, this function will create a Sequential Model using Sequential()class, this Sequential model will be used as the wireframe to add various layers based on the configurations defined in the config parameter.
  + Create First Layer: After the Sequential Model is created, the function will then configure the first layer based on the parameters defined in the first element inside the config parameter. It involves creating different types of layers (LSTM, GRU or SimpleRNN) and wrapping them in a Bidirectional layer if the isBidirectional

parameter is set to True. The input_shape parameter will be used as the input shape for every layer.

+ Appending Activation and Dropout to First Layer: If an activation function is specified in the parameter activation of the config parameter, then it should be added to the correlated layer as well. A dropout layer is also being added with specified dropout rate defined in config parameter to not get overfitted.
+ Rest of layers configuration: Just like the first layer, the function will start iterating from second to last elements of the layers defined in config parameter and add different types of layers, wrapping them in a Bidirectional layer if the isBidirectional parameter is set to True. In addition, Dropout and Activation layers are also appended based on the configurations.
+ Output layer: Finally, a Dense layer will be added with the specified number of units defined in output_units to get the final output.

3. Function to generate metric

Codebase

```python
def metric_plot(model_train_history: tf.keras.callbacks.History, metric_1: str, metric_2: str, plot_name: str):
    metric_1_values = model_train_history.history[metric_1]
    metric_2_values = model_train_history.history[metric_2]
    epochs = range(len(metric_1_values))
    plt.figure(figsize=(12, 6))
    plt.plot(epochs, metric_1_values, label=metric_1)
    plt.plot(epochs, metric_2_values, label=metric_2)
    plt.title(f'{plot_name} Comparison')
    plt.xlabel('Epoch')
    plt.ylabel('Value')
    plt.legend()
```

*Figure 5: Codebase for drawing **metric chart***

Parameters

- model_train_history: tensorflow.keras.callbacks.History: The history values of the trained model.
- metric_1: str: String that represents the name of the first metric to retrieve the value from the model_train_history and also used for displaying as the name of the metric in the plot.
- metric_2: str: String that represents the name of the second metric to retrieve the value from the model_train_history and also used for displaying as the name of the metric in the plot.
- plot_name: str: String that represents the title of the chart.

Functionalities

- The metric_plot() function implements the following procedures:

+ Get Metric Values: Fetch the history metric values from the model_training_history.history dictionary using the name specified in the metric_1 and metric_2.
+ Calculate Epoch Range: After obtaining the Metric Values, the function will calculate the number of epochs based on the length of the Metric Values list of the metric_1 obtained in the first step. This range will be used as the x-axis of the plot.
+ Plot the chart: It will plot the metric values with the epoch range using plot() function from matplotlib library. The metric_1 will display in the blue color while the metric_2 will display in the orange color.
+ Add title and legend for the chart: The function will set the tile of the chart based on the plot_name parameter and adding legend to the plot so that the user can identify which color represents metric_1 and metric_2.

Deploying and Testing the Codebase

1. Result

- Test case 1: LSTM, GRU, RNN with different activations

```
model_config: List[ModelConfig] = [
    {
        'type': 'LSTM',
        'isBidirectional': False,
        'units': 120,
        'return_sequences': True,
        'dropout': 0.2,
        'activation': 'tanh'
    },
    {
        'type': 'LSTM',
        'isBidirectional': False,
        'units': 100,
        'return_sequences': True,
        'dropout': 0.2,
    },
    {
        'type': 'GRU',
        'isBidirectional': False,
        'units': 80,
        'return_sequences': True,
        'dropout': 0.2,
    },
    {
        'type': 'RNN',
        'isBidirectional': False,
        'units': 60,
        'return_sequences': True,
        'dropout': 0.2,
        'activation': 'relu'
    },
    {
        'type': 'LSTM',
        'isBidirectional': False,
        'units': 40,
        'return_sequences': False,
        'dropout': 0.2,
    },
]
```

Figure 6: Configuration of Test case 1

Figure 7: Result of Test case 1

- Test case 2: LSTM Model Only

```
model_config: List[ModelConfig] = [
    {
        'type': 'LSTM',
        'isBidirectional': False,
        'units': 50,
        'return_sequences': True,
        'dropout': 0.2,
    },
    {
        'type': 'LSTM',
        'isBidirectional': False,
        'units': 50,
        'return_sequences': False,
        'dropout': 0.2,
    }
]
```

Figure 8: Configuration of Test case 2

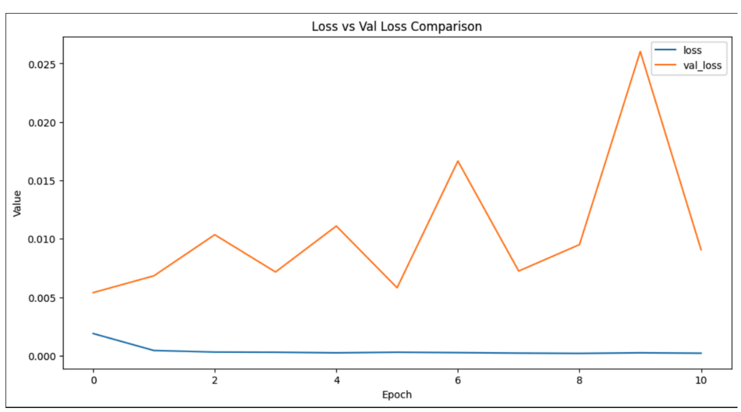

Figure 9: Result of Test case 2

- Test case 3: GRU Model Only

```
model_config: List[ModelConfig] = [
    {
        'type': 'GRU',
        'isBidirectional': False,
        'units': 100,
        'return_sequences': True,
        'dropout': 0.2,
    },
    {
        'type': 'GRU',
        'isBidirectional': False,
        'units': 100,
        'return_sequences': False,
        'dropout': 0.2,
    }
]
```

Figure 10: Configure of Test case 3

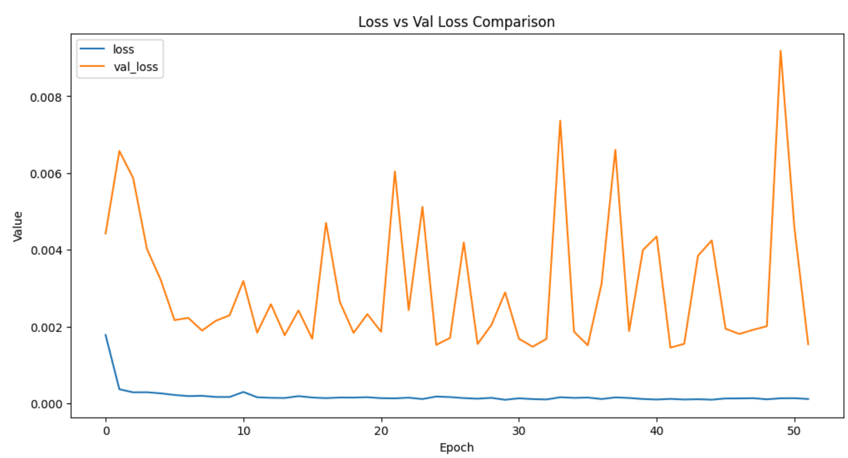

Figure 11: Result of Test case 3

- Test case 4: LSTM and GRU Model Mixed

```
model_config: List[ModelConfig] = [
    {
        'type': 'LSTM',
        'isBidirectional': False,
        'units': 100,
        'return_sequences': True,
        'dropout': 0.2,
    },
    {
        'type': 'GRU',
        'isBidirectional': False,
        'units': 100,
        'return_sequences': False,
        'dropout': 0.2,
    }
]
```

Figure 12: Configure of Test case 4

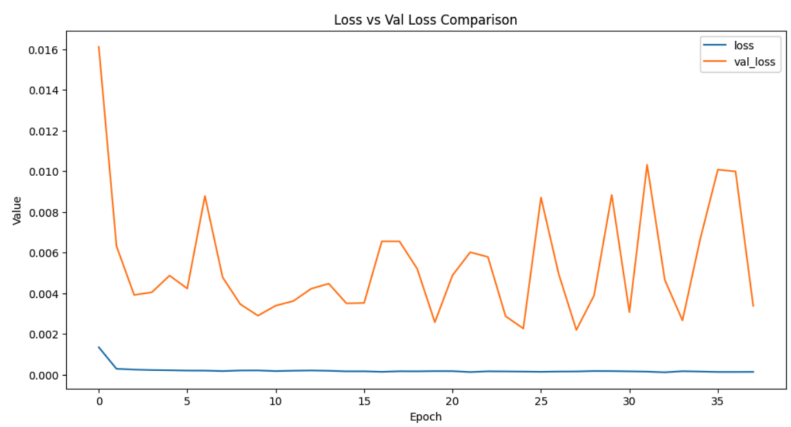

Figure 13: Result of Test case 4

- Test case 5: Bidirectional LSTM Model

```
model_config: List[ModelConfig] = [
    {
        'type': 'LSTM',
        'isBidirectional': True,
        'units': 128,
        'return_sequences': True,
        'dropout': 0.2,
    },
    {
        'type': 'LSTM',
        'isBidirectional': True,
        'units': 128,
        'return_sequences': True,
        'dropout': 0.2,
    },
    {
        'type': 'LSTM',
        'isBidirectional': True,
        'units': 128,
        'return_sequences': True,
        'dropout': 0.2,
    },
    {
        'type': 'LSTM',
        'isBidirectional': True,
        'units': 128,
        'return_sequences': False,
        'dropout': 0.2,
    },

]
```

Figure 14: Configure of Test case 5

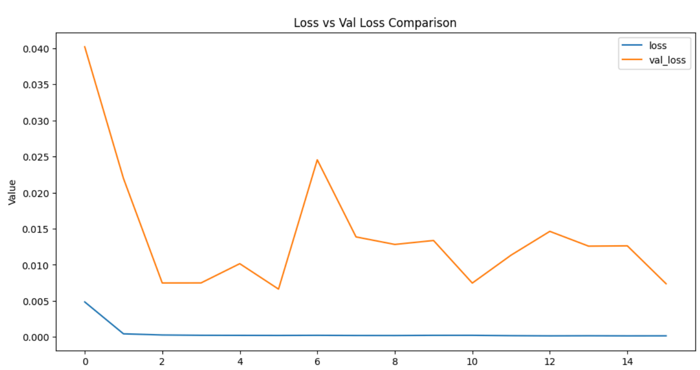

Figure 15: Result of Test case 5

- Test case 6: Bidirectional GRU Model

```
model_config: List[ModelConfig] = [
    {
        'type': 'GRU',
        'isBidirectional': True,
        'units': 128,
        'return_sequences': True,
        'dropout': 0.2,
    },
    {
        'type': 'GRU',
        'isBidirectional': True,
        'units': 128,
        'return_sequences': True,
        'dropout': 0.2,
    },
    {
        'type': 'GRU',
        'isBidirectional': True,
        'units': 64,
        'return_sequences': True,
        'dropout': 0.2,
    },
    {
        'type': 'GRU',
        'isBidirectional': True,
        'units': 64,
        'return_sequences': False,
        'dropout': 0.2,
    }
]
```

Figure 16: Configure of Test case 6

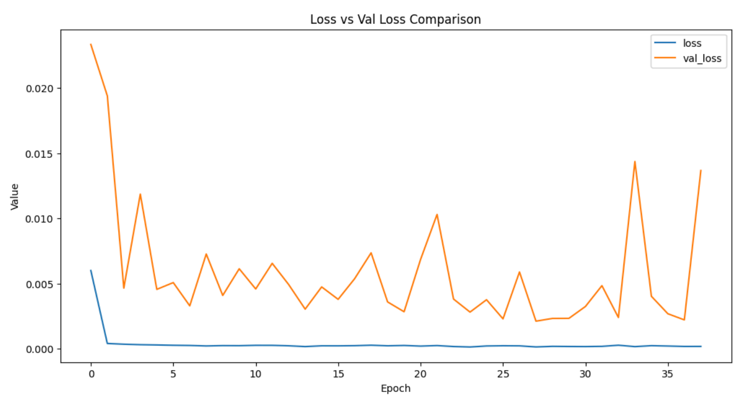

Figure 17: Result of Test case 6