

# Summary Report

**Student Name - Email**

Tai Minh Huy Nguyen - 104220352@student.swin.edu.au

**April 6th, 2024**

## Table of Contents

Introduction .....	3
System Architecture .....	4
Data Functioning.....	6
Data Processing .....	6
Machine Learning And Constructive Improvisation.....	8
Demonstration .....	10
Conclusion .....	11

# Introduction


As a team, we concentrated on AI-driven data modeling for precise future forecasting for a duration of 12 weeks. This paper presents the project's methodology, focusing on the application of Gated Recurrent Unit (GRU), Long Short-Term Memory (LSTM), and Recurrent Neural Networks (RNN).

This Summary Essay will be divided into 3 parts, alongside a Conclusion part:

- Overall system architecture
- Coding Techniques (This will include both implementing data processing and machine learning.)
- Examples to Demonstrate

I have made a clear document about my explanation to the extension of task B7 that I have been put inside. The document is named TaskB7 – Nguyen Tai Minh Huy.docx and you can check it through that data. I will be putting it alongside this document. Or you can check the link down here to read the task.

[https://github.com/heyytamvo/COS30018/blob/b38efd2cc45cfb357ea1279a10766d744d4c1efc/ProjectAssessment/TaskB7/104220352\\_TaiMinhHuyNguyen/SummaryReport/TaskB7%20-%20Nguyen%20Tai%20Minh%20Huy.pdf](https://github.com/heyytamvo/COS30018/blob/b38efd2cc45cfb357ea1279a10766d744d4c1efc/ProjectAssessment/TaskB7/104220352_TaiMinhHuyNguyen/SummaryReport/TaskB7%20-%20Nguyen%20Tai%20Minh%20Huy.pdf)

 TaskB7 - Nguyen Tai Minh Huy.pdf

Without further ado, we will start the summary report now.

# System Architecture

The system of this Stock Market Data can be split into 3 parts.

That is:

- Implementing Data Processing
- Machine Learning
- Constructive Improvisation (Which includes ARIMA, normalization, regularization etc. etc.)

Implementing Data Processing is the first step whenever you start putting in the raw data. This entails preparing the data and converting it into a format that our AI algorithms can analyze. In order to guarantee uniformity throughout the dataset, we manage missing values, clean the data to remove any inconsistencies or errors, and normalize the features throughout this step. Once the data is processed, we divide it into training, validation, and testing sets to train and evaluate our AI models effectively.

Machine Learning is the next crucial step where we will bring the processed data into the three models (RNN, GRU and LSTM). During this phase, we feed the preprocessed data into each model architecture and fine-tune their parameters to optimize performance. We compile the model, specifying the loss function, optimizer, and any additional metrics for evaluation.

## Long Short Term Memory (LSTM)

```
def LongShortTermMemory(layerNums=NUMBER_OF_LAYER,hidden_units=NUMBER_OF_HIDDEN_UNITS, loss_type=LOSS_FUNCTION,
                        optimizerType=OPTIMIZER, dense_unit=1, activation=["tanh", "linear"], dropoutRate = DROP_OUT_RATE):
    model = Sequential()

    for i in range(layerNums):
        if i == (layerNums - 1):
            model.add(LSTM(hidden_units, activation=activation[0]))
            model.add(Dropout(dropoutRate))
        else:
            model.add(LSTM(hidden_units, activation=activation[1], return_sequences=True))
            model.add(Dropout(dropoutRate))

    model.add(Dense(units=dense_unit, activation=activation[1]))
    model.compile(loss=loss_type, optimizer=optimizerType)
    return model
```

Constructive improvisation is an essential step after machine learning to improve the accuracy and resilience of our prediction models. This stage includes a number of methods and approaches, such as adding new models, regularization, normalization, and ARIMA (AutoRegressive Integrated Moving Average), among others.

## Autoregressive Intergrated Moving Average (ARIMA)

```
def ARIMAmodel(train, test, _order=ORDER):  
    history = [x for x in train]  
    y_predict = []  
    for t in range(len(test)):  
        model = ARIMA(history, order=_order)  
        model_fit = model.fit()  
        output = model_fit.forecast()  
        y_predict.append(output[0])  
        obs = test[t]  
        history.append(obs)  
    return y_predict
```

# Data Functioning

## Data Processing

In this part, we were assigned to start up the environment for our Stock Data Prediction. To start, we decided to pick Python as the main language for these Stock Predictions. Henceforth, using Jupyter Notebook, we have organized each of the structure of the code.

The main parts of the code revolves around setting up the necessities. To set up the whole model to run, simply press the play buttons at the side of each code up to down. You will get the results at the end of each task.

We will now move on to the functions.

Importing Dependencies: This is the command to import all the necessary dependencies.

```
# Task B2
import numpy as np
import pandas as pd
import yfinance as yf
import datetime as dt
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
import os

# Task B3
import plotly.graph_objects as go
import plotly.express as px

# Task B4
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, LSTM, GRU, Dense, Dropout, Attention, Bidirectional, Conv1D, MaxPooling1D, Flatten, Concatenate, BatchNormalization
import matplotlib.pyplot as plt
import sys
from tensorflow.keras.optimizers import Adam

# Task B5
from tensorflow.keras import metrics

# Task B6
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf

# Task B7
from sklearn.preprocessing import MinMaxScaler
```

Load and Splitting Data: These two will be used to read and sort out the data from the CSV file.

### Loading Data Function

```
def DataLoader(ticker=TICKER, start_date=START_DATE, end_date=END_DATE, scale=SCALE_DATA, scaling_method=SCALING_METHOD):
    """
    ticker: is the code of the target ticker
    start_date: a start date string with format YYYY/MM/DD
    end_date: an end date string with format YYYY/MM/DD
    scale: a boolean value, True by default
    scaling_method: MinMax(by default), Standard.
    """
```

## Splitting Dataset

```
def datasetSplitter(dataset: pd.DataFrame, look_up_days=LOOK_UP_DAYS,
                    training_ratio=TRAINING_RATIO,
                    feature_columns=['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume']):
    """
    dataset: a Pandas Dataframe
    training_ratio: is equal to TRAINING_RATIO constant
    """
```

And here is the Data Processor, A function to load and process a dataset with multiple features with the following requirements:

- + As inputs, provide the start and end dates for the entire dataset. enabling you to address the data's NaN problem
- + dividing the dataset in accordance with a predetermined train/test
- + Keeping the downloaded data locally on your computer for later usage
- + Giving you the choice to resize your feature columns and save the scalers in a data structure for later use.

```
def Data_Processing_1():
    dataLoader = DataLoader()

    scaledStockData = dataLoader["dataset"]
    datasetScaler = dataLoader["datasetScaler"]
    columnScalers = dataLoader["columnScalers"]

    dataset = datasetSplitter(dataset=scaledStockData)

    print("Loaded Done!\n\nThe result is a tuple as below:\n")
    print("(dataset, scaledStockData, datasetScaler, columnScalers), where:\n")
    print("dataset is a dictionary as below:")
    print("""{
        "X_training_set": {
            'Open': <class 'numpy.ndarray'>,
            'High': <class 'numpy.ndarray'>,
            'Low': <class 'numpy.ndarray'>,
            'Close': <class 'numpy.ndarray'>,
            'Adj Close': <class 'numpy.ndarray'>,
            'Volume': <class 'numpy.ndarray'>
        },
        "Y_training_set": {
            'Open': ...,
            ...
        },
        "X_testing_set": {
            'Open': ...,
            'High': ...,
            ...
        },
        "Y_testing_set": {
            'Open': ...,
            'High': ...,
            'Low': ...,
            ...
        }
    }""")
    print("scaledStockData is a Pandas Dataframe of the Stock Ticker (scaled)\n")
    print("datasetScaler is the Scaler of the dataset\n")
    print("columnScalers is a dictionary: each key is a DataFrame Feature('Open', 'High', etc.) and the correspond value is a scaler of that feature")
    return dataset, scaledStockData, datasetScaler, columnScalers
```

And all of these functions pretty much conclude the data processing part.

## Machine Learning And Constructive Improvisation

Machine Learning is the most crucial part of this project, as it requires for us to setup and train a perfect model of each scenario. These 3 models that we have been training throughout from Task B1 to task B6, hence, even B7, are GRU, LSTM and RNN. Using Python as a baseline language to train all of these models, we will be training them to give us the closest predictions ever, comparing the close Price.

The main difference between these two tasks is that I have added Normalization and Regularization inside the RNN code, this will be the same as the two other models (GRU and LSTM) as well.

Recurrent neural network (RNN) is made to handle sequential data by remembering details from earlier time steps. RNNs are different from standard feedforward neural networks in that they have loops in their architecture, which enables them to behave dynamically over time. Because of this feature, RNNs are especially well-suited for applications like speech recognition, natural language processing, and time series prediction.

### Recurrent Neural Networks (RNNs)

```
def RecurrentNeuralNetworks(layerNums=NUMBER_OF_LAYER,hidden_units=NUMBER_OF_HIDDEN_UNITS, loss_type=LOSS_FUNCTION,
                             optimizerType=OPTIMIZER, dense_unit=1, activation=["tanh", "linear"], dropoutRate = DROP_OUT_RATE):
    model = Sequential()

    for i in range(layerNums):
        if i == (layerNums - 1):
            model.add(SimpleRNN(hidden_units, activation=activation[0]))
            model.add(Dropout(dropoutRate))
        else:
            model.add(SimpleRNN(hidden_units, activation=activation[0], return_sequences=True))
            model.add(Dropout(dropoutRate))

    model.add(Dense(units=dense_unit, activation=activation[1]))
    model.compile(loss=loss_type, optimizer=optimizerType)
    return model
```

On the other hand, LSTM networks enable RNNs to capture long-term dependencies in data by resolving the vanishing gradient issue. For applications involving long-range temporal dependencies, such sentiment analysis and financial forecasting, their complicated memory cell structure is advantageous.



## Long Short Term Memory (LSTM)

```
def LongShortTermMemory(layerNums=NUMBER_OF_LAYER,hidden_units=NUMBER_OF_HIDDEN_UNITS, loss_type=LOSS_FUNCTION,
                        optimizerType=OPTIMIZER, dense_unit=1, activation=["tanh", "linear"], dropoutRate = DROP_OUT_RATE):
    model = Sequential()

    for i in range(layerNums):
        if i == (layerNums - 1):
            model.add(LSTM(hidden_units, activation=activation[0]))
            model.add(Dropout(dropoutRate))
        else:
            model.add(LSTM(hidden_units, activation=activation[1], return_sequences=True))
            model.add(Dropout(dropoutRate))

    model.add(Dense(units=dense_unit, activation=activation[1]))
    model.compile(loss=loss_type, optimizer=optimizerType)
    return model
```

Lastly, The Gated Recurrent Unit (GRU) is a streamlined variant of Long Short-Term Memory (LSTM) networks that provides similar functionality at a lower computational complexity. Sequence modeling tasks such as speech recognition and video analysis are areas in which they thrive.

## Gated Recurrent Unit (GRU)

```
def GatedRucurrentUnit(layerNums=NUMBER_OF_LAYER,hidden_units=NUMBER_OF_HIDDEN_UNITS, loss_type=LOSS_FUNCTION,
                       optimizerType=OPTIMIZER, dense_unit=1, activation=["tanh", "linear"], dropoutRate = DROP_OUT_RATE):
    model = Sequential()

    for i in range(layerNums):
        if i == (layerNums - 1):
            model.add(GRU(hidden_units, activation=activation[0]))
            model.add(Dropout(dropoutRate))
        else:
            model.add(GRU(hidden_units, activation=activation[0], return_sequences=True))
            model.add(Dropout(dropoutRate))

    model.add(Dense(units=dense_unit, activation=activation[1]))
    model.compile(loss=loss_type, optimizer=optimizerType)
    return model
```

ARIMA is an additional feature we add to improve the results of the model as well. It can reprocess the data before applying it to the models.

## Demonstration

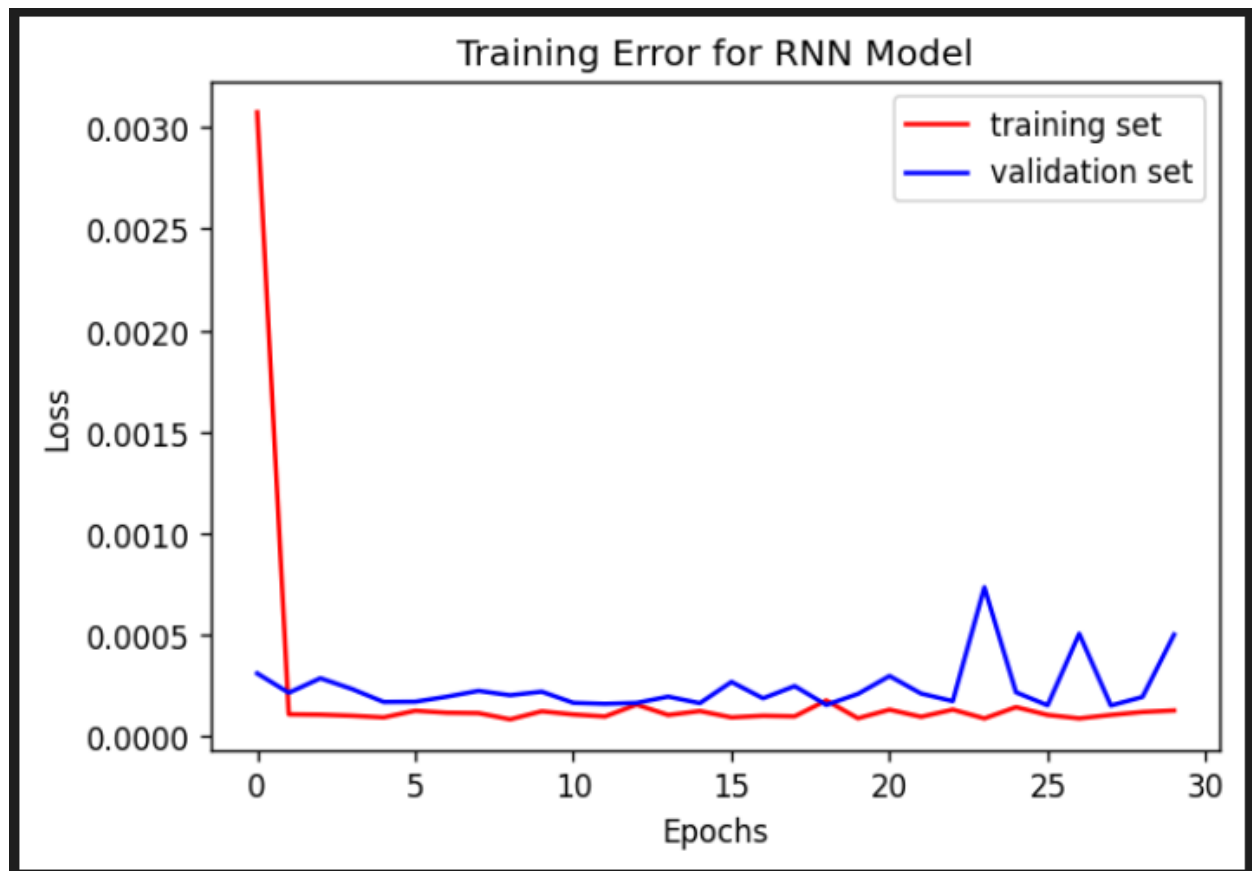
To run this program, simply press the Start button at each of the function, from top to bottom.

```
Importing Dependencies

# Task B2
import numpy as np
import pandas as pd
import yfinance as yf
import datetime as dt
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
import os

# Task B3
```

For each task, they will give you a different outcome, here is one example of Task B6 result.



## Conclusion

We stressed the experimental and iterative aspects of developing deep learning models throughout our project, emphasizing the value of ongoing improvement and optimization. Through the application of constructive improvisation approaches and the strengths of RNN, LSTM, and GRU architectures, our goal was to develop predictive models that would be accurate and dependable enough to predict future trends in a variety of fields, such as natural language processing, finance, and healthcare.

To sum up, our study is a thorough investigation of deep learning techniques and real-world applications in predictive modeling. We have gained important insights into the strengths and weaknesses of RNN, LSTM, and GRU architectures by fusing theoretical knowledge with practical experience, opening the door for further developments in the fields of artificial intelligence and predictive analytics.