**` COS30018**

**Intelligent Systems**

**Option B: Stock Prediction**

**Task B.4 – Machine Learning 1**

**Name: Duc Thuan Tran**

**Student ID: 104330455**

**Tutor: Dr. Ru Jia**

**Tutorial: Friday 2:30 – 4:30**

Table of Contents

[Introduction 3](#_Toc176870985)

[Overview of Previous Model Operations 3](#_Toc176870989)

[Enhancements in the Current Task 5](#_Toc176870990)

[Model Execution and Outputs 7](#_Toc176870991)

[Test Cases Summary and Results 10](#_Toc176870992)

[GitHub Repository 16](#_Toc176870993)

[References 16](#_Toc176870994)

# **Introduction**

# This report presents the development and enhancement of a stock prediction model using deep learning techniques, now updated to version v0.3 of the codebase. The primary goal of this task is to improve the flexibility and efficiency of the deep learning model, enabling it to predict stock prices based on historical data. The project makes use of advanced Recurrent Neural Networks (RNNs), specifically LSTM (Long Short-Term Memory), GRU (Gated Recurrent Units), and their bidirectional variants, to capture both short-term and long-term dependencies in time-series data.

# In this version (v0.3), significant improvements have been implemented, including the introduction of flexibility in the model creation process, allowing the use of different RNN architectures. Additionally, error handling in the predict\_next\_day function has been refined, and the model's structure has been enhanced with the option to include bidirectional layers for better sequential data analysis.

# This report details the changes made to the model\_operations.py file, the exploration of various RNN architectures, and the impact of hyperparameter tuning on model performance. The execution of the model, along with its outputs and the source code, is documented and made available in the GitHub repository for further review and testing.

# **Overview of Previous Model Operations**

In version v0.1 of the stock prediction model, the deep learning architecture was relatively straightforward, consisting of a sequential stack of LSTM layers. The model was designed to capture temporal dependencies in stock prices using the following structure:

1. **LSTM Layers:**
   * Three LSTM layers, each with 50 units, were used to model the sequence of stock prices.
   * The return\_sequences=True parameter was applied to the first two LSTM layers, ensuring that the full sequence output was passed to the next LSTM layer.
   * The final LSTM layer returned only the last hidden state as input to the subsequent Dense layer.
2. **Dropout Layers:**
   * A Dropout layer with a rate of 0.2 was added after each LSTM layer to prevent overfitting and enhance the model’s generalization ability.
3. **Dense Output Layer:**
   * The model concluded with a Dense layer with one unit to predict the final stock price for the next day.
4. **Compilation:**
   * The model was compiled using the Adam optimizer with a mean\_squared\_error loss function, suitable for regression tasks like stock price prediction.

Overall, this version of the model was effective but limited in its flexibility and adaptability to different deep learning architectures. The fixed use of LSTM layers and static parameters did not allow for experimentation with other RNN types, bidirectionality, or more advanced configurations, which were introduced in subsequent versions.

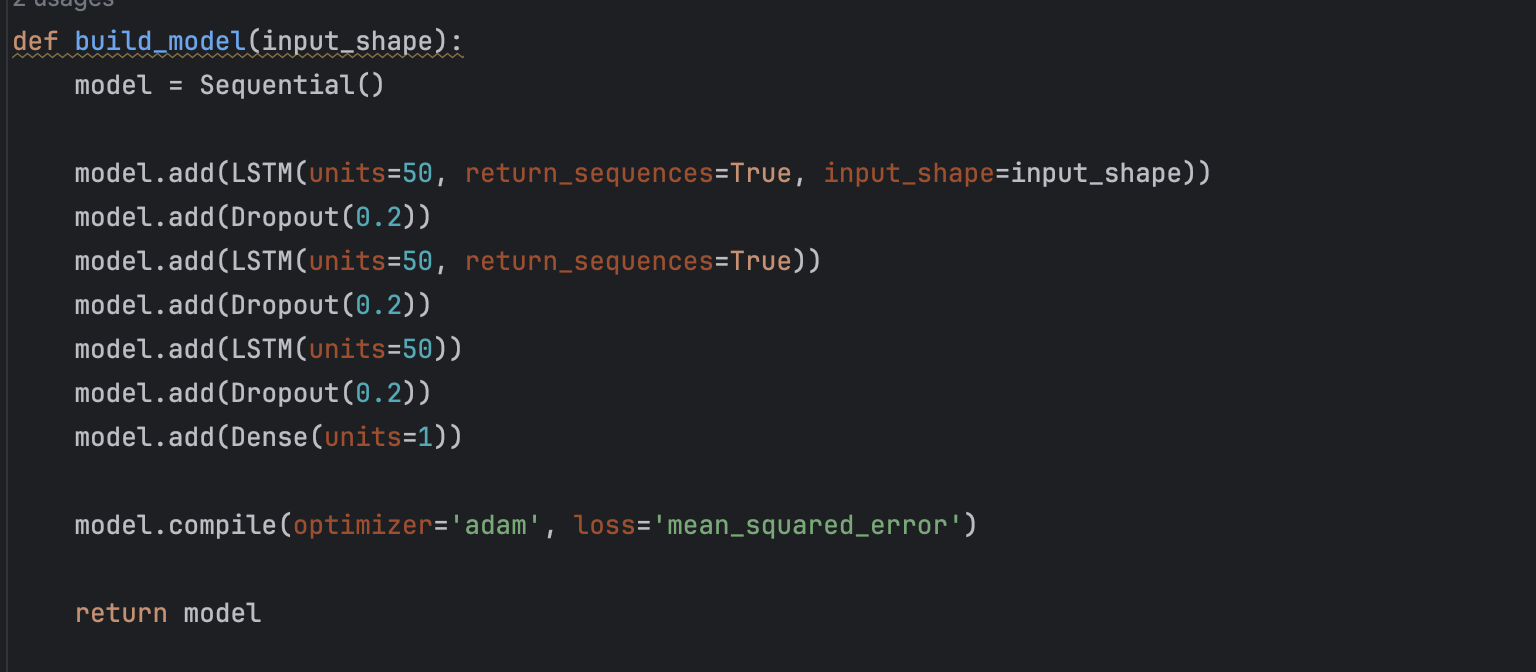


Image 1. build\_model function version v01

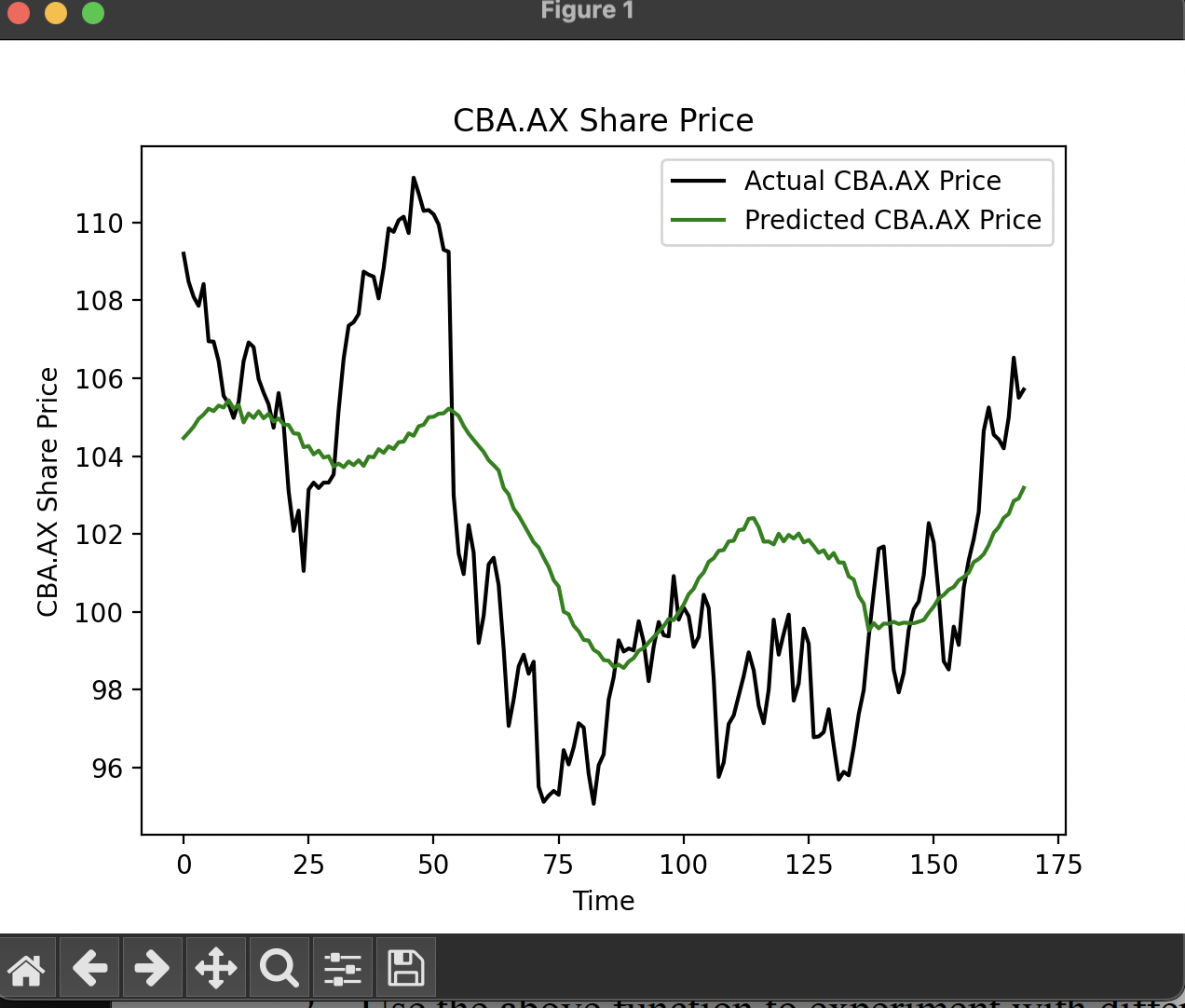


Image 2. Stock-prediction result version v01, v02

# **Enhancements in the Current Task**

In version v0.3, the stock prediction model has undergone significant enhancements to increase its flexibility and adaptability. These improvements include support for different Recurrent Neural Network (RNN) types, such as LSTM, GRU, SimpleRNN, and their bidirectional variants. The enhanced model allows for better experimentation with various architectures and hyperparameters, improving its ability to capture the complexities in stock price prediction. Below are the specific changes introduced in this version:

**1. Layer Type Flexibility:**

* **Previous Version (v0.1):** The model was limited to using only LSTM layers, which, although powerful, did not allow for easy experimentation with other RNN types.
* **Current Version (v0.3):** The build\_model function now includes a layer\_type parameter, which allows the user to select between multiple RNN layer types:
  + **LSTM (Long Short-Term Memory):** Suitable for capturing long-term dependencies in the sequence.
  + **GRU (Gated Recurrent Unit):** A computationally efficient alternative to LSTM that still handles long-term dependencies well.
  + **SimpleRNN:** A basic RNN layer for simpler tasks.
  + **BiLSTM (Bidirectional LSTM):** Captures dependencies in both directions, enhancing performance for tasks where future context is important.
  + **BiGRU (Bidirectional GRU):** Similar to BiLSTM but with fewer parameters, making it faster to train.
* The inclusion of these options greatly enhances the model’s versatility, allowing it to adapt to different types of sequential data and prediction tasks.

**2. Conditional Layer Handling:**

* **First Layer:** The build\_model function checks if the num\_layers parameter is greater than 1 to decide whether the first recurrent layer should return sequences. This ensures that the model can handle both single-layer and multi-layer architectures.
* **Subsequent Layers:** Additional layers are added based on the value of num\_layers, allowing for dynamic depth in the model. Each of these layers inherits the specified layer\_type and applies the return\_sequences parameter conditionally.

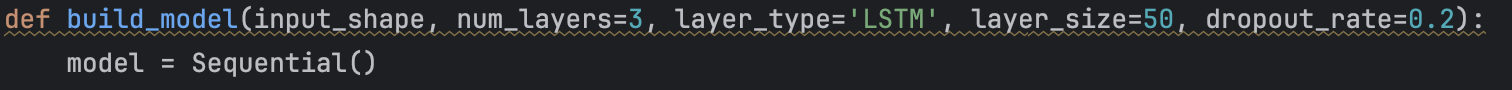


Image 3. Build model function with configurable compilation

**3. Introduction of Bidirectional Layers:**

* In addition to standard LSTM, GRU, and SimpleRNN layers, the enhanced model also supports **Bidirectional LSTM (BiLSTM)** and **Bidirectional GRU (BiGRU)** layers.
* **Bidirectional Layers:** Process the sequence both forward and backward, improving the model’s ability to capture dependencies from both past and future time steps. This is particularly useful in tasks like text processing or time series prediction, where context from both directions can improve accuracy.

A screen shot of a computer code

Description automatically generated

Image 4. First Rnn layer

**4. Dropout for Regularization:**

* Dropout layers are included after each RNN layer to prevent overfitting. The dropout rate is controlled by the dropout\_rate parameter, which remains at 0.2 by default. This ensures that the model remains generalizable even when dealing with complex datasets.

A screenshot of a computer program

Description automatically generated

Image 5. Remaining Rnn layers

**5. Output Layer and Model Compilation:**

* The output layer remains a single-unit Dense layer, as in previous versions, suitable for the regression task of predicting stock prices.
* The model is compiled using the **Adam optimizer** and the **mean squared error (MSE)** loss function, which are standard choices for regression tasks and ensure stable training.

A computer screen with text

Description automatically generated

Image 6. Output layer and Model Compilation

A black screen with white text

Description automatically generated

Image 7. New Neural models adding

**Summary of Changes:**

* **Layer Type:** Added support for LSTM, GRU, SimpleRNN, BiLSTM, and BiGRU.
* **Dynamic Depth:** The number of recurrent layers (num\_layers) is customizable, allowing the model depth to scale based on task complexity.
* **Bidirectional Layers:** Introduced bidirectional variants (BiLSTM, BiGRU) for improved context understanding.
* **Improved Regularization:** Dropout layers help prevent overfitting, making the model more robust.
* **Output and Loss:** The model retains its single-unit Dense output for regression, with MSE as the loss function.

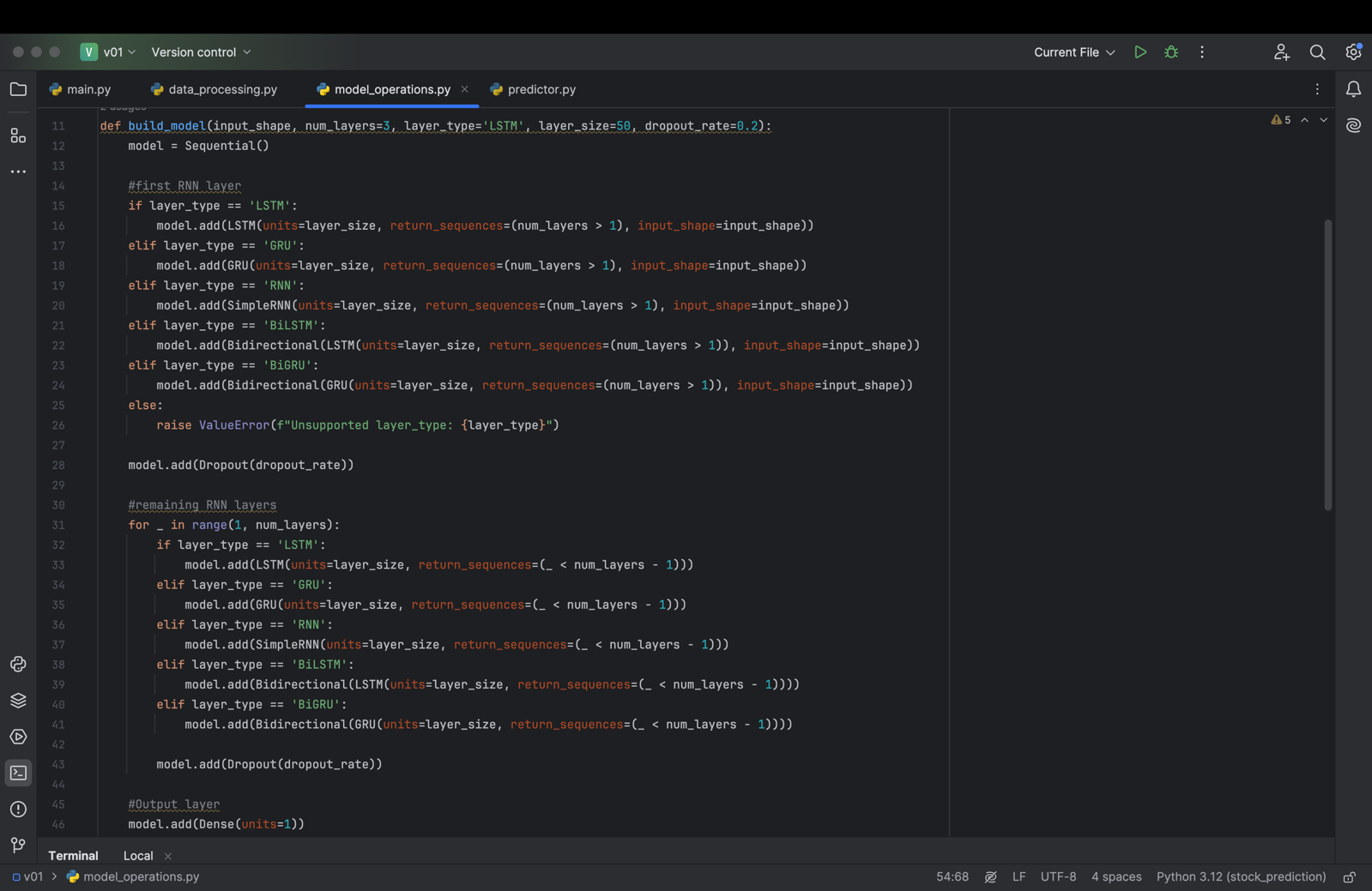


Image 8. build\_model function version 3.0

These enhancements provide more flexibility for model experimentation, allowing users to better tailor the architecture to their specific needs. By offering options for different RNN types and bidirectional processing, the model is now better suited to handle a broader range of sequential prediction tasks.

## **Model Execution and Outputs**

To execute the model and observe the stock prediction results, the script was run from the main.py file. This version of the model was set to run for 25 epochs, following the adjustments made to the deep learning architecture. These modifications were specifically made in the build\_model function, which now includes flexibility for different recurrent neural network (RNN) types, such as LSTM, GRU, SimpleRNN, and their bidirectional variants. The model was configured to use a **GRU** architecture with **4 layers**, **100 units** per layer, and a **dropout rate of 0.3**.

**Key Changes in the Main Code (main.py):**

1. **Model Configuration:**
   * The main code now includes the updated build\_model function, allowing us to define the model with specific configurations. In this execution:
     + num\_layers=4
     + layer\_type='GRU'
     + layer\_size=100
     + dropout\_rate=0.3
   * This configuration was chosen to balance complexity with computational efficiency while still capturing both short-term and long-term dependencies in the stock price data.

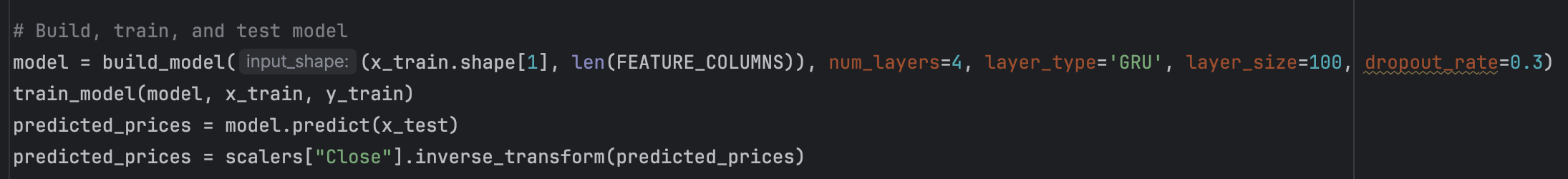


Image 9. Model Execution with GRU Configuration

1. **Model Training:**
   * The model was trained on the preprocessed stock price data for **25 epochs**, which involved fitting the model on the training data (x\_train, y\_train). During each epoch, the model adjusted its weights using backpropagation, aiming to minimize the mean squared error (MSE) between predicted and actual stock prices.
   * The execution took some time to complete as the GRU-based model iterated over the dataset multiple times, adjusting its internal state based on the input sequence.

Image 10. Model training

1. **Model Prediction:**
   * After training, the model was used to predict stock prices on the test dataset (x\_test). The predicted prices were then inverse transformed using the previously fitted scaler to bring them back to the original scale for comparison with the actual prices.
2. **Stock Prediction Results:**
   * Once the model had finished training and predictions were made, the output graph displaying both **actual** and **predicted** stock prices was generated. Compared to the previous version of the model (v0.1), which used only LSTM layers, the results with this GRU-based model showed noticeable improvements.
   * The predicted stock prices more closely followed the trends of the actual prices, particularly in areas where the LSTM model struggled with sudden fluctuations. The GRU's ability to handle long sequences efficiently contributed to a smoother prediction curve.

A graph of a graph showing a price

Description automatically generated with medium confidence

Image 11. Stock-prediction result verion v03

**Comparison with Previous Versions:**

* In the earlier version of the model, the predicted stock prices were more prone to lag behind the actual stock prices, especially when sudden market shifts occurred. The introduction of the GRU architecture with 4 layers, combined with the slightly higher dropout rate, helped prevent overfitting while improving the model's ability to track both upward and downward trends in stock prices.
* The graph now shows better alignment between the **actual** and **predicted** prices, with a reduction in the divergence between the two curves, particularly in volatile periods.

## **Test Cases Summary and Results**

The purpose of this test case section is to evaluate and compare different Deep Learning (DL) architectures—specifically **LSTM**, **GRU**, and **RNN**—for time series data related to stock price prediction. To achieve this, I created a new Python file named model\_experiments.py. This script is responsible for running and testing many configurations across these different DL models. The idea was to observe how changes in hyperparameters such as the number of layers, units per layer, batch sizes, and epochs affect the performance and efficiency of each model type.

**Model Experiments Script (model\_experiments.py)**

The model\_experiments.py file contains the function run\_model\_experiments() which systematically tests the following configurations:

* **Model Types**: LSTM, GRU, and RNN.
* **Number of Layers**: 2, 3, and 4 layers.
* **Units per Layer**: 50, 100, and 150 units.
* **Epochs**: 25 and 50 epochs.
* **Batch Sizes**: 32 and 64.

A screenshot of a computer code

Description automatically generated

Image 12. Model Configurations

After defining the configurations, the script loops through each possible combination of the above parameters. It constructs a model using the build\_model function, which dynamically adjusts the model architecture based on the current parameters (number of layers, units per layer, and model type).

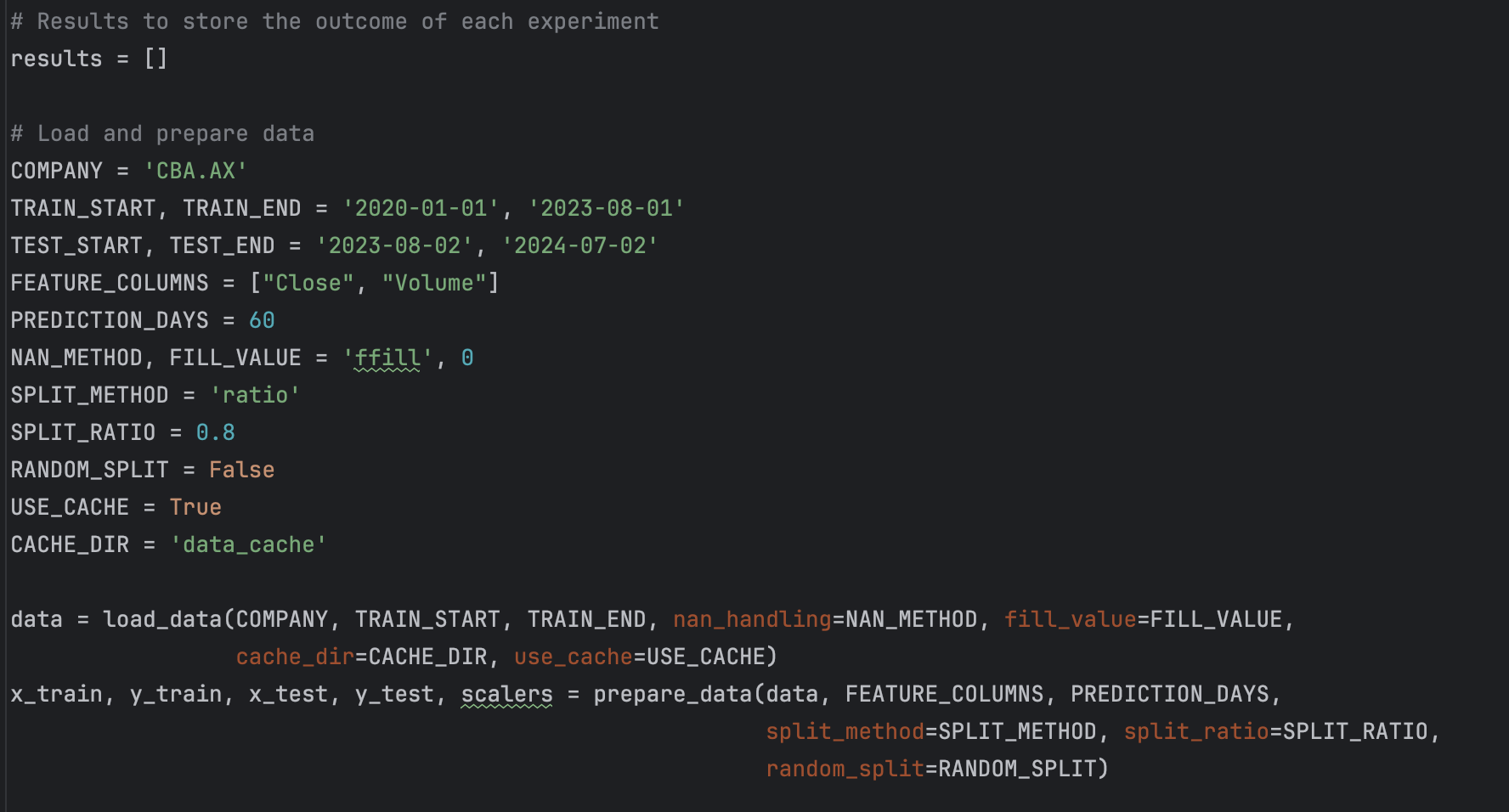


Image 13. Data Loading and Preparation Configurations

For each combination, the model is trained using the train\_model function, which takes the training data, the specified number of epochs, and the batch size. The training process records the **training loss**, **validation loss**, and the **time taken** for each model configuration.

These results are then logged and saved into a CSV file for further analysis. The results include:

* **Model Type** (LSTM, GRU, RNN)
* **Number of Layers**
* **Units per Layer**
* **Batch Size**
* **Epochs**
* **Training Loss**
* **Validation Loss**
* **Time Taken** (in seconds)

A computer screen shot of a program

Description automatically generated

Image 13. Model Training and Evaluation Loop

Given that there are multiple combinations of these hyperparameters, the total number of test cases is over 100. This script automates the process, training and evaluating each model configuration in turn. Since each model is trained for 25-50 epochs, the process takes considerable time—**over 30 minutes to complete** on average.

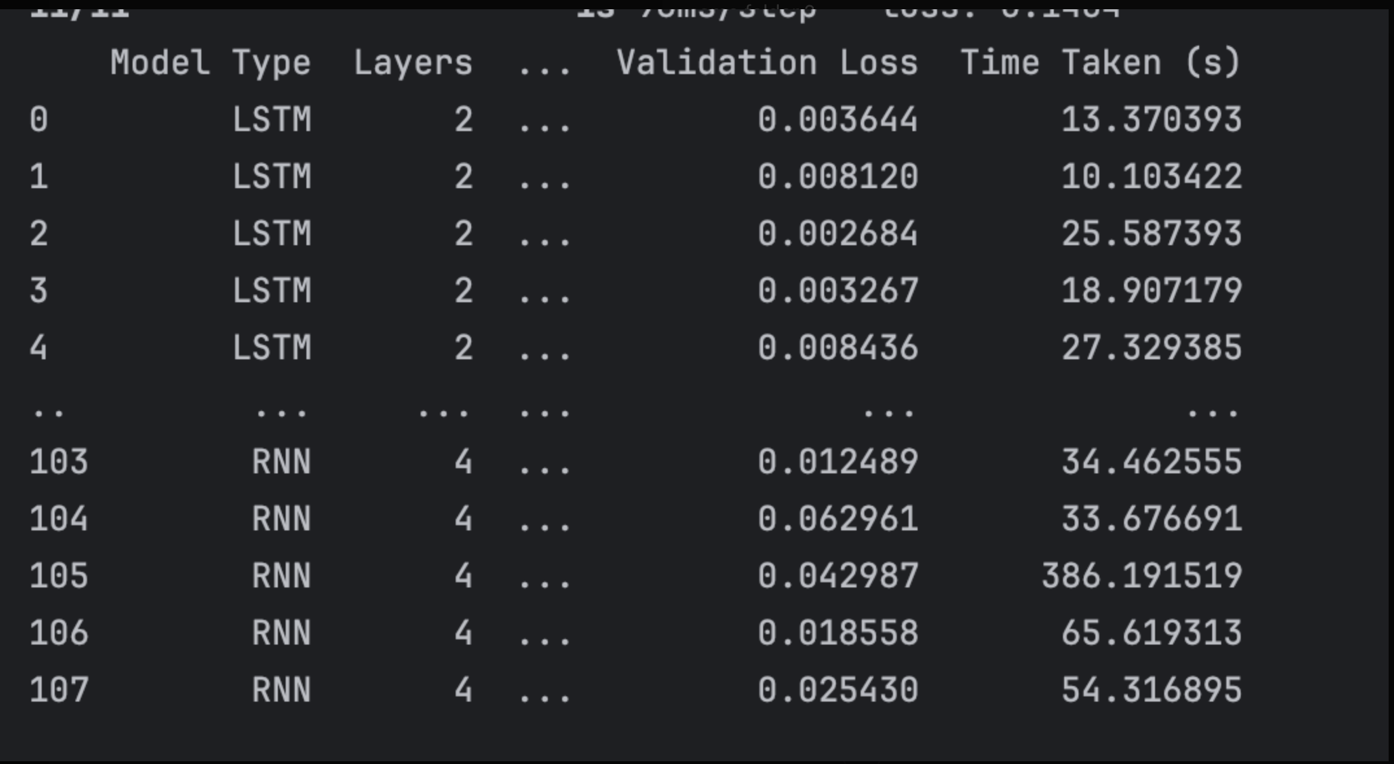


Image 14. Model Experiment Results

A table of numbers with numbers

Description automatically generated

Image 15. LSTM Cases

A table of numbers with numbers

Description automatically generated

Image 16. GRU Cases

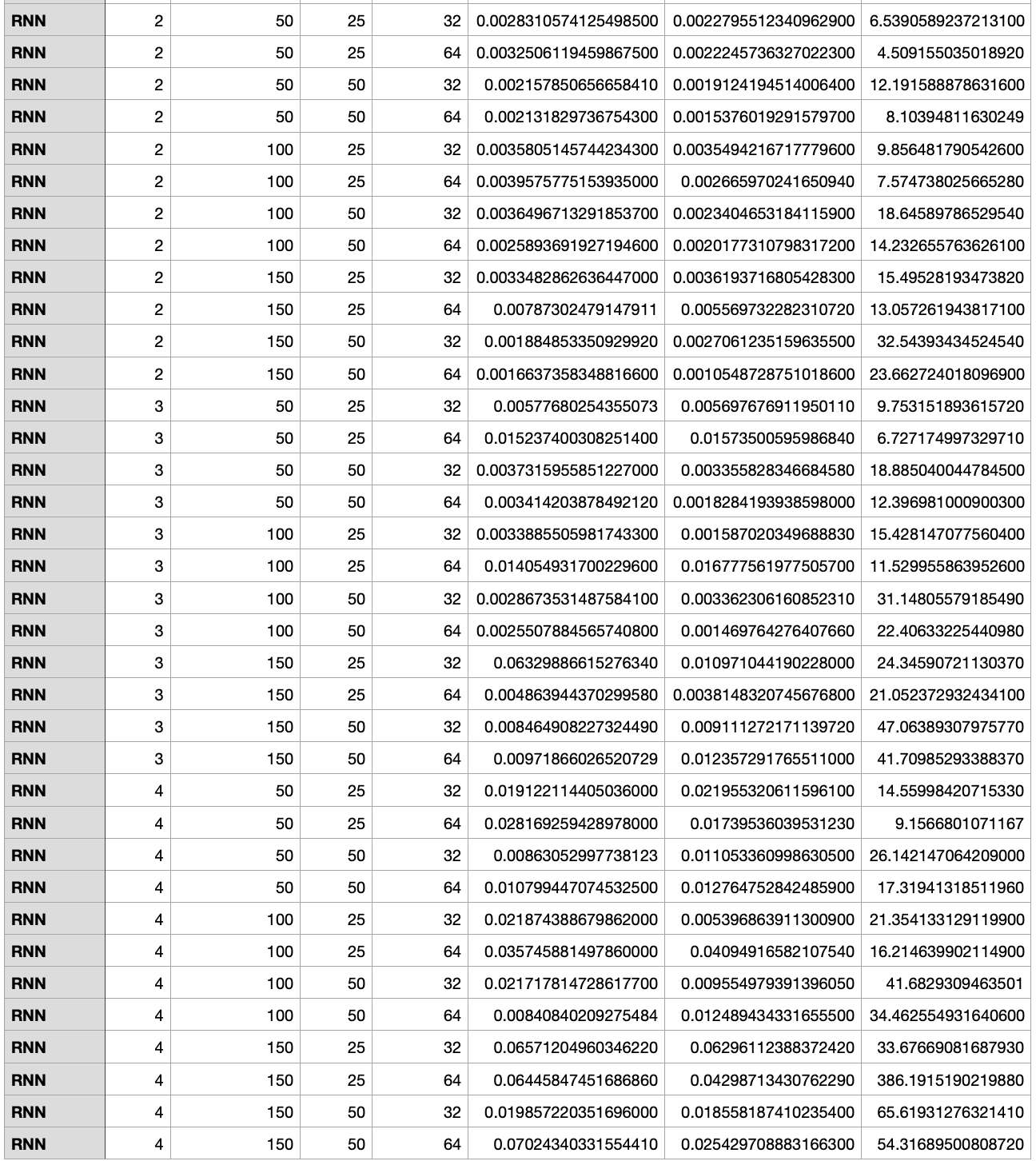


Image 17. RNN Cases

Once the process is finished, the results—including **model type**, **number of layers**, **units per layer**, **training loss**, **validation loss**, and **time taken**—are stored in a CSV file named model\_experiment\_results.csv. This file serves as a record of each experiment, allowing us to analyze and compare the performance of different models based on their configurations.

A computer screen with green text

Description automatically generated

Image 18. Saving Result to CSV file

**Key Observations**

The purpose of this section is to analyze the results of the experiments using data visualization. To better understand the differences between the models and their performance, I created a Jupyter Notebook file. This file is dedicated to visualizing the experimental data and extracting meaningful insights from the CSV file. Using libraries like **matplotlib** and **pandas**, I plotted graphs comparing validation loss, training loss, and time taken for each model type (LSTM, GRU, and RNN) across different configurations.

A screenshot of a computer

Description automatically generated

Image 19. Jupyter Notebook Setup for Model Experiments Analysis

The visualizations helped us break down the results and derive the following observations:

**Validation Loss vs. Units per Layer**

* **LSTM and GRU** consistently performed better than **RNN** in terms of validation loss across all configurations. This demonstrates that these models are better suited for learning complex patterns in time-series data, such as stock price prediction.
* **RNN models** displayed much higher variance in validation loss, especially with a larger number of units. This suggests that the simple architecture of RNN struggles with scalability when dealing with a large number of units.
* **LSTM and GRU** models exhibited relatively stable validation losses as the number of units increased, indicating they were more robust across different architectures.

A screen shot of a graph

Description automatically generated

Image 20. Validation Loss vs Units per Layer

**T Training Time vs. Units per Layer**

* **GRU and LSTM** models took more time to train as the number of units increased. However, **RNN** models—while initially faster—became slower and less efficient as the number of units increased.
* **GRU** showed a significant advantage in terms of training time efficiency. It consistently outperformed LSTM in terms of training speed while maintaining comparable or even better validation loss.
* **RNN** models started off faster, but with larger architectures (more layers and units), their time advantage diminished, making them less appealing for large-scale models.

A screen shot of a graph

Description automatically generated

Image 21. Time Taken vs Unit per layer

From the results of the experiments, we can conclude that **LSTM** and **GRU** models are better united for stock price prediction. However, if both performance (validation loss) and efficiency (training time) are equally important, **GRU** would be the model of choice. It provides a good trade-off between model accuracy and training efficiency.

**Recommendations:**

* For tasks with limited computational resources but still requiring high accuracy, **GRU with 2-3 layers and 50-100 units per layer** is recommended.
* If computational time is less of a concern and the focus is on accuracy, **LSTM with 3-4 layers and 100 units per layer** may provide slightly better generalization on unseen data.
* **RNN** models are not recommended for this task as they show unstable validation loss with increased complexity and longer training times.

## 

## **GitHub Repository**

The project's code base is hosted on GitHub for version control and review. Everyone can access the repository via the following link: [GitHub Repository](https://github.com/ThuanDanchoi/COS30018/tree/main). This repository contains all necessary files for the project and is available for the tutor to review the work.

# **References**

* https://www.youtube.com/watch?v=UuBigNaO\_18NeuralNine. (2022, October 1). *Stock Price Prediction using LSTM in Python*. [Video]. YouTube. Retrieved from
* <https://www.geeksforgeeks.org/best-python-libraries-for-machine-learning/>
* <https://commtelnetworks.com/exploring-the-depths-unraveling-the-intricacies-of-machine-learning-and-deep-learning/>
* <https://www.tensorflow.org/api_docs/python/tf/keras/Sequential>

# 