Advanced Data Analytics

Nelson Heinzelmann
Gaëtan Gallandat
Elyès Ghali
Raphaël Radzuweit
Faculté des Hautes Études Commerciales
Université de Lausanne

Abstract. This paper presents an AI that aims to predict the futures prices in a trading game developed in the Advanced Programming class. The trading game involves investment decisions of a portfolio consisting of bonds, stocks, and cash based on four variable pieces of information: Inflation, Fed rate, Bond prices, and Stock prices. The player has to adjust the weights of the portfolio at each period to attain a maximal return at the last period of the game. The main challenge is that these variables are randomly generated for each game, thus creating new optimal decisions for each game. The AI must process large amounts of data to determine the best portfolio prices. Our objective is to develop an LSTM model that utilizes the current state of the four primary variables to forecast future prices. We employ a rolling window technique to consider past values to achieve this. Additionally, since our game is implemented in Python, we will utilize TensorFlow for its development. Despite notably elevated Mean Squared Error (MSE) and Mean Absolute Error (MAE) values, particularly in the case of bond prices, our LSTM model still significantly outperforms a random decision model.

1. Introduction.

Artificial intelligence (AI) has emerged as a transformative field of study and practice, revolutionizing various aspects of our lives. At its core, AI seeks to replicate human intelligence and cognitive abilities in machines, enabling them to perform complex tasks and make decisions with little or no human intervention. One area where AI, specifically neural networks, has shown great promise is in the realm of financial portfolio management. Neural networks serve as the backbone of AI systems, allowing machines to learn from data, recognize patterns, and make predictions or decisions based on the information they have acquired. These networks consist of interconnected nodes, called artificial neurons. By organizing these neurons into layers and leveraging mathematical algorithms, neural networks can analyze vast amounts of financial data.

Financial portfolio management involves the allocation and management of investments to achieve optimal returns while managing risk. Traditionally, portfolio management decisions have relied on the expertise and intuition of human fund managers, who analyze market trends and economic indicators to make investment choices. However, the complexity and volatility of financial markets often make it challenging for humans to identify the most profitable and risk-efficient investment strategies. Here is where neural networks come into play. By leveraging their ability to process and analyze large volumes of financial data, neural networks can assist in portfolio management by identifying patterns, trends, and correlations that human analysts may overlook. These networks can learn from historical market data, economic indicators, news sentiment analysis, and other relevant factors to generate insights and make informed investment recommendations. Neural networks can also adapt to changing market

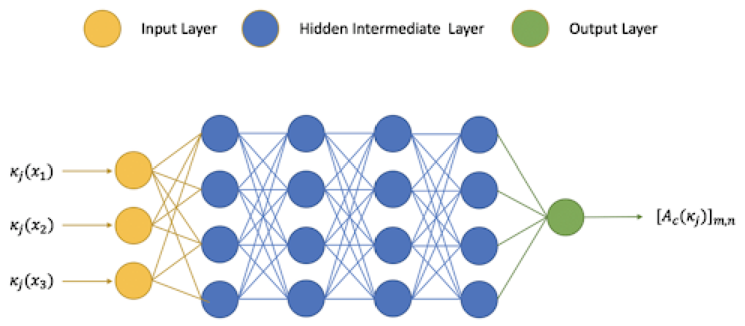

Figure 1.1. *A deep neural network*

conditions and adjust investment strategies accordingly. Through a process called training, neural networks learn from past performance and continuously refine their predictive models. This allows them to adapt to new market trends and optimize portfolio allocations in real-time, maximizing returns and minimizing risks.

2. Research question. In the realm of investment decision-making, the performance of predictive models in asset allocation decisions holds significant importance. Selecting and engineering appropriate features from the input data is a crucial aspect of building effective neural network models. However, the ultimate focus of this research is to evaluate the performance of the LSTM model in making accurate asset allocation decisions within the trading game some of us developed previously. Our primary research question revolves around understanding the performance of the LSTM model in predicting future bond prices and stock prices, which are the key outputs of the model. Specifically, we seek to answer the following question: "What is the performance of the LSTM model in making asset allocation decisions in the scope of a trading game?" Making predictions on financial time series has already widely been studied. The LSTM model is often used for its nice properties. One good example is the study of Arif Istiake Sunny et al(2020) [1] where they find that this type of model can produce high-accuracy predictions if used with the right parameters. Then in recent years, with the rapid growth in the field of machine learning, more complex studies have been published utilising some hybrid models. For example Wu et al.(2021) [2] discuss the use of CNN together with LSTM. In the context of the trading game we were interested in, we decided to use a more simple approach than the studies we just mentioned.

3. Methodology. This section outlines the methodology employed in this study to predict future prices and make investment decisions based on these forecasts. The methodology consists of three primary stages: data preparation, model development, and model evaluation. Each stage is essential for the successful execution of the project.

3.1. Data Preparation. The data preparation phase involved utilizing the original game file, which underwent slight modifications to align with the data preparation process. These modifications included removing game windows, keyboard inputs, and rendering functions while adding getters to extract relevant data.

To generate a comprehensive dataset, 20,000 games were simulated. Each game consisted of 30 periods, which were further divided into two distinct phases: starting periods and decision-making periods. The starting periods encompassed the initial 12 periods, during which players began with a cash allocation of 100% and zero allocations for bonds and stocks. The subsequent 18 periods were designated as decision-making periods, during which random decisions were simulated. Additionally, we incorporated a 6-period rolling window for the input variables to capture more information and potentially improve the precision of our forecasts.

Once the dataset was generated, it was split into training and testing sets using an 80/20 split ratio. This ensured that 80% of the data was utilized for training the LSTM model, while the remaining 20% was reserved for evaluating its performance. A more detailed explanation of the data preparation process will be provided in the subsequent section, dedicated specifically to the data analysis and preparation.

3.2. Model Development and Training. Our methodology involves the development of a deep neural network using TensorFlow, an open-source platform for machine learning. The model is designed with four input nodes representing the variables of Bonds Price, Stock Price, Inflation, and Fed Rate, and two output nodes representing the next periods' bond and stock prices. To determine the optimal structure of the model, we employed a random search method for hyperparameter tuning.

During the hyperparameter tuning process, we conducted 32 trials to fine-tune our model's architecture and optimize its performance. This involved exploring various combinations of hyperparameters, including the number of layers, the number of nodes per layer, and the dropout rate. We explored various options for the number of layers, ranging from 1 to 3, and different numbers of nodes per layer, including 16, 32, 64, and 128. This allowed us to find the optimal balance between model complexity and generalization ability, capturing relevant patterns and relationships in the data. To mitigate overfitting and enhance the model's generalization capability, we incorporated dropout layers after each LSTM layer. Dropout rates of 0, 0.1, 0.2, 0.3, 0.4, and 0.5 were tested to identify the most effective rate. By employing the random search method and selecting the optimal model architecture, we aim to enhance the performance of our deep neural network in making accurate predictions for asset allocation decisions.

The model development and training phase is a crucial component of our research methodology. After generating a substantial amount of data through game simulations, we proceed to train our model using this data. To ensure an efficient balance between computational cost and model performance, we adopt a systematic training approach with a batch size of 32. The model undergoes training for 100 epochs, during which an early stopping mechanism is implemented. This mechanism monitors the model's performance on the validation set, specifically tracking the validation loss. If there is no improvement in the validation loss for a defined number of epochs (in this case, 3 epochs), the training is stopped early. This early stopping strategy helps prevent overfitting, reduces computational time, and lowers the overall computational cost associated with training the model.

Additionally, the same training process, including 100 epochs, a batch size of 32, and the implementation of early stopping, is applied during the hyperparameter tuning phase to optimize the model's architecture and achieve the best possible performance. By implementing early stopping, we ensure that our model remains sufficiently complex to capture important patterns and relationships in the data while avoiding excessive complexity that may hinder generalization to unseen data. This approach not only enhances the model's ability to generalize well and make accurate predictions on new, unseen data but also reduces the computational resources required during the training process.

To further aid in model generalization and provide a robust estimate of its performance, a portion of the training data is held out as a validation set during the training process. This validation set, distinct from the final test set, allows for the hyperparameters to be tuned and the model's learning progress to be monitored. This rigorous training strategy ensures that our model is well-equipped to make optimal decisions in the game simulations. During training, the model's performance is monitored on both the training and validation datasets. The validation dataset is a subset of the training data that the model has not seen during training. It is used to ensure that the model is not merely memorizing the training data but is learning to generalize from it. The performance is visualized through a plot of training and validation losses as a function of epochs, providing insights into the model's learning progress and allowing us to assess its performance.

3.3. Model Evaluation and Applying Trained Models in Simulations. The last step is to assess the performance of our trained models using various evaluation metrics on the test set, which was derived from the original 80/20 data split. Specifically, we calculate the Mean Squared Error (MSE), Mean Absolute Error (MAE), and R-squared (R^2) to gauge the accuracy and goodness of fit of our models.

To further evaluate the effectiveness of our models in practical scenarios, we employ the best-performing model to simulate 100 games. During the decision-making periods of each game (which consists of 18 periods), we utilize the model's predictions to forecast the prices of the next period. Our investment strategy is designed to align with the constraints present in the original game, where negative weights are not allowed and the weights must sum up to 1.

Here is how our investment approach works: If the return of an asset is predicted to be negative, we allocate 0% of our portfolio to that asset. In the scenario where both assets have negative return forecasts, we allocate 100% of our portfolio to cash. On the other hand, if an asset has a positive return forecast, we distribute our investment proportionally based on the predicted total return of both assets. It's important to note that when the return forecast for an asset is negative, we set it to 0 to avoid any negative allocation.

By applying this investment strategy across the simulated games, we can assess the performance and profitability of our model's in-game scenarios. This process allows us to gain insights into the effectiveness of our forecasting approach and its potential impact on investment decision-making. In order to establish a benchmark for comparison, we conducted a simulation of 100 games using a random decision-making approach.

4. Data Description and Preprocessing.

4.1. Data Description. As mentioned in the previous section, we used the game logic to simulate 20000 games, which each consist of 12 + 18 periods. Given the stochastic nature of the variables, each round of the game generates a unique combination of data points, resulting in a diverse and dynamic dataset. Over multiple rounds, we collected a rich and varied dataset, covering a wide range of potential scenarios, thereby providing us with a robust foundation for training our model. In addition, our data is simulated using ARMA processes. This process simulates data in such a way that captures specific behaviours often observed in real-world temporal data.

4.2. Data Preprocessing. The Data Preprocessing is divided into several stages:

1. Data Generation: Generating a large amount of data, as explained previously.
2. Reshaping Data: Once the data is generated, it needs to be reshaped to match the input format expected by the LSTM model, which requires a 3D input. We use a rolling window approach, resulting in a 3D format: (number of samples, size of rolling window, number of features), denoted as (sample, timestamp, feature).
3. Train-Test Split: After reshaping, the data is split into training and test sets, with an 80/20 ratio. This allows us to train the model on a portion of the data (training set) and then evaluate its performance on unseen data (test set).
4. Data Scaling: The final step in the data preparation process involves scaling the input data to ensure that the model is not biased towards variables with higher magnitudes. We use a MinMaxScaler to get the data to [0,1]. We then fitted the scaler on the training data and used this fitted scaler on the test data. This ensures that the scaling is consistent across the data and that the test data is transformed in the same way as the training data. It's important to note that the output data is not scaled in this process as we made this arbitrary decision for the purpose of simplification. This approach may not be optimal, but we will discuss its limitations in more detail later on.

5. Implementation. Now, we'll discuss how all of what we have mentioned until now has been implemented from a code-wise perspective. To begin with, our code has the following structure:

```
Data_Analysis_Project
 ├── scenes_ai.py
 ├── nn_model.py
 ├── main_ml.py
 ├── ml.py
 ├── data_processing.py
 ├── train_model.py
 ├── model/
 └── HyperTunning_Trials/
```

5.1. Data Generation and Preprocessing. The data for our model is generated and processed through the `DataProcessing` class in the `data_processing.py` script. This class is responsible for simulating investment games and generating the data used for training and testing the model. It contains three methods: `generate_data` which does the random generation using the game logic defined in the `scenes_ai.py` script which the number of games as input, which we set to 20'000.

Then the `get_train_test_split` which first setup the input features ($\mathbf{X}$) using a sliding rolling window of 6, the next period stock and bonds prices are then used as the prediction target ($\mathbf{y}$). This rolling window approach allows the model to capture temporal dependencies and patterns in the data. The generated data is then split into training and testing datasets using an 80/20 split ratio, and normalized using the MinMaxScaler of the `sklearn` library. At the same time, the input data is reshaped to fit the 3D shape mentioned earlier.

5.2. Model Architecture. The model architecture is defined in the `create_model` function within the `nn_model.py` script. The model is constructed using the Keras Sequential API and consists of LSTM layers. The first layer is designed to handle the four input features, and subsequent layers (excluding the last layer) are added dynamically using a loop. As the final layer should not return a sequence we did not specify `return_sequences=True`. Additionally, the output layer is added with two outputs. To enhance the LSTM model's performance, we incorporate `tf.keras.layers.Dropout` layers after each LSTM layer. The dropout rate is determined during the hyperparameter tuning phase.

For optimization, we utilize the widely used `Adam` optimizer. Its effectiveness has been demonstrated in various applications, including LSTM-based models. The activation function for the LSTM layers is the standard `tanh`, while the output layer employs the `linear` activation function. Finally, the mean squared error (MSE) is employed as the loss function. This architecture configuration enables the effective processing of time series data. Also, by adding a layer through a loop, and having all parameters set as attributes of the function, we can easily run a hyperparameter tuning where we try a lot of different possible architectures.

5.3. Model Compilation and Training and Testing. For model compilation and training, we have implemented the `train_model` function in the `train_model.py` script. This function takes important information as attributes and fits the model accordingly. The entire process is executed within the `main_ml.py` file. It begins by creating a game scene and generating data using the `DataProcessing` class. The data is then split, and the shape of the input data is obtained.

Next, we define the `model_builder` function, which establishes the search space for optimal parameters. We utilize the `keras_tuner` library to perform a random search within this defined space, with a maximum of 32 trials. Although this number of searches results in a training time exceeding six , it ensures the discovery of a sufficiently effective model architecture. All iterations of the search process are stored in the `HyperTunning_Trials` directory.

Finally, the best model, identified by the lowest validation loss, is retained. It is retrained using the training set and its performance is evaluated on the testing set. Various metrics, such as Mean Squared Error (MSE), Mean Absolute Error (MAE), and R-squared, are calculated for each output to assess the model's performance. We then proceed to save it using the saving function `save_model_and_scaler` for later use.

As we already did the parameter searching, we have added a version where only the optimal model is set up for training with the wanted parameters hard coded. The search and saving methods have been commented out for the same reasons.

6. Simulation and Evaluation. The next step is to evaluate it in a dynamic investment environment using the `ml.py` script. Here, we simulate 100 games, adjust the portfolio based on the model's predictions, and analyze the outcomes. To facilitate this process three functions are added, `preprocess_state_values`, `adjust_portfolio` and `run_multiple_games`.

The `preprocess_state_values` allows us to only consider the desired features as the input data. It also makes use of the saved scaler used when training the model to ensure accurate predictions. The `adjust_portfolio` function makes investment decisions based on the forecasted prices. It calculates the predicted returns and adjusts the allocation of assets accordingly. Also, to correctly adjust the weights it makes use of the buy and selling functions in `scenes_ai.py` by taking the difference between the current and predicted weights time the portfolio value as input for the `change_amount`. Finally, `run_multiple_games` defines the loop inside of which the games are played and the use of the model. Also, the data had to be reshaped again for the 3D format as well as the rolling_window added.

As a Benchmark to evaluate the performance, we added a `random_games` function that plays the same amount of games making random choices. The final portfolio values over the 100 games are plotted, with the mean of the model version and the mean of the random model. Furthermore, additional plots are added to better understand how precise the predictions are.

6.1. Results. The first important result is the values found for the architecture. The random search revealed that a two-layer configuration, with one layer consisting of 64 nodes and the second layer with 32 nodes, along with a dropout rate of 0.2, yielded the best results. Indeed, it got the lowest value for the validation loss with an MSE of 8.99. When retraining this model, it stopped because of the early stopping after epoch 15. One interesting thing we could observe is the dynamic of the two-loss function (train and validation). Indeed, at the first epoch, the training loss is very high (4828,2661) whereas the validation loss is drastically lower (13,072). But this fastly converges as at the second epoch the training loss is down to 12,90 for a validation loss of 12,99. They then both slowly decrease to values slightly lower than 10, but the validation loss starts increasing slightly (around 1e-2 magnitude). This might be due to how TensorFlow adjust the loss function on the validation and train set. It seems like, during the first epoch, the newly trained setup is not yet applied on the training set but already on the validation set, creating this high disparency. For more detailed information, please refer to Fig 8.1 in the appendix, which provides visualizations of the loss functions during the training process. When then tested on the test set we find:

We can observe that the error term are very high. This is due to the scaling of our output variables. Indeed, by having a closer look at our Data, stock prices vary around 100 and bond prices around 200, meaning that there is approximately 1% in Stock prices and 8% in Bonds prices. There is also a wide difference in their respective R^2 implying that there is more

Metric	Stock Prices	Bond Prices
MSE	1,226	16,767
MAE	0,884	3,266
R^2	0,636	0,254

Figure 6.1. *Performance Metrics for Stock Prices and Bond Prices on Test Set*

predictability in the stock prices. Furthermore, the substantial decrease in error for bond prices between MSE and MAE suggests the presence of outliers that negatively impact the model's performance when predicting bond prices. Here, maybe scaling the output features in some way could have resulted in a better prediction of the model, such as using log prices instead which is a common approach in the field of finance. Now let's discuss the main results of our paper, being the efficiency when it comes to asset allocation in the scope of the trading game. In Figure 6.2 we find very impressive results when investing based on the trained model. Indeed, over the 100 games, 98% of them return a positive total return with a mean final portfolio value of 550349.009. Note that the player initially starts with 500'000. On the contrary, the randomized model finds positive returns in 56,99% of the games, for an average portfolio of 500813.865. So we see that there is a huge gap in the performance between the 2 models suggesting that the model has been able to make good investment decisions despite its rather high error.

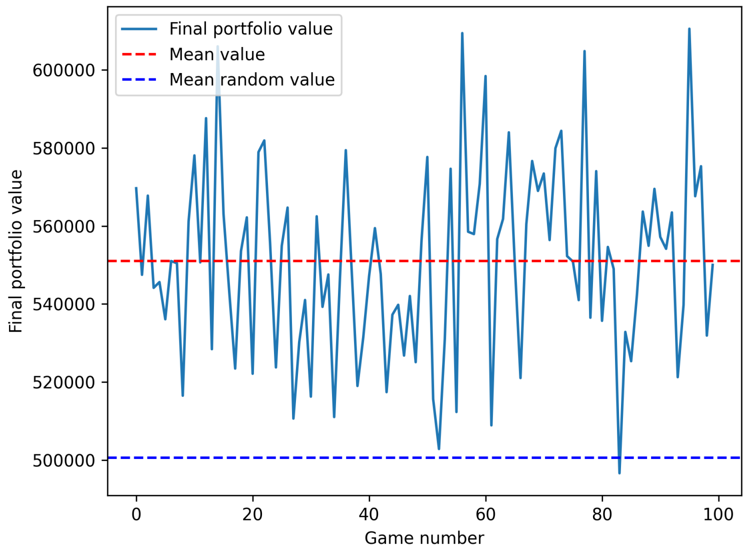

Figure 6.2. *Dynamics of final portfolio value using LSTM Random model*

To better understand this phenomenon, we have decided to further investigate how the predicted prices compare to the actual prices to see if the model maybe does a good job at capturing the global pattern but struggles in determining the exact magnitude of changes which is shown in the MSE and MAE.

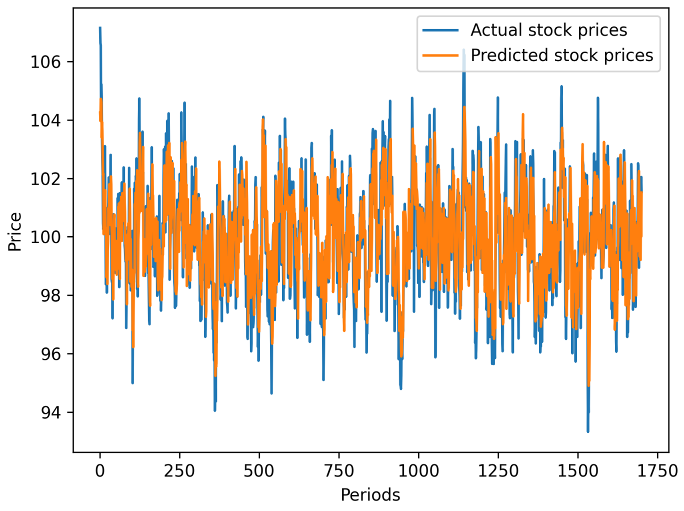

Figure 6.3. *Observed vs Predicted stock prices*

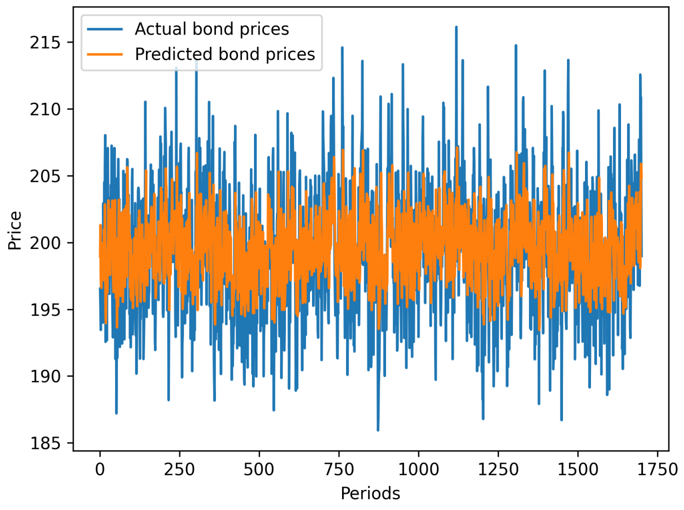

Figure 6.4. *Observed vs Predicted bond prices*

Figures 6.3 and 6.4 support the hypothesis of the pattern being well captured by the model. We particularly observe in Fig 6.3 as the predicted and observed stock prices seem to move together. Also, even when they are outliers such as around period 1500 the model understands it and predicted an outlier as well. But this is not the case for the dynamics of bond prices. Indeed, as seen before, outliers seemed to be a big problem in the predictability of bond prices. We can observe this in Fig 6.4 as there seems to be a rather good understanding of the pattern in the price in terms of whether will it increase or decrease. But there is a clear struggle when it comes to forecasting the magnitude of changes, and even more when there are clear extremes. This might be due to how the bond prices are being generated in the `scenes_ai.py` file. By having a fast look at it we see that the ar coefficient for the ARMA process is lower for bonds (0.5) compared to stock (0.8). Also, they have an equal sigma for a mean that is 2 times higher for bonds. This could result in less autocorrelation as well as higher volatility making the bond series less predictable. More information on the error in the predictions in prices can be found in the Appendix with figures 8.2 and 8.3

7. Conclusion. This paper introduces an AI model utilizing LSTM (Long Short-Term Memory) neural networks to predict optimal prices in a trading game. The main goal of this AI model is to enable precise asset allocation decisions by forecasting future bond and stock prices, utilizing four key variables: Inflation, Fed rate, Bond prices, and Stock prices. The LSTM model is built using TensorFlow and trained on a dataset consisting of 20,000 game simulations.

The paper provides a comprehensive methodology that covers data preparation, model development and training, and model evaluation. The data preparation phase involves creating a comprehensive dataset, which is then divided into training and testing sets. The model development phase includes constructing a deep neural network, with hyperparameter tuning to optimize its architecture. During training, an early stopping mechanism is implemented to prevent overfitting, and the model's performance is assessed using evaluation metrics such as MSE, MAE, and R-squared. To evaluate the model's effectiveness in real-world scenarios, it is applied in simulations of the trading game.

Through this research, the paper successfully demonstrates the potential of LSTM-based AI models in predicting optimal prices and aiding asset allocation decisions within the trading game context. Despite the inherent limitations in accuracy, the LSTM model outperforms random decision-making and provides valuable insights into effective investment strategies. Future studies can focus on enhancing the model's accuracy by exploring additional variables and features while addressing the limitations identified in this research. The results of this study contribute to the field of AI in financial portfolio management and provide a strong foundation for further advancements in this domain.

Finally, there is still a lot of room for improvement. We have already mentioned the possibility of rescaling the output by taking their log values instead. Other than that, it would be interesting to explore the use of different models, such as combining an LSTM with a CNN model to make better use of their respective strengths. Making a reinforcement learning model would also provide an interesting point of view and maybe give more insights into decision-making. Also, using the same model on real financial data would be an important step to see how it would generalize in a more realistic environment. In this case, adding more input variables would be a necessity and one would also need to improve its computational capacities as a regular personnal computer may no more be suited. Nevertheless, this project has shown to provide a great foundation concerning the prediction of financial returns.

REFERENCES

[1] Md. Arif Istiake Sunny, Mirza Mohd Shahriar Maswood, and Abdullah G. Alharbi. Deep learning-based stock price prediction using lstm and bi-directional lstm model. In *2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, pages 87–92, 2020.

[2] Jimmy Ming-Tai Wu, Zhongcui Li, Norbert Herencsar, Bay Vo, and Jerry Chun-wei Lin. A graph-based cnn-lstm stock price prediction algorithm with leading indicators. *Multimedia Systems*, 29:1751 – 1770, 2021.

8. Appendix.

Appendix 1: Note that this provides rather Low information because of the high change in scale. For a better understanding of the evolution of the loss function, you can refer to the code output

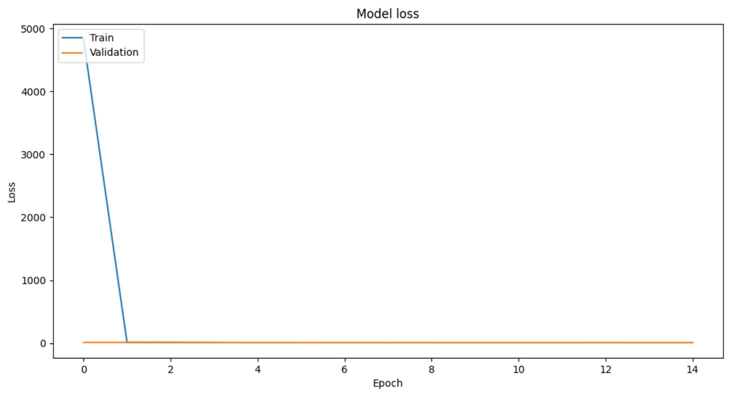

Figure 8.1. *Model Loss vs validation Loss*

Appendix 2: The Absolute difference between observed and predicted Bond prices.

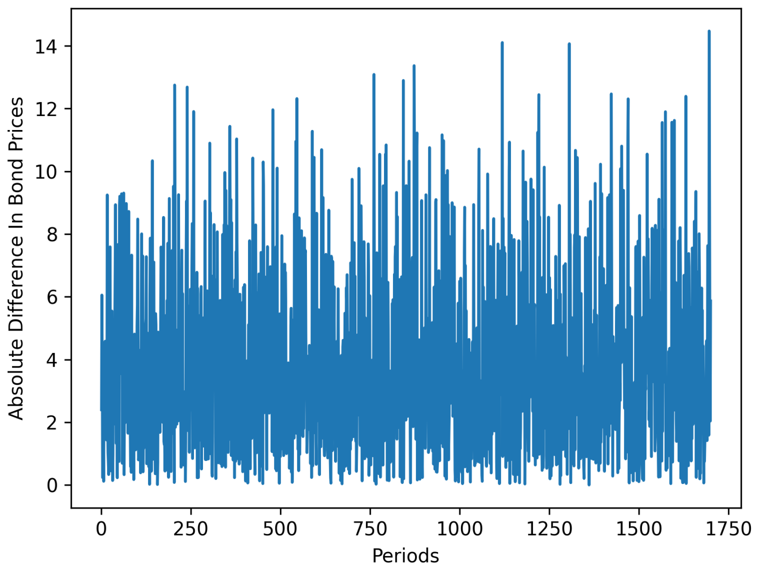

Figure 8.2. *Absolute difference in bond prices*

Appendix 3: The Absolute difference between observed and predicted Stock prices.

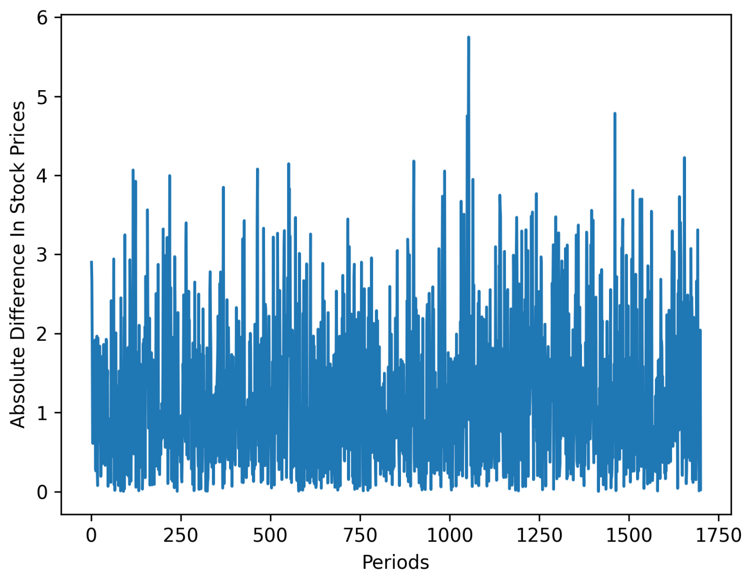

Figure 8.3. *Absolute difference in stock prices*