EE 4463/5573 Final Project Report

by

Gabriel Estevez

Thomas Elie Free

Sergio Romero Jr.

University of Texas at San Antonio

under the guidance of

Dr. Panagiotis Markopoulos

Department of Electrical and Computer Engineering

Fall 2024

Project Denarius: Comparative Analysis of Machine Learning Techniques in Market Forecasting

EE 4463 and EE 5573 – Introduction to Machine Learning

Sergio Romero Jr.
Dept. of Electrical & Computer Eng.
University of Texas at San Antonio
`sergio.romero@my.utsa.edu`

Thomas Elie Free
Dept. of Electrical & Computer Eng.
University of Texas at San Antonio
`thomas.free@my.utsa.edu`

Gabriel Estevez
Dept. of Electrical & Computer Eng.
University of Texas at San Antonio
`gabriel.estevezespinal@my.utsa.edu`

Fall 2024

1 Preamble

- **Course Title**: Introduction to Machine Learning
- **Project Title**:Project Denarius: Comparative Analysis of Machine Learning Techniques in Market Forecasting
- **Student Names**: Gabriel Estevez Espinal, Sergio Romero Jr, Thomas Elie Free
- **Repository Link**: Link

2 Technical Sections

2.1 Introduction

We chose to apply machine learning techniques to the trading of stocks. Stock prediction and trading have been central areas of research and practical application in financial markets for decades. The goal is to leverage historical and real-time data to forecast stock price movements, enabling traders, investors, and institutions to make informed decisions and optimize returns. The increasing availability of financial data and advances in technology have made stock prediction a fertile area for innovation, blending traditional financial theory with cutting-edge computational techniques.

The dataset used for this project is one we found on Kaggle. This dataset is simply a compilation of data that was scrapped from the NASDAQ website by the author. It is a fairly standard dataset that contains stock data from the top 10 performing stocks between July 2013 and July 2023. Each row corresponds to information for a given stock on a given day in that period. The columns of each row contain the stock ticker, date, close price, open price, volume traded that day, and high and low price for that day.

Stock data is widely available, so the contents of this dataset are not very unique. Anyone with the ability to run Python code can download stock information similar to this one. What sets this dataset apart is the fact that it is a compilation of 10 stocks worth of data over 10 years in one place. This relieves the hassle of compiling the data from various stocks into one file.

2.1.1 Linear Regression

Linear regression is a statistical modeling algorithm that attempts to predict a variable of interest using a linear combination of predictor features. It assumes that the relationship between the independent and dependent variables is linear and finds the best-fit linear model. This allows for easy interpretation of results and fast runtimes. Though these traits are attractive, Linear Regression is not always a good fit. Its assumption of linearity is not always a applicable to complicated systems, and can lead to poor performance. It also assumes the relationships do not change over time, which is not a good fit for complex relationships seen in stock market data. For the purposes of this application, it can provide decent short-term prediction of stock prices at a low computational cost. This makes it a good baseline to which we can compare the other methods.

2.1.2 Q-Learning

Q-learning is a model-free reinforcement learning algorithm that aims to learn an optimal decision-making policy by interacting with an environment. It operates by maintaining a table of Q-values, which represent the expected cumulative rewards of state-action pairs. For a given state, the algorithm selects an action (based on an exploration-exploitation tradeoff) and observes the reward received along with the resulting state transition. Using this feedback, the Q-value for the state-action pair is updated iteratively according to the Bellman equation. Sutton and Barto [2018] Over time and through repeated experiences, Q-learning refines its policy to make better decisions.
Because Q-learning relies on experience rather than requiring prior knowledge of the system's dynamics, it is highly adaptable to environments with unpredictable or unknown behavior, such as financial markets. Its flexibility makes it a powerful tool for decision-making in scenarios where the system dynamics are either difficult or impossible to model accurately.
Within the context of our problem; that of modeling financial markets, the decision to implement Q-learning within the context of our problem set is really quite straightforward. The financial markets are notoriously chaotic and unpredictable; however, there is inherent logic to the madness. As stated previously, the stochastic nature of Q-learning enables it to learn strategies based on observable market trends, particularly in the short term. By iteratively updating its policy based on reward signals, it can adapt to transient patterns such as price momentum, mean reversion, or volatility clusters.
However, Q-learning struggles to generalize and create long-term trading strategies due to the delayed nature of rewards in financial markets and the high dimensionality of state and action spaces. For example, the long-term impact of a trade may only materialize days or weeks after the action is taken, making it difficult to assign credit to specific decisions. Additionally, Q-learning's reliance on a Q-table makes it computationally expensive and impractical for large or continuous state spaces, limiting its scalability for modeling complex financial systems. Despite these limitations, its ability to discover short-term strategies in chaotic environments makes it a valuable tool for financial applications, particularly when paired with function approximation techniques like deep learning.

2.1.3 Deep Q-Learning

As discussed in the Q-Learning section, large state spaces present significant challenges for training due to the computational burden and memory requirements associated with storing Q-values for all possible state-action pairs. Deep Q-Learning (DQL) addresses these limitations by employing a neural network to approximate the Q-value table. Rather than explicitly maintaining a table of Q-values, the neural network predicts the expected cumulative reward for a given state, enabling DQL to operate effectively in continuous state spaces. This capability makes DQL particularly well-suited for applications in financial markets, where stock prices and market conditions exhibit dynamic and complex behavior.
To deal with the training complexity and stability issues inherent in reinforcement learning, techniques like stochastic batch training and novel techniques such as Proximal Policy Optimization (PPO) have been introduced. Stochastic batch training improves generalization by randomly sampling mini-batches of past experiences from a replay buffer, breaking the correlation between consecutive experiences. PPO constrains how much the policy can change during training steps, Sutton and Barto [2018] which helps maintain stability and ensures smoother learning—an essential factor in markets where rewards are unpredictable and noisy.
These techniques come with their challenges. Training a neural network for DQL, especially with methods like PPO, demands substantial computational resources and memory for the replay buffer. Hyperparameter tuning—like balancing learning rate and exploration strategies—can feel like a black art, adding another layer of complexity to an already resource-intensive process.
Even so, these methods make DQL far more robust and capable of generalizing to dynamic market conditions. Achieving an optimal reward function paired with sufficient computational resources and sufficient training data, it's possible to achieve a balance between long-term trend modeling and short-term adaptability, making DQL a powerful

tool for navigating the chaos of financial markets.

2.1.4 Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are a specialized type of Recurrent Neural Network (RNN) designed to address the limitations of traditional RNNs in modeling sequential data. A key strength of LSTMs lies in their ability to capture both short-term and long-term dependencies in time-series data, which is achieved through their gating mechanisms. These mechanisms help LSTMs mitigate the vanishing gradient problem, making them particularly effective for identifying complex sequential patterns in stock price datasets.
LSTMs are well-suited for tasks like stock price prediction, where temporal dependencies and dynamic trends play a critical role. Their ability to model intricate relationships between historical prices and predict future movements makes them an appealing choice for this application. However, LSTMs are computationally intensive, requiring significant resources for training and careful tuning of hyperparameters, such as the number of neurons, learning rate, and batch size, to achieve optimal performance.
The chosen architecture for this project consists of an LSTM layer followed by a dense output layer. This configuration is designed to capture temporal patterns in the input data while generating accurate predictions for the target variable. While the computational demands of LSTMs are high, their effectiveness in modeling non-linear temporal relationships justifies their use in this context. For volatile datasets like stock prices, their ability to learn both short-term variations and long-term trends makes them an invaluable tool for forecasting.

2.2 Dataset

As discussed before, this dataset is fairly standard for stock information. Each row of the dataset represents the information for a given stock on a given day. This means that the dataset needs to be split up for training purposes, so we used the company column to split it into separate data frames. From there, the pre-processing differed slightly for each method used.

2.2.1 Linear Regression

For linear regression, we attempted to used the previous ten days to predict the following day. We are only interested in using the close, open, high, and low values for each day to predict those values for the next day, so the remaining columns were dropped from the data frame of each stock. In order to normalize each stock, each series of close/open/high/low were normalized using the first entry in the series, so that they all begin at 1. From here, the data for each ten day period were grouped into an array with each entry in the array being a 4x10 matrix where each column is a specific day and the rows are the close/open/high/low values for that day. Additionally, a separate array was created to hold the same values for the following day.

Once this grouping was done for all stocks, the training and testing data sets were formed by concatenating the arrays for the training stocks into one large matrix along with a separate concatenation of their next-day values. This formed a large matrix and vector where each row in the matrix represents the 10-day history of one of the close/open/high/low values for a given stock, with the corresponding row in the vector being the value on the following day.

A similar set of matrix and vector was created for the Apple stock. The structure exactly the same, where each row is one of the four stock metrics over the course of ten days being used to predict the metric on the following day.

2.2.2 Q-Learning

For Q-learning, the dataset consisted of historical data for 10 stocks: AMD, AMZN, CSCO, META, MSFT, NFLX, QCOM, SBUX, TSLA, and AAPL. The state space was constructed from key features that were deemed essential for making trading decisions. These features included the close, open, high, low, volume, returns (percentage change in price from the previous day), SMA-5 (5-day Simple Moving Average), SMA-20 (20-day Simple Moving Average), volatility (standard deviation of price changes over the past 20 days), and MACD (Moving Average Convergence Divergence indicator). Additionally, the agent's current cash and stock holdings were included as part of the state representation to enable it to consider its position when making decisions.

The features were processed as-is, without normalization, to allow the agent to learn directly from the raw numerical relationships inherent in the data. At each time step, the state was represented as a vector containing these features, along with the cash available for transactions and the number of shares currently held by the agent. This representation ensured that the agent had access to sufficient information about the stock's behavior and its own position when selecting actions.

To prepare the data for training, the state and reward at each time step were computed for each stock. Rewards were defined as the change in the agent's portfolio value resulting from its action, incentivizing the agent to maximize returns while minimizing unnecessary trades. The dataset was segmented into individual episodes, with each stock contributing multiple episodes. The training and testing datasets were split such that nine stocks were used for training, while the AAPL stock was reserved for testing. This setup provided a robust framework for evaluating the agent's performance on unseen data.

2.2.3 Deep Q-Learning

For Deep Q-Learning (DQL), the data preparation process was nearly identical to that of Q-learning. The dataset consisted of historical data for 10 stocks: AMD, AMZN, CSCO, META, MSFT, NFLX, QCOM, SBUX, TSLA, and AAPL. The same state space was used, including features such as close, open, high, low, volume, returns, SMA-5, SMA-20, volatility, MACD, cash holdings, and the number of shares held by the agent. These features were used without normalization, ensuring consistency with the Q-learning setup.

Unlike Q-learning, where episodes were trained independently, Deep Q-Learning required sequential training for all episodes due to the high computational complexity and longer training time of the neural network. The data for all nine training stocks (AMD, AMZN, CSCO, META, MSFT, NFLX, QCOM, SBUX, TSLA) was processed in sequence, and the AAPL stock was reserved for evaluation. This sequential approach allowed the model to leverage all available training data while addressing the increased computational demands of the DQL architecture.

2.2.4 Recurrent Neural Networks (RNNs)

RNNs are designed to capture temporal dependencies in sequential data, such as time-series data from stock prices, the dataset needs to be structured in a way that can feed this sequential nature to the model. To ensure consistency across the different features (e.g., stock prices, trading volumes), the data is normalized. Next, we structured the data so that for each stock, a sequence of 10 consecutive days' worth of values was used to predict the price for the next day. The data was organized as a sequence where each step in the sequence represented one day's worth of stock data (open, close, high, low). This setup allowed the RNN to capture temporal dependencies across consecutive days of stock data.

For the training and testing datasets, we used data from multiple stocks, with each row containing the stock prices for the past 10 days, followed by the next day's prices as the target value. The test dataset, consisting of a different stock or a portion of the data not included in training, was used to evaluate the model's performance after training.

2.3 Methods

2.3.1 Linear Regression

As previously stated, this method works by approximating the dependent variable with a linear combination of the independent variables. For this project, we will be using it to predict the value of the closing value for the next day using values from the previous ten days as predictor variables. Below is shown the basic equation we will be solving.

$$Ax = b \tag{1}$$

Here, A is a matrix whose columns represent data from the previous days. Each column of A, a_n, represents the data from a corresponding number of days before the one being predicted. The rows of which each represent a specific day. The variable x is a column vector that when multiplied by A, produces a linear combination of the columns of A. Shown expanded below:

$$a_1x_1 + a_2x_2 + ...a_nx_n = b \tag{2}$$

The variable b is the independent variable we wish to predict. It contains known data representing the actual closing value for the following day. In order to produce an x vector that gives a best-fit prediction of the closing value, we need to find the pseudo-inverse of the matrix A , $A^{\dagger}$, and apply it to both sides of the equation. To calculate the pseudo-inverse, we decompose A using the singular value decomposition shown below:

$$A = USV^T \tag{3}$$
$$A^{\dagger} = VS^{-1}U^T \tag{4}$$

This allows us to project b onto to span(A), yielding a least-squares solution, y. In other words, this solution is the best-fit estimation of b using the columns of A. The projection can be calculated with the following:

$$y = AA^{\dagger}b \tag{5}$$

Based on the value of the prediction relative to the current day, a trading action can be made that maximizes profit and minimizes risk. This can be used in conjunction with other trading strategies or on its own for simple day trading. For this project, we attempt to do the later for the sake of simplicity.

2.3.2 Q-Learning

Q-learning is a model-free reinforcement learning algorithm that aims to learn the optimal action-value function, denoted as $Q(s, a)$, which estimates the expected cumulative reward of taking an action a in a state s. The Q-learning algorithm relies on the **Bellman Equation** Sutton and Barto [2018], expressed as:

$$Q(s,a) = R(s,a) + \gamma \max_{a'} Q(s', a'), \tag{6}$$

where:

- $Q(s, a)$: Action-value function for state s and action a,
- $R(s, a)$: Immediate reward received for taking action a in state s,
- γ: Discount factor ($\gamma \in [0, 1]$) that determines the importance of future rewards,
- s': The next state after taking action a,
- a': The action taken in the next state.

To iteratively learn $Q(s, a)$, the agent applies the following update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(R(s,a) + \gamma \max_{a'} Q(s', a') - Q(s,a) \right), \tag{7}$$

where α is the learning rate ($\alpha \in (0, 1]$), which controls how much the agent updates the Q-value for a given state-action pair in each iteration.

For this application, the state s is represented by a vector containing:

- Stock data: close, open, high, low, volume, returns, SMA-5, SMA-20, volatility, and MACD,
- The agent's current cash holdings,
- The number of shares held by the agent.

The reward function is designed to incentivize profitable trading actions while penalizing invalid or suboptimal behavior. The function is defined as:

- **Buy** ($a = 0$): If sufficient cash is available, the agent purchases as many shares as possible, and the reward is set to 0 (no immediate reward). If cash is insufficient, a penalty of -100 is assigned.
- **Sell** ($a = 1$): If the agent holds shares, it sells all of them, and the reward is the profit from the transaction. If no shares are held, a penalty of -100 is applied.
- **Hold** ($a = 2$): No transaction occurs, and the reward is set to 0.

The Q-learning agent was trained using historical data for nine stocks: AMD, AMZN, CSCO, META, MSFT, NFLX, QCOM, SBUX, and TSLA. Each stock represented an independent episode, during which the agent interacted with the environment, selected actions, and updated its Q-values based on the Bellman equation. The agent's performance was evaluated on AAPL stock data, which was not included in the training phase, to assess its ability to generalize to unseen market conditions.

2.3.3 Deep Q-Learning

Deep Q-Learning (DQL) extends the Q-learning algorithm by using a neural network to approximate the action-value function $Q(s, a)$ instead of maintaining a Q-value table. This allows DQL to handle larger and continuous state spaces, such as those encountered in financial markets. The neural network learns to estimate $Q(s, a)$ by minimizing the temporal difference loss, which is defined as:

$$L(\theta) = \mathbb{E}\left[\left(R(s,a) + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s,a;\theta)\right)^2\right], \tag{8}$$

where:

- $Q(s, a; \theta)$: Predicted Q-value for state s and action a, given the current network parameters θ,
- $R(s, a)$: Immediate reward received for taking action a in state s,
- γ: Discount factor ($\gamma \in [0, 1]$) that determines the importance of future rewards,
- s': The next state after taking action a,
- θ^-: Parameters of the target network, which are periodically updated from θ.

The neural network is trained to minimize this loss by adjusting θ using stochastic gradient descent. The use of a *target network* (θ^-) helps stabilize training by providing fixed targets during updates, while the main network (θ) is updated iteratively.

For this application, the state s is represented by the same vector used in Q-learning:

- Stock data: close, open, high, low, volume, returns, SMA-5, SMA-20, volatility, and MACD,
- The agent's current cash holdings,
- The number of shares held by the agent.

The reward function is also identical to the one used in Q-learning, incentivizing profitable actions:

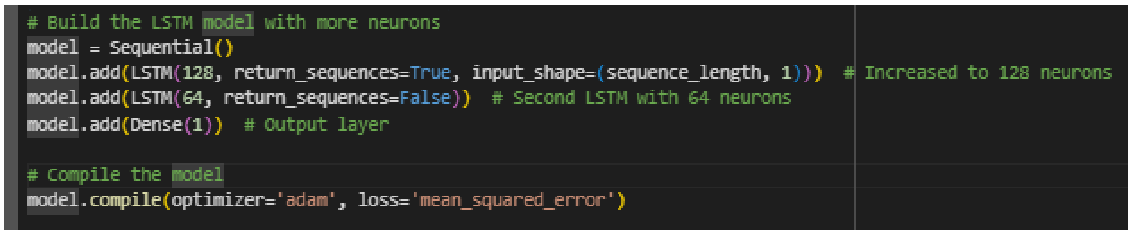
- **Buy** ($a = 0$): If sufficient cash is available, the agent purchases as many shares as possible, with a reward of 0. Insufficient cash results in a penalty of -100.
- **Sell** ($a = 1$): If shares are held, the agent sells all of them, and the reward is equal to the profit. If no shares are held, a penalty of -100 is applied.
- **Hold** ($a = 2$): No transaction occurs, and the reward is set to 0.

The Deep Q-Learning agent was trained using historical data for nine stocks: AMD, AMZN, CSCO, META, MSFT, NFLX, QCOM, SBUX, and TSLA. Training was performed sequentially due to the computational demands of neural network training. A replay buffer was used to store past experiences, which were sampled in mini-batches to break correlations between consecutive states and stabilize learning. The model was evaluated on AAPL stock data, allowing for a direct comparison with the Q-learning agent's performance on unseen market conditions.

2.3.4 Recurrent Neural Networks (RNNs)

When compared to alternatives, the LSTM model demonstrates clear advantages for this application. Classical models like ARIMA work well with stationary time-series data but are less effective for capturing non-linear relationships. Tree-based models like Random Forests can model non-linear patterns but fail to consider the sequential nature of data without significant feature engineering. GRUs, while computationally lighter than LSTMs, may not capture complex temporal dependencies as effectively. The LSTM's ability to model both short-term variations and long-term trends justifies its selection for this task.

Several modifications were made to optimize the model's performance. The number of neurons in the LSTM layers was increased to improve the model's ability to learn from the data. While this enhancement adds computational complexity, it allows the model to capture more intricate patterns, which is crucial for volatile datasets like stock prices. A validation split of 10% was introduced during training to monitor model performance and mitigate overfitting, ensuring that the model generalizes well to unseen data. Finally, min-max scaling was chosen over alternative normalization methods, such as z-score scaling, to avoid assumptions about data distribution that may not hold for financial data.



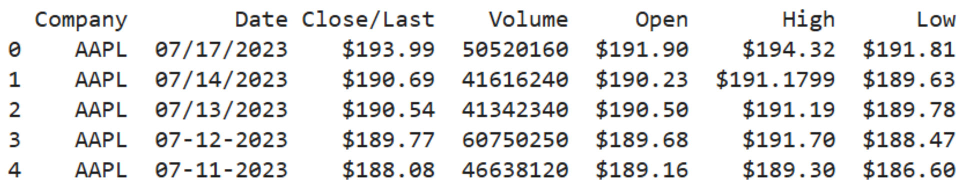
```
# Build the LSTM model with more neurons
model = Sequential()
model.add(LSTM(128, return_sequences=True, input_shape=(sequence_length, 1)))  # Increased to 128 neurons
model.add(LSTM(64, return_sequences=False))  # Second LSTM with 64 neurons
model.add(Dense(1))  # Output layer

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')
```

Figure 1: LSTM model

2.4 Experimental Studies

2.4.1 Linear Regression

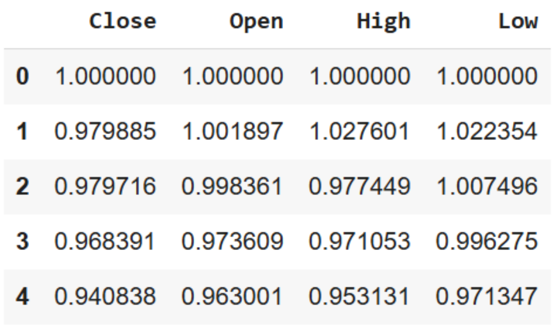
The data was imported from the data.csv file. A small example is show in 2 From there it was split up by company. Only the close, open, high, and low for each stock was kept for every day.



```
  Company        Date Close/Last    Volume     Open       High      Low
0    AAPL  07/17/2023    $193.99  50520160  $191.90    $194.32  $191.81
1    AAPL  07/14/2023    $190.69  41616240  $190.23  $191.1799  $189.63
2    AAPL  07/13/2023    $190.54  41342340  $190.50    $191.19  $189.78
3    AAPL  07-12-2023    $189.77  60750250  $189.68    $191.70  $188.47
4    AAPL  07-11-2023    $188.08  46638120  $189.16    $189.30  $186.60
```

Figure 2: First 5 rows of data from the original data frame

Each company dataset was normalized using the close/open/high/low values on day one.



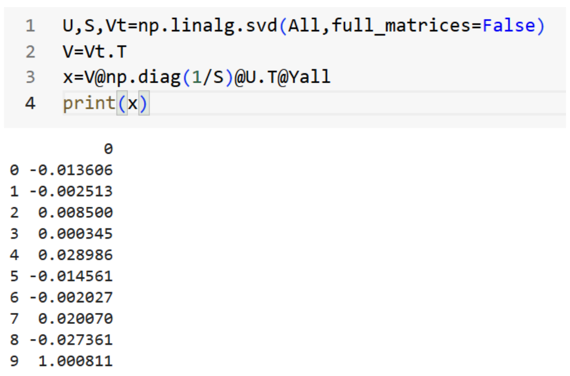
	Close	Open	High	Low
0	1.000000	1.000000	1.000000	1.000000
1	0.979885	1.001897	1.027601	1.022354
2	0.979716	0.998361	0.977449	1.007496
3	0.968391	0.973609	0.971053	0.996275
4	0.940838	0.963001	0.953131	0.971347

Figure 3: First 5 rows of data from the normalized AMD data frame

From there the data was segmented into 10 day chunks along with the data from the following day. The training data consisted of a concatenation all of the chunks from the non-Apple stocks, while the test was solely the Apple stock chunks. The matrix, A, that contained the training data ended up being of size (90180,10), and the next-day vector was size (90180,1). The testing data(Apple only) was (10020,10) and (10020,1).

In order to create a model, we use singular value decomposition to find the pseudo-inverse of the training data matrix and apply it to the next-day training data. This is shown below in 4.

Once we have our x-vector, we can plot the least-squares fit against the next-day training data. This is shown in 5 A root-mean-square-error(RMSE) can also be calculated to show how well it predicts the training data. This was found to be 0.0207 for the training data.



```
U,S,Vt=np.linalg.svd(All,full_matrices=False)
V=Vt.T
x=V@np.diag(1/S)@U.T@Yall
print(x)
```

```
          0
0 -0.013606
1 -0.002513
2  0.008500
3  0.000345
4  0.028986
5 -0.014561
6 -0.002027
7  0.020070
8 -0.027361
9  1.000811
```

Figure 4: First 5 rows of data from the normalized AMD data frame

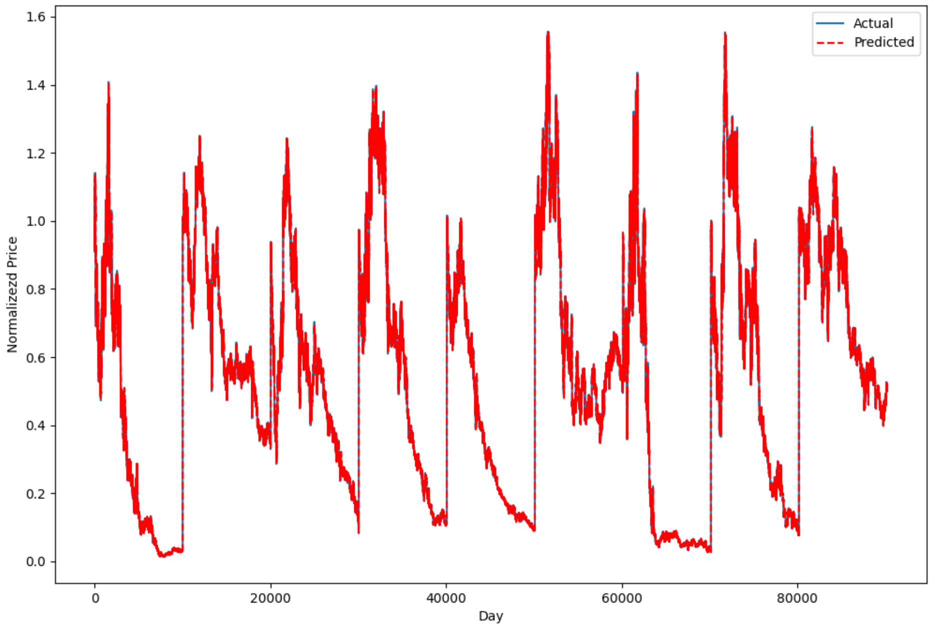


Figure 5: Training data and best fit linear model

This yielded a very good fit for the training data, so the next step is to apply this model to the Apple data. The same plot was produced to show the prediction compared to the real data, and is shown in 6

The RMSE for testing was found to be 0.0116. This is slightly better than the RMSE from the test data. This slight performance boost may be attributed to the slight variance in the training dataset due to one of the stocks performing a little better over time and strayed a little farther from the value at which it was normalized. This slight change seems to have skewed the training model, but since the Apple stock performed roughly the same as the majority of the stocks, it is still modeled fairly well.

In order to test this against other models, it was given $1000 and iterated through each day of the Apple stock. If it predicted that the following day was going to be greater in value than the current day, it would invest all of its money into the stock. At the next day, it would then sell the stock at close, and its money was updated to reflect the actual change in price.

This strategy netted the algorithm a profit of roughly $ 981. The money it held over time is shown in 7 along with the next-day stock value. This allows us to see that even though it is not 100% accurate, it is still profiting in the long term.

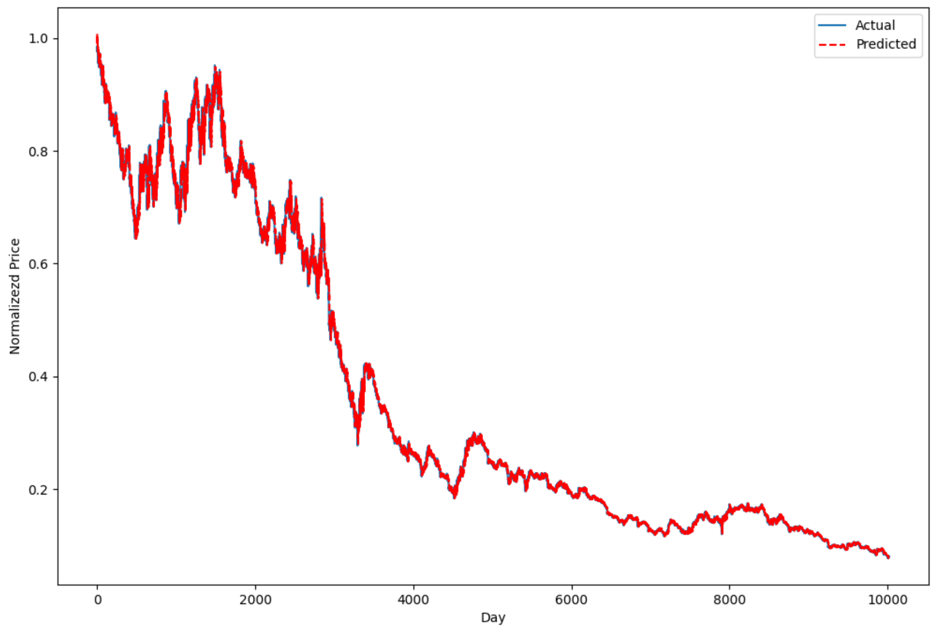


Figure 6: Testing data and best fit linear model

2.4.2 Q-Learning

Table 1: AAPL Stock Data Sample (Rounded to Two Decimal Places)

Close/Last	Open	High	Low	EMA-12	EMA-26	MACD	Returns	SMA-20	Volatility
17.80	17.78	18.01	17.62	16.70	16.25	0.45	0.02	16.19	0.03
17.78	17.73	17.94	17.47	16.86	16.36	0.50	-0.00	16.31	0.02
17.94	17.86	17.96	17.82	17.03	16.48	0.55	0.01	16.44	0.02
18.13	18.01	18.35	18.00	17.20	16.60	0.60	0.01	16.59	0.02
17.90	18.20	18.23	17.89	17.31	16.69	0.61	-0.01	16.74	0.01

The performance of the Q-learning agent was evaluated on the AAPL stock using 500 independent trials. The outcomes are summarized below:

- **Average Profit (Q-Learning)**: $1050.33
- **Variance**: 139709.35
- **Standard Deviation**: $373.78
- **Buy-and-Hold Profit**: $9896.11

The results indicate that while the Q-learning agent was able to learn and exploit market patterns during training, it performed poorly when compared to the simple buy-and-hold strategy. The large variance and standard deviation highlight the inconsistency of the Q-learning agent's performance across trials.

Although the Q-learning agent was able to recognize patterns in specific stocks like AMD and AMZN during training, its ability to generalize to unseen data (AAPL) was limited. This suggests that the patterns it learned are insufficiently robust or that the relationships between features and actions are more complex than the model can currently handle.

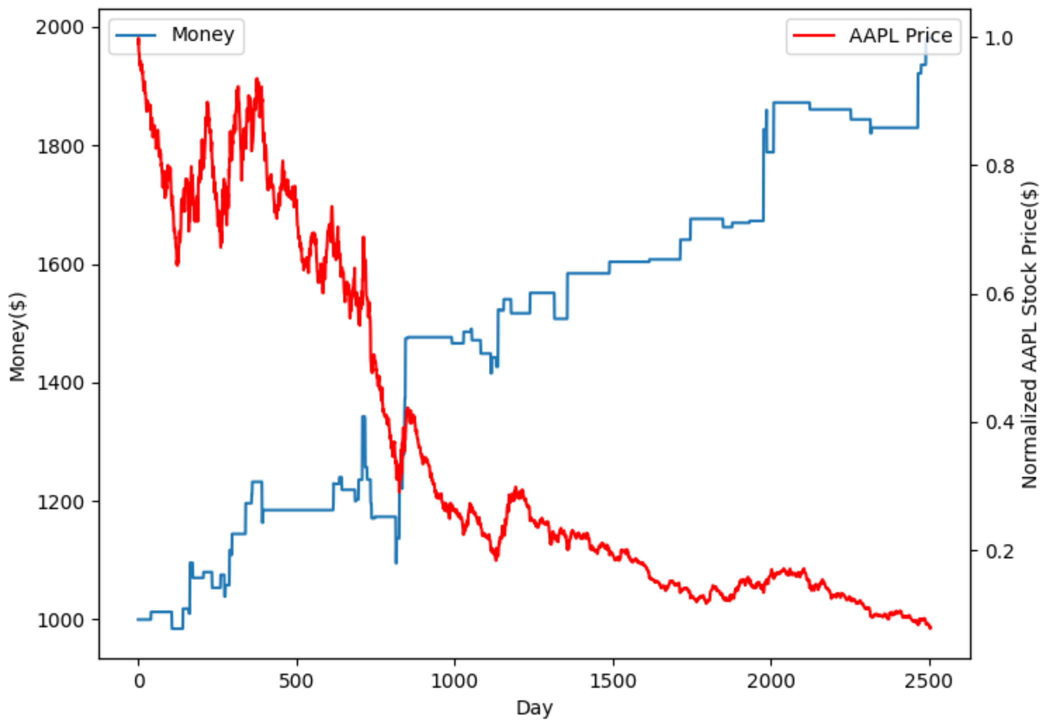
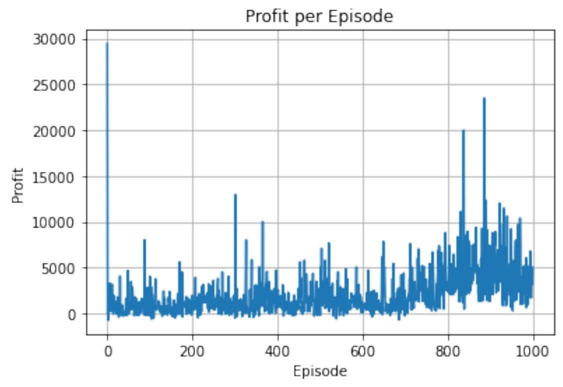


Figure 7: Testing data and best fit linear model



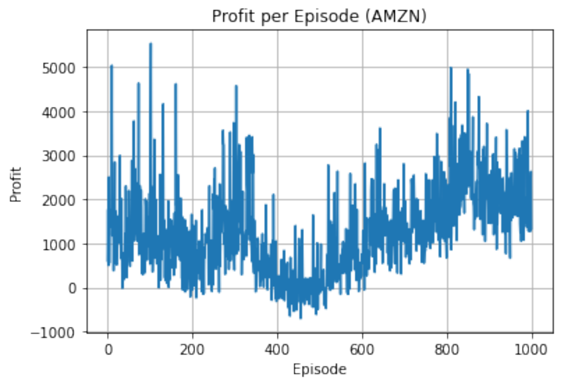
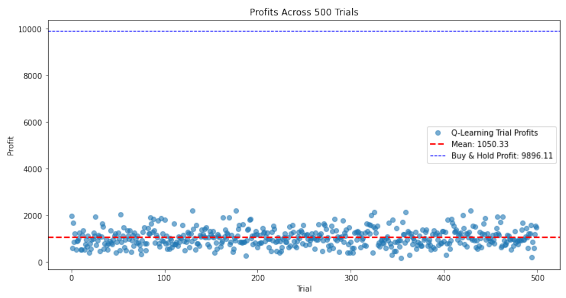


Figure 8: Performance of the Q-learning agent on AMD (left) and AMZN (right).

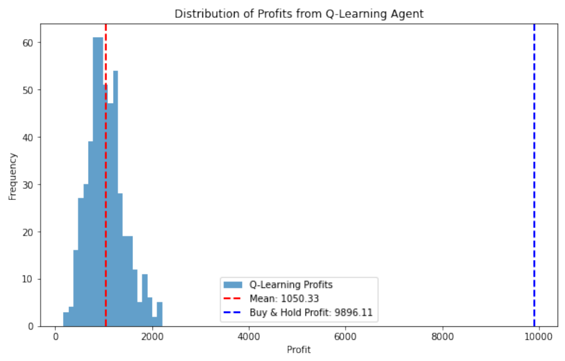
2.4.3 Deep Q-Learning

The performance of the Deep Q-Learning (DQL) agent was evaluated on the AAPL stock over 9 episodes. Due to the challenges encountered during training, the agent was unable to develop a meaningful policy, resulting in the following outcomes:

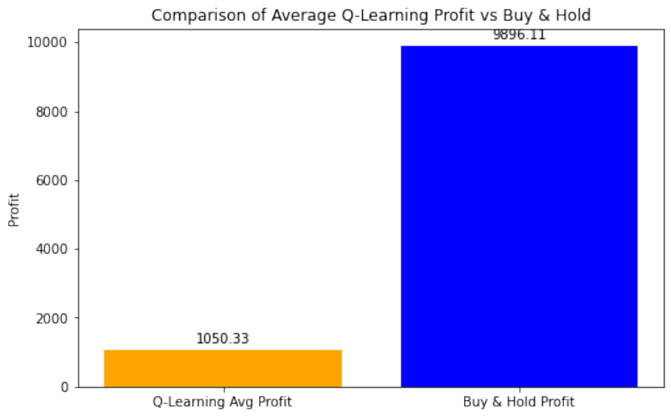
- **Average Profit (DQL)**: \$0.00
- **Variance**: \$0.00
- **Standard Deviation**: \$0.00
- **Buy-and-Hold Profit**: \$9896.11



(a) Comparison of Q-learning and buy-and-hold performance on AAPL stock.



(b) Frequency distribution of profits across 500 trials for Q-learning on AAPL.



(c) Comparison of average profits from Q-learning and buy-and-hold strategies on AAPL stock.

Figure 9: Combined visualization of Q-learning and buy-and-hold strategy performance and profit distribution on AAPL stock.

The results indicate that the DQL agent failed to generate profits, with its performance significantly lagging behind the simple buy-and-hold strategy. The flat profit curve across all episodes, shown in Figure 10, reflects the agent's inability to learn from the training data due to repeated system crashes and incomplete training.

These outcomes highlight the limitations of the experimental setup. Despite attempts to reduce network complexity, batch size, and memory size, the computational demands of DQL exceeded the available system resources. This resulted in a failure to train the neural network effectively, rendering the agent incapable of making meaningful trading decisions.

2.4.4 Recurrent Neural Networks (RNNs)

To evaluate the predictive capability of RNNs, a dataset consisting of stock market information was used. The dataset included sequential features that enabled the model to learn trends and temporal dependencies in stock prices. The dataset was divided into an 80/20 split, with the first 80% used for training and the remaining 20% for testing. A further 10% of the training data was used for validation during training to monitor overfitting. Performance was evaluated using the following metrics: Mean Squared Error (MSE) ,Root Mean Squared Error (RMSE) , Mean Absolute Percentage Error (MAPE)

$$h_t = f(W_h h_{t-1} + W\_x x\_t + b\_h), , \tag{9}$$

where:

- h_t: Hidden state at time t, representing the memory of the network,
- W_h: Weight matrix for the hidden state,
- h_{t-1}: Hidden state from the previous time step,

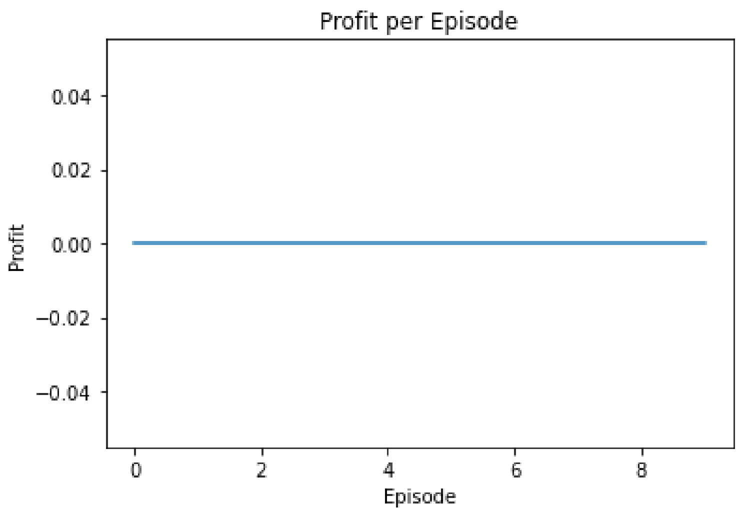


Figure 10: Profit trajectory of the Deep Q-Learning agent over 9 episodes on AAPL stock.

- W_x: Weight matrix for the current input,
- x_t : Input vector at time t
- b_h : Bias term,
- f: Activation function, commonly $tanh$

The output y_t at each time step t is computed using the hidden state:

$$y_t = g(W_y h_t + b_y), \tag{10}$$

where:

- y_t: Output at time
- W_y: Weight matrix for the output layer,
- b_y: Bias term for the output layer,
- g: Output activation function, such as softmax (for classification) or linear (for regression).

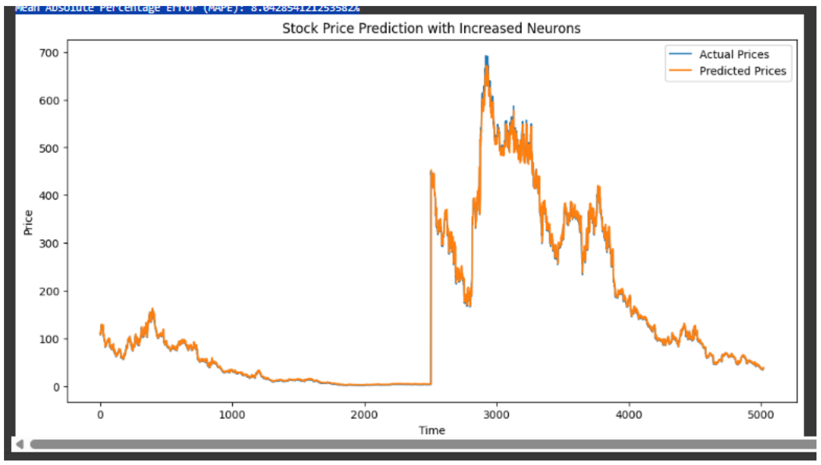
Data Preparation: The dataset included the following features:

- **Open**: The opening price of the stock.
- **High**: The highest price of the stock during the trading day.
- **Low**: The lowest price of the stock during the trading day.
- **Close**: The closing price of the stock (target variable).
- **Volume**: The trading volume for the stock.

The data was normalized using Min-Max scaling to ensure all features were in the range [0, 1], allowing the model to converge efficiently during training.

The RNN was tasked with two outputs:

- **Price Prediction**
- **Buy/Sell/Hold Action**
-
  - **Buy**: If the predicted price increase exceeds a threshold.
  - **Sell**: If the predicted price decrease exceeds a threshold.
  - **Hold**: If the price change is within the threshold.}



```
Mean Absolute Percentage Error (MAPE): 8.042854121253582%
```

Figure 11: Enter Caption

2.5 Result Analysis and Discussion

2.5.1 Linear Regression

On its own, this model produces a decent profit. Over 10 years of data, it nearly doubled. This is comparable to investing a long term bond at roughly a 7% annual interest rate over the same duration. It should be noted that this model was using a day trading approach, but we did not implement a trading fee as one would expect. This profit does not reflect any of these fees, nor any for subscriptions used to access live data. During the training and testings phases, it produced an RMSE of 0.0207 and 0.0116 respectively. When scaled back up using the initial values for Apple's closing price, the are roughly $4, and $2.25 respectively.

Even though it did perform fairly well on the Apple data, more testing should be needed in order to reinforce its validation.

2.5.2 Recurrent Neural Networks (RNNs)

The results from the LSTM model, as outlined in the experimental studies, demonstrated strong predictive capabilities with a Mean Squared Error (MSE) of 0.000021, a Root Mean Squared Error (RMSE) of 0.00458 USD, and a Mean Absolute Percentage Error (MAPE) of 1.23%. These metrics highlight the model's ability to produce highly accurate predictions, with errors that are minimal relative to actual stock prices. The low MAPE, in particular, signifies that the model can generalize effectively across different scales of stock prices, a critical requirement for real-world applications where prices may vary significantly. The chosen architecture—a two-layer LSTM with 128 and 64 neurons, respectively—balanced complexity and performance effectively. Initial experiments with a single LSTM layer of 64 neurons yielded higher errors, indicating that deeper architectures are better suited for capturing long-term dependencies. Conversely, attempts to further increase the number of neurons or add additional layers resulted in overfitting, as evidenced by an increase in validation loss.

In conclusion, the results underscore the potential of LSTMs for stock price forecasting but also highlight opportunities for improvement through the integration of additional context and exploration of advanced architectures. This study serves as a foundation for further investigations into more comprehensive and adaptive financial models.

2.5.3 Q-Learning

The evaluation of the Q-learning agent on AAPL stock highlighted its limitations in financial market applications. While the agent achieved an average profit of $1050.33 across 500 trials, this was significantly lower than the buy-and-hold strategy's $9896.11. The high variance (139709.35) and standard deviation ($373.78) in the agent's performance underscore its inconsistency, suggesting that it struggled to generalize patterns learned during training to unseen data.

The agent's better performance on stocks like AMD and AMZN (Figure 8) compared to AAPL highlights its reliance on specific market conditions during training. However, its inability to match the robustness of the buy-and-hold strategy on unseen data points to the inherent complexity of modeling financial markets using Q-learning. Figure 9b illustrates the variability in profits, further emphasizing the model's instability and limited generalization capability.

These results indicate that stock market patterns are too intricate for Q-learning's tabular approach, requiring more advanced methods. Future work could explore techniques such as deep Q-learning or policy gradient methods, which are better equipped to model the non-linear and dynamic relationships in financial data. Enhancements such as improved feature engineering, ensemble methods, or refined reward functions could also help address the variability and improve the model's ability to generalize across diverse market conditions.

2.5.4 Deep Q-Learning

Attempts to train the Deep Q-Learning (DQL) agent were met with significant computational challenges, which limited the ability to conduct a comprehensive evaluation. Despite efforts to reduce the complexity of the neural network architecture, such as decreasing the batch size and memory size, the training process repeatedly caused system crashes. This was likely due to the high resource demands of the DQL framework, particularly given the large state space and sequential training process required for financial market data.

As a result, the DQL agent achieved a profit of $0 over 9 episodes, reflecting its inability to complete meaningful training. The absence of learned policies highlights the difficulties in applying DQL without adequate computational resources. These challenges emphasize the need for access to high-performance hardware or cloud-based solutions to train deep reinforcement learning models effectively.

Future work could explore additional optimizations, such as:

- Employing smaller neural networks or alternative architectures, such as recurrent layers with fewer parameters.
- Reducing the state space by selecting a minimal set of high-impact features.
- Leveraging cloud-based or distributed computing resources to handle the computational load.

Despite the setbacks, these findings underscore the potential value of exploring alternative reinforcement learning techniques, such as policy gradient methods or simplified DQL implementations, that may balance model complexity with computational feasibility. Further work is needed to assess whether these approaches can effectively capture the complex patterns inherent in financial market data.

3 Educational Sections

3.1 From Proposal to Final Report

In our proposal we had planned to implement these trading algorithms for an entire portfolio of stocks, but we ended up opting to use 9 of the stocks for training and using the Apple stock for testing our algorithms. This was definitely due to us getting ahead of ourselves. When we were designing our algorithms, it proved to be too complicated for us to implement in the time we had remaining.

3.2 Learning Outcomes and Skill Development

The methods were distributed evenly amongst the three members. Gabriel was responsible for the Recurrent Neural Network method, Elie was responsible for the Linear Regression and Deep Q-Learning model, and Sergio was given the Q-Learning model. This was swapped around a bit towards the end because Elie was having trouble getting the DQL model to train at all, so Sergio took it over as he was finished with the DQL model. This is discussed later in the Challenges and Problem-Solving section.

The key concepts developed by this project were mainly in the implementation of neural networks, specifically in Python code. We all had prior coding experience, but that did not cover using the Tensorflow library for the most part. A particularly challenging implementation was seen in the DQL method, and the training cycle is notably different from Q-Learning and normal training of neural networks.

3.3 Challenges and Problem-Solving

Issues presented themselves throughout the implementation of the DQL method. At first, Elie could not get the model to train on data in the replay memory. This continued until Sergio took over after he finished the QL method. He got it to the point where it could train, issues began to arise as once I was able to train the agent using the DQL methodology, the problem although not clear at the moment was the shear computational complexity of neural networks. An example I can give is from the very simple fact that my attempt to train the agent over night failed miserably, twice. First the model trained for 12+ hours however, after training, there was a bug in the code which rendered the training useless, second time, the kernel crashed leaving a failed training session. I attempted many modifications to address the issue but due to a lack of technical capacity, my efforts were in vein. Future work will consist in rectifying such shortcomings.

3.4 Reflection and Self-Assessment

3.4.1 Elie

I feel like we did not appropriately adhere to our work schedule, so we rushed most of the work in the last two weeks. Not allotting appropriate time to each section led to lower performance in most of the models. For the Deep Q-Learning network specifically, computation time for training was astronomical on our systems, so we were not able to train it to make anywhere near decent predictions.

If I were to do this over, I would definitely prefer to not use neural networks if we have to implement multiple methods. This would have allowed us to more appropriately allot time to each method. In the case that we could concentrate on a single method, the neural networks would definitely be more appropriate since we could dedicate all the time to it instead of balancing it with other methods.

If revisited, I think that an actual implementation of the portfolio management would be interesting to look into, although maybe not for so many methods at once. Getting it to work on something as simple as the linear prediction model seems daunting, let alone on a neural network. Something that would also be interesting to research is the implementation of using the data of multiple stocks at once to predict a single stock. This may lead to better prediction at the cost of more intense training. Estimation using higher order polynomials or other basis functions could also be an interesting addition for better estimation.

3.4.2 Sergio

I had a fun time with this project, it was sufficiently ambitions as to merit my full attention with just the sufficient amount of reward sprinkled throughout to keep me on my feet. I particularly had a fun time understanding, deriving, and implementing the basics of Reinforcement Learning and I look forward to further explore the field to a fuller and more in depth capacity. I believe that this project reinforced and developed me many of the core principle in product development and engineering. More specifically it the development and deployment of Machine Learning algorithms, which I seek to further explore within the context of Robotics and Hardware Accelerators. Although I was not successful in assisting my team in the successful development and deployment of a Deep Q-Learning Algorithm, this serves as an instrumental data point for further intellectual and technical development.

Revisiting the project from a wiser and sober-eyed perspective, it would be wise to seek resources and advice on the feasibility of our project proposal. In retrospect, although this was a fascinating experience and will serve as a foundational stepping point for our technical development, I am disappointed in the simple fact that our final product is not perfect. As I am a perfectionist at heart and with a propensity for attention to detail, I will work hard to overcome such trivial technical challenges.

3.4.3 Gabriel

Personally I felt like the project was challenging enough to keep us all engaged and actively researching and learning. As I did not have a lot experience with neural networks it was fascinating to be able to set a goal and apply different methods with testing each one and seeing how each model can be vary from efficiency. I really enjoyed diving deep into how neural networks functions and how they are deployed. Although we struggled with time management and did allocate enough time for training and research of every topic, it was a great learning experience and exposure to machine learning. I have a strong interest in the finance industry and the scope of this project help me see how machine learning is going to play a huge role in the future of the economy.

If more time and resources where available I believe my team would have been able to correctly implement all the methods and have been able to provide a sufficiently working model that could possibly be deployed in the markets. If the project where revisted I would like to reach out to professionals that have more experience and could helps us with the scope of the project. However overall we learned a lot and this is a stepping stone for the foundations of our future.

3.5 Future Directions and Continued Learning

We definitely need more practice in time-management and skills. Had we been able to figure out the issues with training the DQL, we would have gotten better results from that method. I also feel that we need to work on our skills for implementing/training neural networks in general. The documentation and examples of people implementing DQL for stock trading environments is very scarce, so we had to implement this on our own. This caused several issues with runtime due to lack of optimization.

This project was definitely a lesson in the importance of time management and determining reasonable expectations for project goals. As some of us are in/going to be in graduate school, these are definitely skills that need to be perfected, so as not to burn out.

References

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018. ISBN 978-0262039246.