Complexity Analysis of Reinforcement Learning Models Applied to Stock Trading

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in

Computer Science

By:

Jason Davis, Erich Schwarzrock, and Hezekiah Owuor

Project Advisors:

Marcel Blais, Randy Paffenroth, Stephan Sturm, and Joerg Osterrieder

Sponsored By:

Zurich University of Applied Sciences

Date: March 28, 2022

Abstract

In this report, we analyze the effect of data complexity on the performance of financial reinforcement learning models. We created six models which were identical except for the complexity of their learning data. The goal for each was to make as much money as possible by investing in only a single stock. We trained these models on daily index fund data, intraday index fund data, and daily foreign exchange data. We then analyzed the effect that the different data complexities had on both the training and testing returns. Simpler models cannot learn anything and will perform poorly, while if a model is too complex, the agents will overfit the training data and perform poorly on testing data. State spaces with moderate complexity tend to perform the best.

Acknowledgements

We would firstly like to thank our project advisors, Professors Randy Paffenroth, Marcel Blais, and Stephan Sturm for helping us start this project by providing excellent background information regarding both finance and reinforcement learning, as well as providing feedback and guidance throughout the project. We would also like to thank Dr. Jörg Osterrieder of the Zurich University of Applied Sciences (ZHAW) for providing us with the project and helping to guide our research and methodology. Special thanks go to Piotr Kamil Kotlarz and Fernando De Meer Pardo for providing important background information regarding the finance aspect of the project and helping with the analysis of our project as a whole. Finally, we would like to extend thanks to ZHAW itself for funding our access to Google Colab to facilitate training our models.

Contents

1 Introduction **6**

2 Background and Literature Review **7**
2.1 Finance . . . 7
2.2 Reinforcement Learning . . . 9
2.2.1 Q-Learning . . . 9
2.2.2 Deep Neural Networks . . . 10
2.2.3 Fully-connected Neural Networks . . . 10
2.2.4 Deep Q-Networks . . . 11
2.2.5 Improvements to the Deep Q-Network . . . 12
2.2.6 Double Deep Q-Network . . . 12
2.3 Related Works . . . 13

3 Methodology **14**
3.1 Data Sources . . . 15
3.2 Model Design . . . 16
3.3 Neural Network Architecture . . . 18
3.4 Training Process . . . 19
3.5 Training and Testing Split . . . 20
3.6 Improving the Original Model . . . 20
3.7 Software Development Environment . . . 22
3.8 Project Management Software . . . 22
3.9 Programming Integrated Development Environments (IDE) . . . 22

4 Results **23**
4.1 Index Funds . . . 25
4.2 Forex . . . 29
4.3 Intraday . . . 33

5 Conclusion **34**
5.1 Future Work . . . 35

A Software Requirements **38**

List of Figures

1 An example of a deep feedforward neural network consisting of and input layer with two neurons, two hidden layers with five and four neurons respectively and an output layer with one neuron. 11
2 Graph displaying the percentage frequencies of the actions across epochs for index funds model 2. 25
3 Graph displaying the percentage frequencies of the actions across epochs for index funds model 3. 26
4 Graph displaying the percentage frequencies of the actions across epochs for index funds model 5. 26
5 Graph displaying the index fund training returns for models 1 - 3 for each epoch, with a moving average of the past 100 epochs, the NAV was reset to 1 after each epoch. 27
6 Graph displaying the index fund testing returns for models 1 - 3 for every 10 epochs, with a moving average of the past 10 tests, the NAV was reset to 1 after each epoch. 27
7 Graph displaying the index fund training returns for models 4 - 6 for each epoch, with a moving average of the past 100 epochs, the NAV was reset to 1 after each epoch. 28
8 Graph displaying the index fund testing returns for models 4 - 6 for every 10 epochs, with a moving average of the past 10 tests, the NAV was reset to 1 after each epoch. 28
9 The forex training returns for each epoch, with a moving average of the past 100 epochs, the NAV was reset to 1 after each epoch. 29
10 The forex testing returns for every 10 epochs, with a moving average of the past 10 tests, the NAV was reset to 1 after each epoch. 29
11 The frequencies of the actions across epochs for forex model 1. 30
12 The frequencies of the actions across epochs for forex model 2. 31
13 The frequencies of the actions across epochs for forex model 3. 32
14 The intraday training returns for each epoch, with a moving average of the past 100 epochs, the NAV was reset to 1 after each epoch. 33

15 The intraday testing returns for every 10 epochs, with a moving average of the past 10 tests, the NAV was reset to 1 after each epoch. 33
16 The frequencies of the actions across epochs for intraday model 1. 34
17 The frequencies of the actions across epochs for intraday model 2. 35
18 The frequencies of the actions across epochs for intraday model 3. 36

List of Tables

1 State spaces for each equity index model. 16
2 State spaces for each forex model. 17
3 Neural network architecture 18

1 Introduction

With increased access to large amounts of data comes significant advancements in areas like machine learning and artificial intelligence (AI), which have enabled us to enhance our lives and tackle complex problems. The financial market is a prime example of a field where researchers are employing mathematical and computational techniques to aid in the decision-making process, the reason being that financial markets are very dynamic and ever fluctuating. Thus, the financial market presents unique challenges to consider, such as trading costs, current market conditions, historic actions, and accurate financial environment representations, to mention a few. Before the development of AI, it was the job of investors and traders to use this information to make optimal decisions that maximize expected return and reduce risk within the context of a trading system. However, due to market complexities, it can be challenging for agents to consider all the relevant information to take an informed position. This is where reinforcement learning (RL), an area of machine learning, comes into play. Through repeated interaction with a market environment, an RL agent can learn optimal trading strategies by taking certain actions, receiving rewards based on these, and adapting future actions based on previous experience.

Reinforcement learning has a rich history of use in a variety of fields. The Arcade Learning Environment, developed in 2012 by Bellemare et al., provides an interface to a collection of hundreds of video games for the Atari 2600 game console [1]. The platform was designed to evaluate the development of domain-independent agents trained through traditional RL techniques by measuring their average scores over 30 trials. Even in the realm of finance, RL is not a novel concept. In the 1990s, Moody and Saffell experimented with real-time recurrent learning in order to demonstrate a predictable structure to U.S. stock prices [2]. They claimed that their agent was able to make a 4000% profit over the simulated period of 1970 to 1994, far outperforming the S&P 500 stock index during the same timespan.

However, previous studies on applying reinforcement learning in finance have provided insufficient analysis of their chosen model compared to similar ones. For instance, Wu et al. constructed their own technical indicators to add to their reinforcement model [3]. They did not test their model against simpler models, they only tested it against the turtle trading strategy, which works by buying a stock once it goes above the high of the past n days, and then selling it once it drops below the low of the past m days, where

$n > m$ [4]. Testing against this is problematic, as one must consider the well-studied phenomenon known as the "curse of dimensionality." Simply put, as one embeds a data set into a higher dimensional space, while keeping a fixed number of data points, the density of the data points gets smaller and thus it becomes harder to prevent models from overfitting. Somewhat paradoxically, this could lead to more complex models performing worse than simpler ones. Thus, it is important to test the model on multiple dimensionalities of data to make sure the data is not too complex that it overfits, or too simple that it can't learn enough.

Since these papers do not provide an in-depth analysis, our objective for this project is to assess how altering the complexity of data available to a trading agent affects its overall performance relative to the market. To do this, we adopt a double deep Q-network (DDQN) [5] algorithm to trade in three environments, each focusing on one of daily equity index, intraday equity index, and daily foreign exchange trading. Each market environment contains multiple state spaces with varying amounts of data and asset dimensionality, such as 1-day returns, 5-day returns, and currencies. Using these data the agent is able to take three potential actions; buy, flat, and sell short. The algorithm is trained and tested using real market data over the past two decades.

Following the Introduction, our paper is organized as follows: In Section 2, we discuss the financial and RL foundation to our work. Section 3 details the methodology and decision-making process for testing our model's performance, as well as a break-down of tools used. In Sections 4 and 5, we examine our results, make conclusions about the process, and provide suggestions for future work on this topic.

2 Background and Literature Review

2.1 Finance

The first market that we investigate are equity indices. Equity indices are a collection of stocks which, when combined, are "statistical indicators of changes in the market value of a certain group of shares or stocks" [6]. In simpler terms, equity indices help investors gauge the overall performance of the stock market. The indices are typically composed of stocks such as Microsoft, Exxon Mobil, or Boeing, which meet a certain performance criteria. Examples of indices include the Standard and Poor's 500 (S&P 500), the Nasdaq Composite, and the Dow Jones Industrial Average (DJIA or simply, "the Dow") [7]. However, equity indices are exactly just that: indices. They only serve to measure, and cannot be bought directly unless one buys each individual constituent stock. Rather than doing this, which may be costly (due to individual transaction costs), time-consuming, and difficult to manage, investors will buy either mutual funds or exchange traded funds (ETFs) that replicate the returns of an index [8]. Due to their transparency, lower trading cost, and overall similarity to regular stocks, we have elected to use ETFs for this element of our trading agent. Specifically, we decided to use the SPDR S&P 500 ETF Trust in our models, as it is well-known and has abundant data available.

The second market we will be focusing on is the foreign exchange (forex) market. The forex market solely focuses on exchanging different foreign currencies. This market is significantly different from the index market because it is decentralized and run by numerous banks and financial institutions, it is open 24 hours each day since there are traders all around the world, and it is the most liquid market in the world [9]. We believe that these differences make this a worthwhile market to train our model on, as it increases the diversity of our testing and allows us to analyze the effect of complexity on model performance in various markets. Although the market is decentralized, there is still easy access to market data for exchange rates, allowing us to provide our model with plenty of training data [9]. Since the forex market is the largest market in the world, liquidity is rarely an issue. This allows us to assume that our trades will have negligible effects on the market. This gives us confidence that our models will perform as expected when they are trading real currencies.

We use the ten most traded currencies within the market, known as the G10 currencies, for the pairs to use for our model. Naturally, we chose the United States Dollar as our base currency, as it is the most traded currency within the forex market [10]. We decided to use the British Pound as the other currency, since it has a nice combination of bullish and bearish trends, rather than consisting of a single trend. We then chose 4 other pairs to add to the more complex models as predictors.

2.2 Reinforcement Learning

2.2.1 Q-Learning

Q-learning was an early advancement in reinforcement learning, developed in 1989 by Chris Watkins [11]. As with any reinforcement learning algorithm, Q-learning involves the use of a state space, an action space, and a reward [12]. The state space, S, is the set of all possible states that the Q-learning agent could possibly be in. The action space, A, is the set of possible actions that the agent can take in a given space. Lastly, the reward, R, is information given to the agent that allows it to determine whether the action taken is desirable. Typically, positive reward values are associated with a desirable action, with higher positive values correlating to better rewards. The Q-learning algorithm also considers an update function for each state-action pair. Prior to the learning process, each pair is initialized with an arbitrary Q-value, or "quality" value. At each time step, the Q-value of the current state-action pair is updated by the following equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)], \quad (1)$$

where s_t is the state at time t, a_t is the action taken at time t, α is the learning rate (between 0 and 1), R_t is the agent's observed reward at time t, and γ is a discount factor to weigh future rewards (between 0 and 1) [12]. This function calculates the observed Q-value using the value of $R_t + \gamma \max_a Q(s_{t+1}, a)$, since the observed Q-value is simply the reward plus the expected future reward of the best action ($\max_a Q(s_{t+1}, a)$). The best Q-value is used because we want to take the best action. The function then calculates the difference between the observed Q-value and the old Q-value, and then moves the new Q-value towards the observed value by adding this difference, scaled according to the learning rate.

The Q-learning agent selects actions through an ϵ-greedy process, which aims to balance environmental exploration and the optimization of what is already known about the environment from previous actions [13]. This works by using a probability, ϵ, to determine at each step whether the agent will explore (choose a random action) or exploit (choose the action believed to be best) [13]. The starting value of ϵ is typically close to 1.0 and decays by a given factor throughout the learning process to a value close to 0, all of which are determined by the researchers [13].

2.2.2 Deep Neural Networks

We can further extend our discussion of Q-learning by adding an extra component to the system. This can be done by employing the use of deep neural networks, which are especially useful if there is a very large state space, something that the simpler Q-learning model cannot handle well. As well as being well-suited to deal with high-dimensional data inputs, neural networks are also capable of making use of new data as training happens in an incremental fashion. A typical neural network is represented by a collection of neurons, which are arranged as several layers [14]. When given standardized data, the neural network is able to derive features based on the input. This means that Equation (1) defined in the previous section can be represented as a neural network, taking in the state as an input, and eventually producing a Q-value for each action as an output [15].

2.2.3 Fully-connected Neural Networks

There are multiple types of neural network architectures, each with their unique benefits given different contexts. In this subsection, we expand on the popular fully-connected neural network (FCNN) architecture, as it is the architecture used in this project. Such networks can also be referred to as feedforward neural networks. An example of such a network can be seen in Figure 1, where every neuron computes the output $f^{(3)}$ using all the outputs from the previous layer [16].

In order to train the network, one simply needs to generate the error of the current network and adjust the weights of each edge to try and minimize that error. A common approach is to use stochastic gradient descent to adjust the weights. Stochastic gradient descent translates the neural network into a system of equations, calculates the local gradients of all the weights, with respect to mean squared error loss, and then optimizes the weights to minimize the loss [16].

One of the benefits of FCNNs is that they are versatile in their potential configuration, in that the number of layers and the number of neurons in each layer can be modified [16].

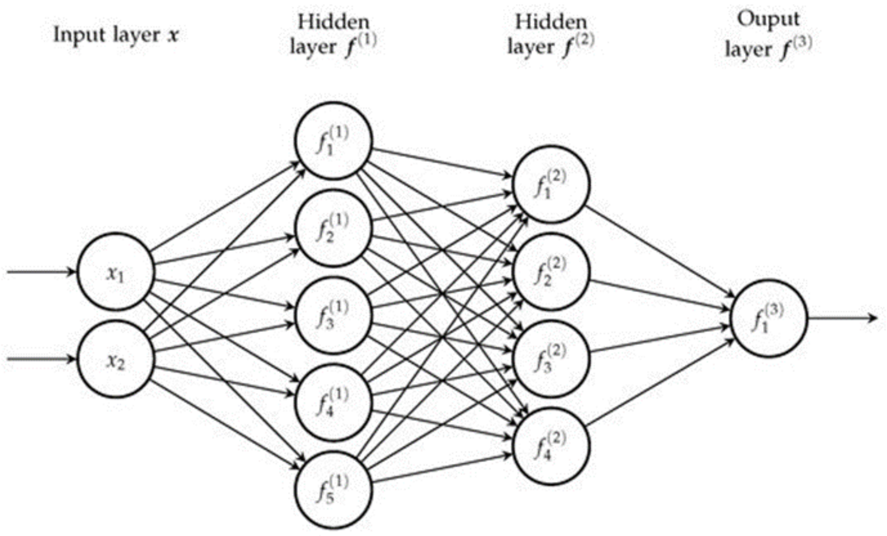

Figure 1: An example of a deep feedforward neural network consisting of and input layer with two neurons, two hidden layers with five and four neurons respectively and an output layer with one neuron [16].

2.2.4 Deep Q-Networks

We can now use the information we have covered to formally introduce the deep Q-network (DQN). The primary difference between the Q-learning and deep Q-learning (DQL) is the addition of a deep neural network. As mentioned previously, this network is used to approximate a Q-value by learning weights. More traditional neural networks learn these weights by minimizing the training error, for example, in classification problems we have access to the actual classifications of the inputs, and thus we can easily calculate, and minimize the error we get. However, with DQL networks we do not have the luxury of knowing the correct Q-values, thus we can not simply calculate the error. Instead we use a similar equation as Equation (1), namely Equation (2), where the target output of action a_t with the input of s_t, is the observed reward plus the expected future reward, as calculated by the NN. We can compare these target values with the predicted values to generate the error, and then use that error to train the NN

$$Q_{target}(s_t, a_t) = R_t + \gamma \max_a Q(s_{t+1}, a). \tag{2}$$

Based on this, we can say that the goal of the DQN is to learn weights that can approximate Q-values with the most accuracy. The agent will receive an initial state from the environment, x, within which it can take certain actions. In the beginning, the agent will adopt an ϵ-greedy policy, meaning that it will take random actions with high probability [13]. Over time, the agent will take fewer random actions using the algorithm to approximate Q-values and weights, based on subsequent states provided by the environment [17].

2.2.5 Improvements to the Deep Q-Network

Over the years there have been several improvements to the deep Q-learning approach, namely the use of target networks and experience replay.

"Experience Replay" creates a more stable training environment by allowing the agent to use past experiences [18]. For example, say we have a replay memory M, composed of a tuple of states, actions, and rewards gathered from previous time steps. At each time step t, a random mini-batch is selected from M that is then used to train the neural network weights using stochastic gradient descent. In practice, the replay memory tends to be large [14]. Experience replay has been shown to increase sample efficiency and reduce the autocorrelation of samples [16]. However, one potential drawback of this technique is that if the data is imbalanced, the model can become biased from this mechanism [19].

The other important feature is the use of a slowly changing target network. This target network has the exact same architecture as the Q-network; however, the difference is that its weights are only updated after a specified amount of time steps [16]. Essentially, the main deep network will find the best action through steps previously discussed. Then, using this action, the target network will determine the target Q-value resulting from taking that action in the next state. By training in this fashion, we can attain better results in a shorter amount of time [18].

2.2.6 Double Deep Q-Network

Deep Q-learning tends to have a bias due to its tendency to always select the largest Q-value, resulting in the overestimation of the expected reward of each action. This is because in DQN, the action selection and evaluation use the same target network [20]. Van Hasselt showed in "Deep Reinforcement Learning with Double Q-Learning" that this bias can adversely affect the training process if it does not apply uniformly [16]. To solve this problem, the DDQN separates estimations of action values from the selection of actions. It does this by using two different networks, one to select the next action for a given next state, and another to obtain the best action value [14].

2.3 Related Works

Due to the success of reinforcement learning in other fields, it was quickly applied to financial markets. As touched upon earlier, in 1998 Moody and Saffell developed two reinforcement learning models to buy and short S&P 500 stocks, one of which was a real-time recurrent learning model and the other was a simple Q-learning model [2]. They found that both models outperformed the simple buy-and-hold strategy. More recently in 2017, Deng et. al. developed a more advanced reinforcement learning model using deep learning and fuzzy layers to help alleviate the effects of noise in the data [21]. Their models performed better than older reinforcement learning models on trading the stock-IF index (the first index-based futures contract traded in China) and silver and sugar commodity futures contracts [21].

However, neither of these papers provided any analysis of the complexity of the data they provided to their models. They only compared their models to other stock trading models. As mentioned previously, the curse of dimensionality can cause these models to perform very poorly if the dimensional space of their data is too high compared to the number of data points. Thus, it is important to fine tune the dimensionality of the data we give to each of these models. Since the papers lack that sort of analysis, perhaps their models could have performed better if they were given simpler data.

A more recent paper briefly addressed this issue by adding a neural network between the state space and the model [3]. Wu et. al. developed and applied a Gated Recurrent Unit (GRU) to their model. GRU is a neural network that separates the model from the raw state space. The network's goal is to learn the best state variables, or combination of state variables, that yield the best results when training the model [3]. This aims to allow the model to ignore irrelevant or noisy state variables and focus on the best predictors, hopefully minimizing the effect of the curse of dimensionality. Since this paper also fails to test their model against simpler models that exclude the GRU layer, it might be the case that the GRU layer actually leads to overfitting, leading to the conclusion that hand-picked state spaces are the better option. It is important to take this into account when creating these models and thus there is a need to find out what role state space complexity plays in the performance of models.

Reinforcement learning models have also been developed for the forex market. Neves et. al. developed a deep Q-learning model which traded in the Euro to USD market [22]. The model performed relatively well with an average of 16% yearly profit. Another group of researchers created a deep Q-network to trade within the USDJPY and EURUSD markets [9]. Their model was able to outperform a simple buy/hold strategy for both markets as well as experienced human traders within the EURUSD market [9]. However, there was no statistically significant difference between the experienced human traders and the model within the USDJPY market [9]. Like the previous papers, this paper did not provide any analysis of the model with respect to simpler state spaces, which leads to the same issues listed above.

3 Methodology

The goal of this project is to evaluate how the complexity of data provided to a DDQN agent affects its performance. Our project was, broadly speaking, split into two main phases: creating the trading agent and refining the agent. Our team used open-source code from Chapter 22 of Jansen's *Machine Learning for Algorithmic Trading* as the base framework for the trading environment, agent, and architecture [5]. This textbook serves as a resource for RL applications in finance and includes a number of examples for doing so, primarily using Python. From this, our team was able to adapt the textbook example to the project goals and objectives, which were gathered and adapted through weekly meetings with Dr. Osterrieder. He desired for our team to focus on the effect of data complexity on a model rather than working to optimize the model itself, the latter having been extensively researched in other works. The second phase of the project involved the training and testing of the agent within state spaces, each with varying complexities. To work towards a comprehensive analysis of complexity, our agent was trained and tested with data from both the equity index and foreign exchange markets. Aspects like the DDQN algorithm, random seed, and network architecture were kept the same between state spaces to allow for accurate comparison.

Python was the language of choice for the implementation, primarily due to it being the language used in Jansen's example [5]. Furthermore, Python is a relatively easy to use language, with members of the group having previous experience using the language in machine learning applications. Lastly, Python boasts a vast array of third party libraries for machine learning, reinforcement learning, and neural networks, which were employed to make the implementation of algorithms and architecture significantly less complicated.

3.1 Data Sources

For the equity index aspect of the project, we elected to use data from the SPDR S&P 500 ETF Trust (SPY), Nasdaq Inc. (NDAQ.O), the SPDR Dow Jones Industrial Average ETF Trust (DIA), the United States Oil Fund (USO), and SPDR Gold Shares (GLD). Our experiments center around using the daily closing price for each of these various assets in order to calculate investment rewards for the agent. The data itself was gathered from the Eikon financial tool by Refinitiv, as well as Yahoo! Finance [23]. For each of SPY, NDAQ.O, DIA, USO, and GLD, the data spans from their respective inceptions to December 31st, 2020.

We ran an intraday trading model on the index fund data, where the model traded every five minutes. We used the same index funds as above, but instead gathered the price of the stocks in five minute intervals from Refinitiv. We were only able to go back three months but due to the frequency of the interval this still gave us more data points than the 20 years of daily data. Due to the intervals being so small there were many data points that were blank. This is to show that that stock was not traded within that five minute interval. To standardize the data and for the sake of not removing chunks of time from the data set, we simply filled the null values with the last recorded price.

For the forex component, we used daily exchange rate data between the United States Dollar and Great British Pound (GBPUSD), Euro (EURUSD), Swiss Franc (USDCHF), and the Japanese Yen (USDJPY). We also added an unrelated exchange rate between the Canadian Dollar and the New Zealand Dollar (NZDCAD). We used the data from February 7th, 2002 to December 31st, 2020, gathered through Refinitiv [24]. We were restricted by Refinitiv's 20 year-to-day history limit, but also found that Yahoo! Finance only had data from December 1st, 2003 onward. Refinitiv was chosen in order to maximize our available data. We used the daily close prices for each of these exchanges, as specified in Refinitiv, since the market technically only closes on Sundays.

3.2 Model Design

The state space for equity indices is broken down into six models, each building upon the previous model. That is, Model X has all of the elements of Model X - 1, in addition to what else is listed. The related indices mentioned under Model 5 are NDAQ.O and DIA, as they are two other popular ETFs. The unrelated indices for Model 6 are USO and GLD, aspects of the materials stock sector recommended to us by Dr. Osterrieder.

Model 1	1-Day Return
Model 2	Previous Action
Model 3	Previous Price
Model 4	2-Day Return 5-Day Return 10-Day Return 21-Day Return
Model 5	2 Related Indices' - 1-Day Return - 5-Day Return - 21-Day Return
Model 6	2 Unrelated Indices' - 1-Day Return - 5-Day Return - 21-Day Return

Table 1: State spaces for each equity index model.

For the sake of comparability we used the same basic state spaces that we used for the equity index models for the forex models. The related exchange rates we chose were the U.S. Dollar to the Euro and the U.S. Dollar to the Swiss Franc, as they are geographically close to the United Kingdom. For the unrelated exchange rates we used the U.S. Dollar to the Yen and the Canadian Dollar to the New Zealand Dollar. As with the equity index state spaces, each state space for the forex models builds off of the last.

To train each agent, we decided to use the following reward function for our models:

Model 1	1-Day Return
Model 2	Previous Action
Model 3	Previous Price
Model 4	2-Day Return 5-Day Return 10-Day Return 21-Day Return
Model 5	2 Related Exchange Rates' - 1-Day Return - 5-Day Return - 21-Day Return
Model 6	2 Unrelated Exchange Rates' - 1-Day Return - 5-Day Return - 21-Day Return

Table 2: State spaces for each forex model.

$$r_n = (a_n * M_{n+1}) - (tradingCosts * |a_n - a_{n-1}|) - dailyCost, \tag{3}$$

where

$$a_n = \max(neuralNetwork.predict(S_n)) \in \{-1, 0, 1\}$$

and

$$dailyCost = \begin{cases} 0 & a_n \neq a_{n-1} \\ 0.0001 & a_n = a_{n-1} \end{cases}$$

where a is the action, M is the 1-day return of the market, and S_n is the state at time n, which is a list of all the state variables as defined in Tables 1 and 2. The $a_n * M_{n+1}$ component is the major factor within the reward function. It simply calculates the percent difference incurred through the action and the resulting stock return. For example, if the agent decided to buy the stock and the stock went up by 0.01, the agent would have gained a reward of 1 * 0.01 = 0.01, whereas if the agent decided to short, it would have lost -1 * 0.01 = -0.01. The number of trades are calculated by the equation $|a_n - a_{n-1}|$, because if both actions are the same, the agent does not need to trade anything to execute the action. If the agent wants to buy and it previously only held cash, it would only have to buy $|0 - 1| = 1$ unit of the stock. Whereas if the agent wants to then short, it would first have to sell all the stock it already has and then short that amount again, resulting in $|1 - (-1)| = 2$ units of trading it would have to do. The amount needed to trade is then multiplied by the trading costs. The daily cost is used to disincentivize the agent from being too passive. If the agent repeats an action, the daily cost is set to the specified value, in our case 0.0001, but if the agent performs a different action, and thus trades something, the daily cost will be set to 0. For the equity index daily trading agent we set the trading and time costs to 0, as one can typically buy SPY stock without incurring significant trading costs, but for the forex and the equity index intraday agent we set the trading cost to 0.001 and the time cost to 0.0001.

3.3 Neural Network Architecture

Input, Length of S_n	256-node dense	256-node dense	.1 drop out	Output, size 3

Table 3: Neural network architecture

For our DDQN agent we used two neural networks. Each neural network started out with a layer of input nodes, the size of the current model's state space, and then went through two 256-node dense layers and one drop out layer with 0.1 dropout frequency. The dropout layer is used to help prevent overfitting. It then went to a 3-node output layer, where the three nodes correspond to the three actions.

These neural networks were tasked with predicting the Q-values for the state passed in. Specifically each output node was to output the resulting Q-value if that action was taken at the input state. We used two networks in the form of a target and online network, where the online network is trained while the target network acts as a base to predict future Q-values. This helps the agent converge quicker, as the future predictions stay relatively stationary as training continues [18]. We trained the network by using the following equation:

$$TargetValue_n = r_n + \gamma \max(TargetNetwork.predict(S_{n+1})), \quad (4)$$

where r_n is the reward observed at time n, S_{n+1} is the state observed after taking the action, and γ is the future reward discount factor, which tells the agent how much to prioritize immediate rewards versus future rewards. This equation simply takes the reward we received and then adds the reward we are executed to get in the new state (scaled according to γ). It uses the target network to predict the future reward. If we instead used the online network to predict future values, the target values will fluctuate a lot, since the future rewards will constantly change as the agent trains. We use the target network because it will stay stationary as the online network trains. In our training, after every 100 batches, we updated the target network with the online network's weight.

3.4 Training Process

Before the agent begins the training process, several variables and hyperparameters are set. These variables are kept the same between the different models. One such important variable is the trading cost incurred when the agent takes a long or short position. After discussing the matter with Dr. Osterrieder, the trading cost for our index fund model was set to 0, as per his request. However, the opposite is true for our foreign exchange and intraday models. In these two models, trading costs become more significant due to the frequency with which the agent is trading. The trading cost is set to one basis point (bps), equivalent to 0.1% (or 1/10th of a percent), whilst at every step where the agent does not take a different action, there is a time cost of 0.0001. The batch size was set to 4096, which was kept the same from Jansen's original textbook model [5]. Each model is trained for a maximum of 1,000 episodes, with each episode containing 252 days (representing the average number of trading days in a year).

We define each trading day that the agent operates in as an episode step. At the beginning of each episode, the training environment will reset itself and the agent will begin training in the first episode step. The agent's first course of action is to choose an action given the current state. It does this via the epsilon decay policy described in earlier portions of this paper. Specifically we had epsilon decay linearly for 250 episodes. This means that it will take 250 episodes for the agent to transition from choosing a random action 100% of the time to eventually having a 1% chance of choosing a random action, and then epsilon is multiplied by .99 for each episode after. If a random action is not taken, the agent instead uses the Q-function to determine the best possible action to take at the given step. The simulation performs the agent's requested action, giving it an appropriate reward, and updates the portfolio's Net Asset Value (NAV). The NAV calculated by the following formula:

$$NAV_{n+1} = NAV_n * (1 + r_n).$$

This formula simply updates the NAV by multiplying it by 1 plus the reward of that time step (r_n), since the reward is just the percent change of the NAV. By tracking the difference between the agent's NAV and that of the market, we can track the agent's performance.

3.5 Training and Testing Split

To accurately assess how our model performed, we split our data into a training set and a testing set. For each environment, we dedicated the last two years of trading data to be used for testing. For example, if we have 20 years worth of trading data, the first 18 years would be used to train the agent whilst the last two years would be used for testing its performance against the market.

3.6 Improving the Original Model

We found two main ways we could improve the model. The first flaw was how it handled performing actions. Previously, the 1-day returns would be calculated at the end of the trading day and an action would be chosen. However, since the trading day was already over, the action could not be executed until the start of the next trading day. Because we only use the closing price, our simulation would execute the action at the closing price of the next day. This means that it was taking a full 24 hours for our actions to be executed, making it much harder for the agent to learn due to market changes that would occur in the meantime. To improve upon this issue we assumed we could execute the action at the closing price of the current day. With our data being only of closing price, this is not an entirely realistic assumption, but we feel it is close enough.

The more significant improvement we made was changing the reward function. The neural networks are passed sets of data in the format of

$$\{S_n, S_{n+1}, a_n, r_n\}, \tag{5}$$

where

$$a_n = \max(neuralNetwork.predict(S_n)),$$

and

$$r_n = (a_{n-1} * M_{n+1}) - costs$$

where S is the state, a is the action, r is the reward, and M is the 1-day return of the market. The issue is in the reward function, specifically $a_{n-1} * M_{n+1}$. This is because a_{n-1} is the investment (or lack thereof) caused by the previous action, since it took 24 hours to execute. So pairing a_n with r_n is not ideal, as the reward's most vital component is calculated using the previous action. Simply put, a_n was being rewarded not based on the reward caused by the current action, but based on the reward caused by the previous action. This issue led to our agent never investing. We changed this this by allowing the action to execute at the closing price of the day it is calculated, thus the reward for that action becomes

$$r_n = a_n * M_{n+1} - costs$$

This then pairs the action with the reward it generated, rather than the reward the previous action generated. Our model was able to learn much easier, thus leading to better results. In the example where data increases from \$1.00 to \$50.00, then decreases from \$50.00 to \$1.00, the model was able to successfully predict when to buy and short.

We tested the fixed model on the SPY data again and the agent learned to always buy if there were no trading costs. However, it still held when there were trading costs. This was more expected than the previous behavior, but still not ideal. But once we tested the second model, where it also knew what the previous action was, the agent learned to always buy. This is because the agent learned that if it already invested all available capital, it will not be penalized by trading costs, so it was better to choose the previous action as to not get hit with the trading costs. Since SPY generally tends to increase, the agent learned to invest in it, as on average it will make money. And on the third model, the agent started to learn more complex strategies, rather than always performing a single action.

3.7 Software Development Environment

Our sponsors and advisors gave our team much leeway in the decision about what tools to use when it came to software and packages for development, albeit with some guidance. Where possible, we opted to use tools that we were most familiar with to save time and resources. In cases where additional resources were needed, the appropriate research was done in order to determine the best available tool.

3.8 Project Management Software

GitHub was the chosen tool for version control of our code base. It was chosen due to its usefulness and popularity in code management as well as team familiarity. We used a shared repository to organize and maintain the code base. Branches were initially created to divide up work into different sections, which were eventually merged. GitHub also allowed us to access different versions of files as well as acting as a backup.

Google Drive was used to store administrative files like presentations and meeting minutes, as well as technical files in terms of raw data and testing/training results.

3.9 Programming Integrated Development Environments (IDE)

Our team used three different IDEs for the duration of the project based on personal preference and need. In early development stages of the project, PyCharm and Jupyter Notebook were used. PyCharm is a cross-platform IDE developed by JetBrains that provides many useful features for Python developers. Some of these features include code completion, error and syntax highlighting, keyboard shortcuts, and a debugger [25]. Our team's familiarity with the product, coupled with the IDE's Jupyter Notebook integration made it an extremely useful tool for development and debugging.

Jupyter Notebook was the other IDE used during early stages of development. It is an application that allows the editing and running of documents via web browser. It can be installed on a remote server or accessed through the Internet. Jupyter Notebooks can contain both code and rich text elements, which are divided into runnable cell blocks [26].

One of the downsides to running our code on the previous two IDEs is that they use the resources of the machine it is running on. As the project progressed into final training and testing, it quickly became apparent that using our own CPUs/GPUs would prove limiting given time constraints. We needed a way to run models while simultaneously being able to work on analysis and bug fixes. To solve this problem, our team requested funding from the ZHAW through Dr. Osterrieder for a month-long subscription to Google Colab Pro Plus. Google Colab is a cloud based Jupyter Notebook environment that provided our team with invaluable benefits [27]. The primary benefit being the capability for background execution, allowing us to run a model after exiting the browser. Additionally, the subscription provided access to longer run times, more memory, and faster GPUs, allowing our team to quickly and confidently experiment with different state spaces.

We also worked to make the Jupyter Notebook file, and thus in part the Google Colab file, more user friendly. The Jupyter Notebook used for this project was adapted from Section 22.4 of Jansen's Machine Learning for Algorithmic Trading, 2nd Ed GitHub repository [5]. The notebook was then further integrated into GoogleColab, where we made several formatting changes and recorded more in-depth notes for future users. These changes include an overview and introduction to our project, instructions for proper use, data descriptions, environmental and modular analysis, and additional comments. We also added a section which includes a variety of variables that the user can easily adjust to specify how they want the model to run. For example, there is a model variable which specifies which state space to use, a max_episodes variable to specify how long to train the agent for, trading_cost_bps and time_cost_bps variables, and an epsilon_decay_steps variable to specify how many episodes it takes for epsilon to fully decay.

4 Results

In order to quantify the validity of our hypothesis, we visualize the results of training the agent through a series of comparative graphs. The models using equity index data and those using foreign exchange data are distinguished from each other due to the difference in their simulated markets. These simulated markets are equal between models in their respective market domains, since each model uses the same seed to randomly simulate the market. For the graphs, we compare the simulated market behavior and the behavior of the various models, in order to measure their performances against each other to see which is “the best.” For scaling and visibility purposes, models with greater returns may be graphed separately to avoid overshadowing weaker models.

Two graphs are employed for each market space. In the first graph, we measure the moving average of each model’s annual return over the last 100 episodes, where each episode is a simulated year in the market. For example, at episode 500, we consider the annual returns from the 401st simulated year to the 500th simulated year, average them, and plot that point on the graph. These points are the average minus one for easier readability, and can be interpreted as, “At episode X, the last 100 episodes saw an average increase or decrease of capital by Y percent.” The plot of the graph begins at episode 99, as the data is 0-indexed and requires 100 values to produce an average. Also included in this graph is the moving average of the simulated market, allowing for easier comparison to the model averages.

4.1 Index Funds

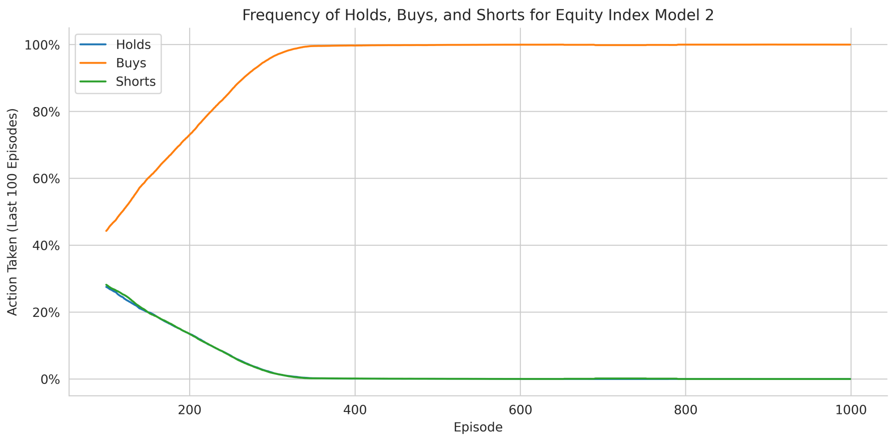

Figure 2: Graph displaying the percentage frequencies of the actions across epochs for index funds model 2.

We begin our analysis by focusing on the actions learned by the agent during the training process of different models. In Figure 2, one can see the action frequency of the agent over 1000 episodes, recalling from Table 1 that only 1-day returns as well as the previous action were used in model 2. From the graph, it is evident that over time, the agent learns to buy. After around 300 episodes, the agent determines the best strategy is to buy 100% of the time. It is important to mention that graphs for action frequency for models 1 and 2 are essentially identical, with no noteworthy differences between them.

Comparatively, we see the agent begin to make more nuanced trading decisions when looking at the action frequency of model 3, as shown in Figure 3. Model 3 builds upon the first two models by also considering the previous stock price. Initially, the agent continues learning to buy until a peak of about 80%. At this point, the frequency of buying seems to dip, then oscillate around 70%, with the agent doing a combination of holding and shorting for the latter 30% of actions. Even though we can conclude from this graph that the dominant strategy is buying, it may also be an indicator that giving the agent access to additional stock price data can aid in its decision-making.

This claim is further supported when observing action frequencies of more complex models. Looking at the graph from Figure 4, we can see a significant

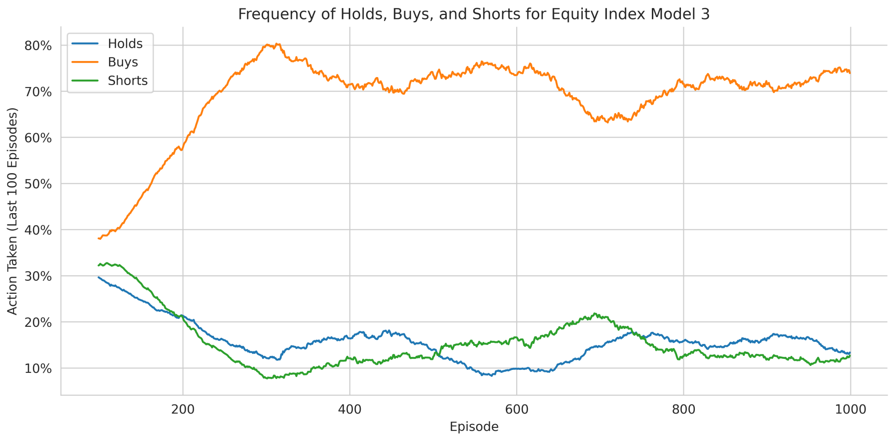

Figure 3: Graph displaying the percentage frequencies of the actions across epochs for index funds model 3.

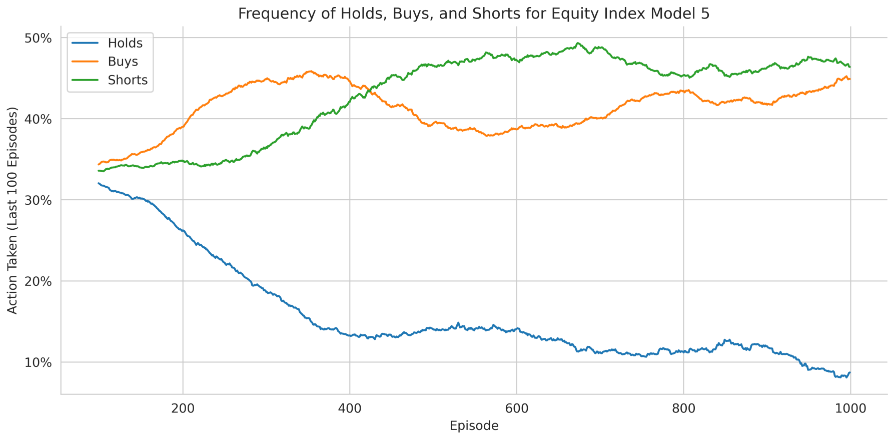

Figure 4: Graph displaying the percentage frequencies of the actions across epochs for index funds model 5.

change in behavior of the agent relative to previous models. Whilst the agent's decision to conduct less holds over time is shared between all models, it seems the addition of other equity indices leads to more balanced trading actions. This behavior is very different to earlier graphs, where we saw the agent largely prioritize a single action.

Another useful measure of performance is to graph the rolling average of agent and market returns.

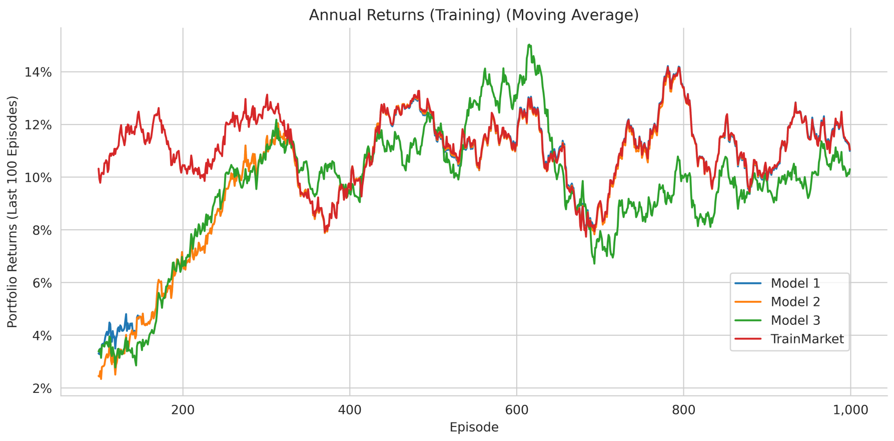

Figure 5: Graph displaying the index fund training returns for models 1 - 3 for each epoch, with a moving average of the past 100 epochs, the NAV was reset to 1 after each epoch.

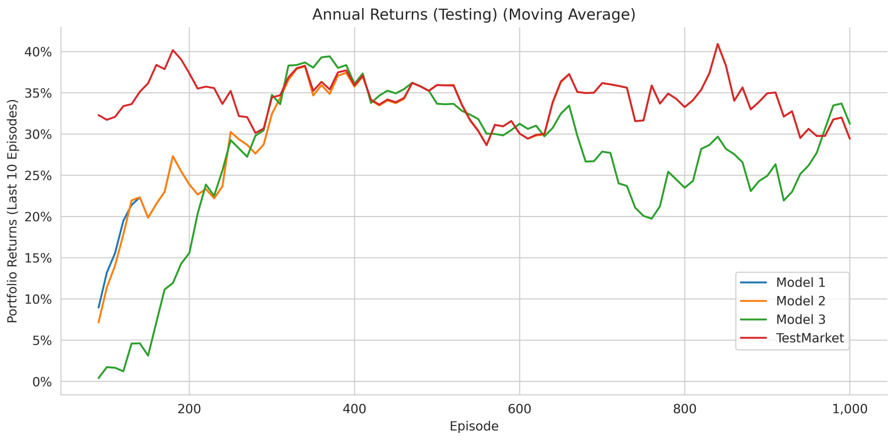

Figure 6: Graph displaying the index fund testing returns for models 1 - 3 for every 10 epochs, with a moving average of the past 10 tests, the NAV was reset to 1 after each epoch.

Figures 5 and 6 show the annual returns of the agent and market for the training data and testing data respectively. For the training data, models 1

- 3 show that the agent did primarily learn to buy. Comparing this to the testing data in Figure 6, we can tell that these models performed poorly, and that there was not overfitting.

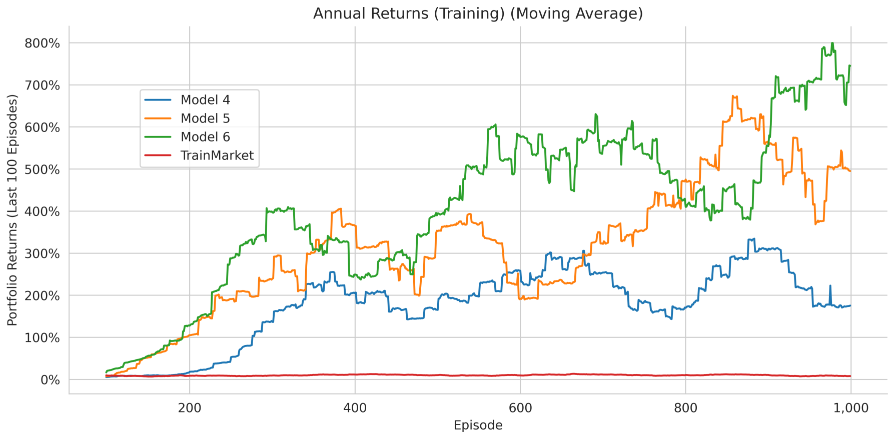

Figure 7: Graph displaying the index fund training returns for models 4 - 6 for each epoch, with a moving average of the past 100 epochs, the NAV was reset to 1 after each epoch.

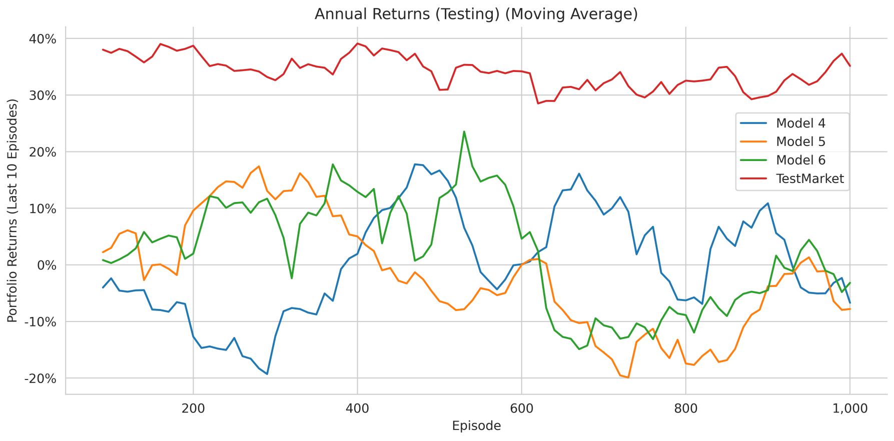

Figure 8: Graph displaying the index fund testing returns for models 4 - 6 for every 10 epochs, with a moving average of the past 10 tests, the NAV was reset to 1 after each epoch.

Figures 7 and 8 show additional annual returns but now for models 4 - 6.

In the training data, models 4 - 6 tremendously outperform the market by hundreds of percent, with model 6 displaying the best performance by far. However, when comparing to the testing data we see overfitting of models 4 - 6. This is most likely due to the presence of too much information that may have hindered the performance of the agent.

4.2 Forex

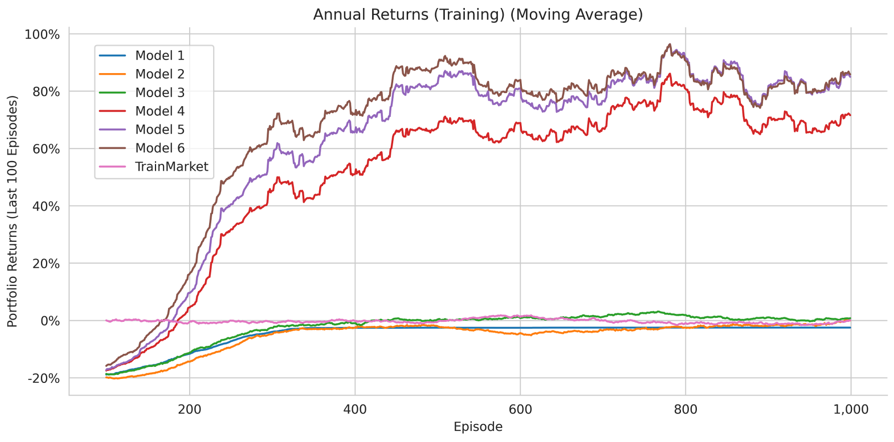

Figure 9: The forex training returns for each epoch, with a moving average of the past 100 epochs, the NAV was reset to 1 after each epoch.

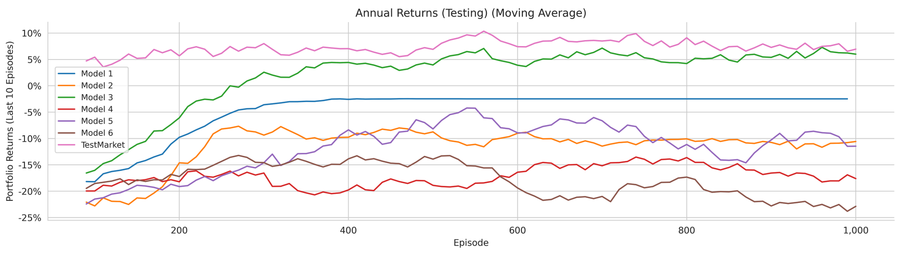

Figure 10: The forex testing returns for every 10 epochs, with a moving average of the past 10 tests, the NAV was reset to 1 after each epoch.

The forex models followed a similar pattern to the equity index models. Like the equity index models, models 1 - 3 hovered around a couple percentage points, and then all of a sudden models 4, 5, and 6 all performed much

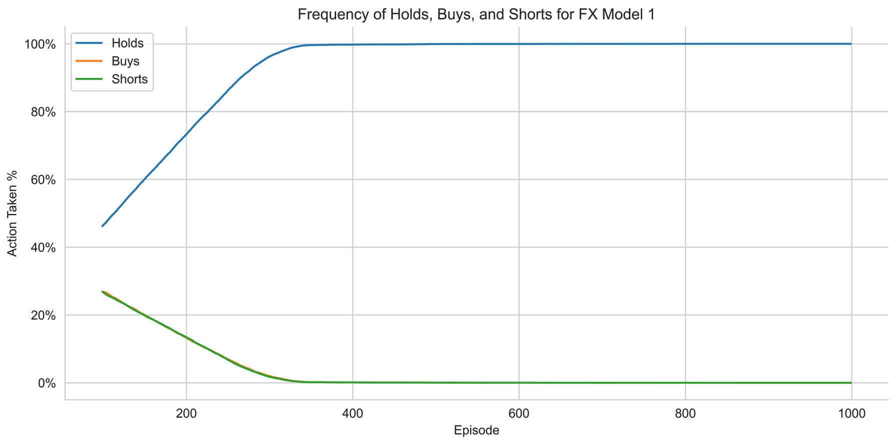

Figure 11: The frequencies of the actions across epochs for forex model 1.

better with the training data, as shown in Figure 9. However, as shown in Figure 10, the testing data shows that models 4, 5, and 6 greatly overfit the training data, as they all ended up losing anywhere from 5 to 20% of the capital per year. This is much different from the Index Fund data where model 6 actually performed pretty well around episode 270.

Like with the equity index data, model 1 just opted to always hold its money, as shown in Figure 11, since there was not enough information within the 1-day returns to learn anything. Once we moved onto the 2nd model, the agent learned to always short the exchange, as shown in Figure 12. It learned to always short because over the training data the exchange rate went down slightly, however when it comes to the testing data this was an even worse strategy because the exchange rate went up during the two years of testing data. Model 3 is where the agent was able to learn something. As we can see in Figure 13 the agent chose between all three actions as it learned, rather than just converting to only choosing one action. By the end of training it chose buying about 70% of the time, shorting about 20% of the time, and holding 10% of the time. This shows that the agent developed some non-trivial strategy for investing. Figure 10 shows this strategy outperforms the prior two models' returns on the testing data. However this model was still unable to match the market's return. This is probably due to the fact that the agent has trading fees, and since the forex market doesn't fluctuate that much, it is much harder to make back the trading costs. Then for models 4, 5, and 6, the testing data shows that the agent horribly overfit to the training

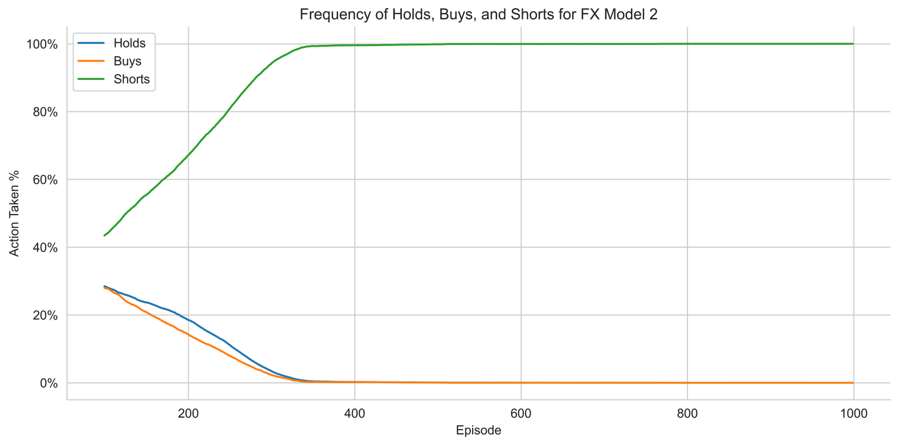

Figure 12: The frequencies of the actions across epochs for forex model 2.

data, as all three models performed much worse than model 3. Models 4 and 6 even performed worse than both models 1 and 2 on the testing data, and even model 5 only performed better than model 2 about 50% of the time.

This shows that within the forex market it is very easy to over-fit to the data, and a less complex data set might be better than a more complex model. However if the data set is too simple it will perform worse. Thus it is important to add enough complexity so the model can learn, but not too much that it overfits. One way to verify this is to save some data for validation runs, so an accurate measurement of the agent's performance can be observed.

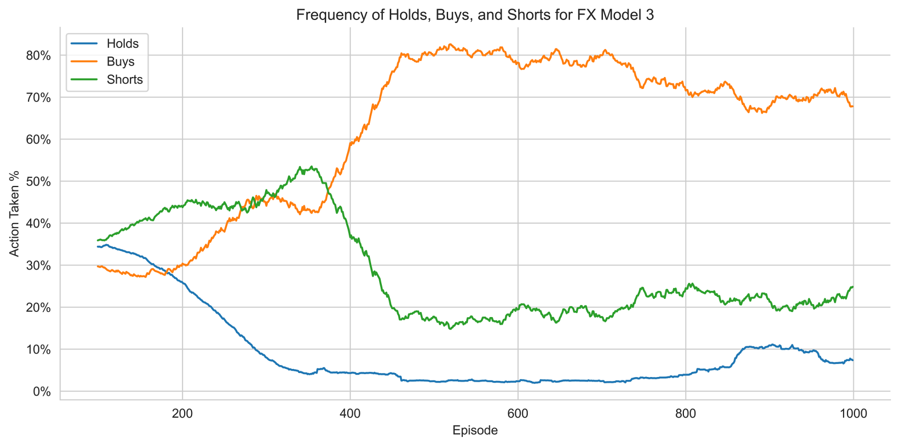

Figure 13: The frequencies of the actions across epochs for forex model 3.

4.3 Intraday

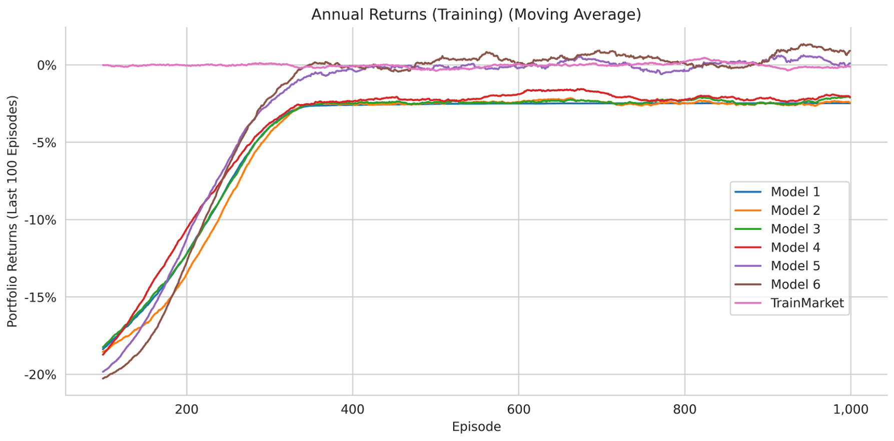

Figure 14: The intraday training returns for each epoch, with a moving average of the past 100 epochs, the NAV was reset to 1 after each epoch.

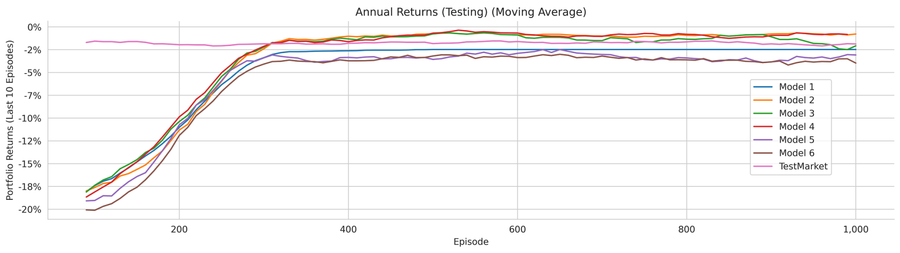

Figure 15: The intraday testing returns for every 10 epochs, with a moving average of the past 10 tests, the NAV was reset to 1 after each epoch.

We also used intraday data on the models. For these models the agent was tasked to trade every five minutes, and the state variables that involve daily returns are scaled to five minute returns, so the 1-day return was changed to 5-minute return, the 2-day return was changed to the 10-minute return, etc. As far as the agents' performance, Figure 14 shows that they substantially differed from the prior two markets. We did not see the typical large yearly returns for models 4, 5, and 6. Instead only models 5 and 6 suddenly increased in returns, compared to the other models, but they only increased to be about even, or slightly better than the test market. However even with

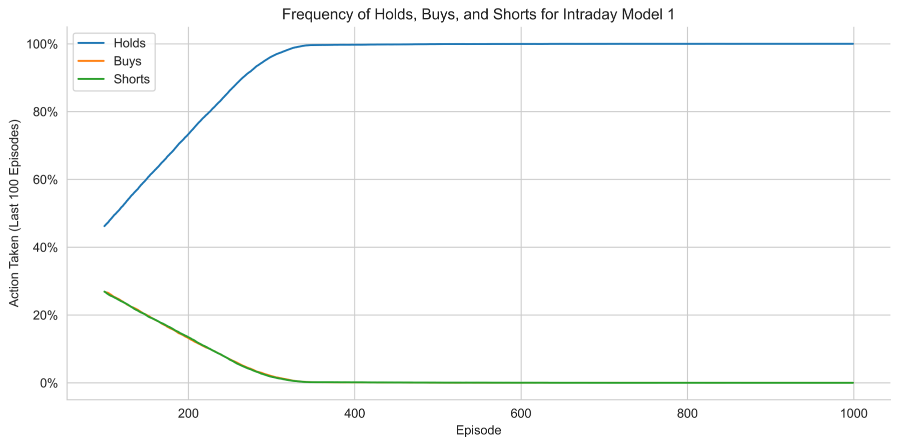

Figure 16: The frequencies of the actions across epochs for intraday model 1.

these low returns the agents were still susceptible to overfitting. As we look at the returns of the testing data we can see that models 5 and 6 performed even worse than model 1, which only ever held onto its cash. Looking at Figure 15 it is clear that the best models were 2, 3, and 4, where 3 is slightly worse than 2 and 4. Though all of these models still only have about a -1% return, it is slightly better than the market return.

As far as the strategies developed by the agent, they followed the same pattern as the forex and daily index funds models. As shown in Figures 16, 17, and 18, in the first two models always either held or shorted, while in models 3, 4, 5, and 6 learned more complex strategies where the agent would short most of the time but also occasionally buy.

5 Conclusion

This project aimed to provide an exploration and analysis of how altering the complexity of data available to a trading agent affects its performance. We trained and tested the agent in the index fund and foreign exchange markets, with data obtained from Refinitiv. The initial stage of the project involved designing state spaces that ranged from simplistic to complex, as well as working with the code base to modify the environment to our needs. In the latter half, the gradual training and testing of our agent within these state

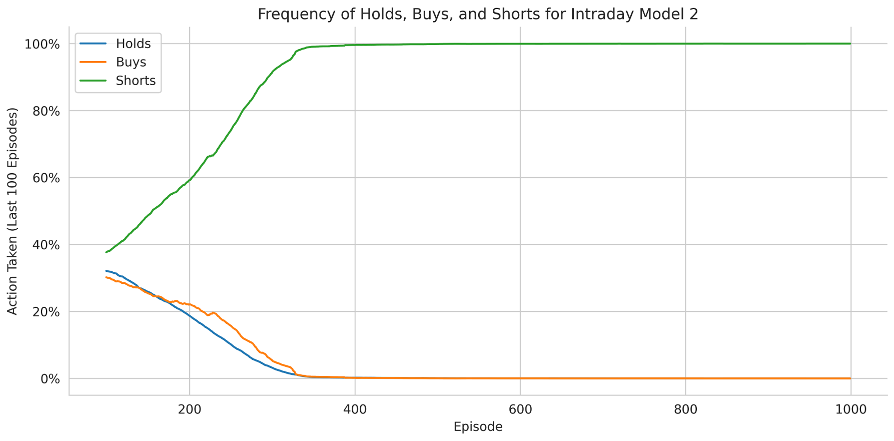

Figure 17: The frequencies of the actions across epochs for intraday model 2.

spaces took place. Through various bug fixes and additions to the code, we were able to output valuable data visualizations. Overall, we believe that our investigation supports our initial thesis, that the addition of more data to the agent would improve its performance, only up to a certain point. Based on our findings, it would not be favorable to further complicate the data for a slightly better return. Instead, a more appropriate strategy would be to use only the more relevant features. We encourage experimentation with different features to find those that are most relevant. We found that models 3 and 4 performed the best across all markets. Our testing data showed that within the forex and daily equity indices markets all models performed worse than the market. However, intraday models 2, 3, and 4 slightly outperformed the market. These models averaged about a -1% return, as opposed to the daily equity indices models which had about 25% returns. But the daily equity indices models' returns were still less than the 35% average return of the market. Although none of these models performed particularly well, our results illustrate the importance of ensuring that financial models do not overfit to the training data.

5.1 Future Work

Based on the results gathered, the team has some suggestions for future work in dimensional complexity analysis. Chiefly, we think that it would be

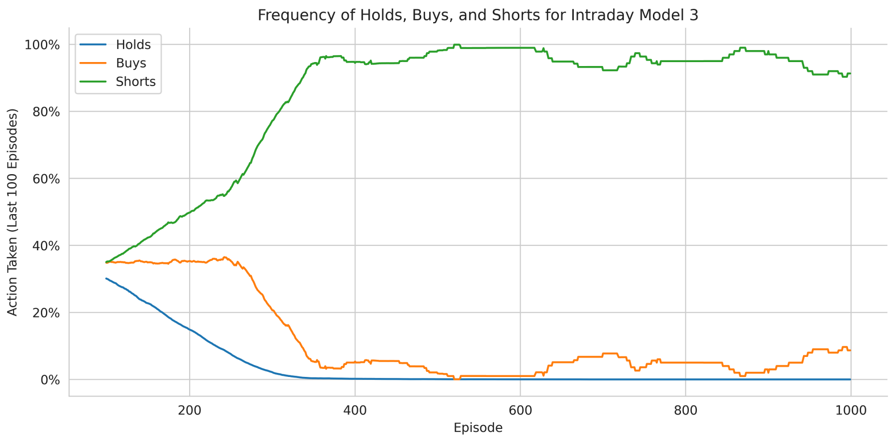

Figure 18: The frequencies of the actions across epochs for intraday model 3.

beneficial to form models more incrementally, adding only a single dimension at a time. There is a significant jump in annual returns between model 3 and model 4 in both equity index and forex models, as demonstrated in the difference between Figures 5 and 7 (for equity indices), and in Figure 9 for forex. Models 4, 5, and 6 express similar behavior, so there appear to be some missing steps between model 3 and model 4. Rather than adding all of 2-day, 5-day, 10-day, and 21-day returns to model 4, it may help to have three additional models where the first adds 2-day returns, the second adds 5-day returns, the third adds 10-day returns, and then the updated model 4 would just be adding 21-day returns. With these incremental additions, it may be easier to draw the line between complexity and efficacy.

Concerning the intraday trading data, we were restricted by only being able to view and download a maximum of the last three months of trading history on Refinitiv. Yahoo! Finance and other market services did not seem to have any intraday data at all, so we worked with the limited but available option. 5-minute intervals were used in order to make up for the loss of data points from this shortened range. Future projects employing intraday data would benefit from a larger date range, resulting in a greater number of data points and thereby more accurate performance.

Lastly, although the purpose of this project was to analyze the effect of increasing model complexity on performance (relative to the market) rather than to create a “good” model, it would be interesting to examine how these models would perform when tested on alternative equity indices or forex conversions.

A Software Requirements

As previously mentioned, the project was developed in Python. In order to successfully run the model, Python version 3.5 (or greater) needs to be installed. Once this is done, the following open-source modules will need to be installed:

TensorFlow is a library for numerical computation and large-scale machine learning. The library allows access to machine learning and deep learning models and algorithms in an easy to access format. TensorFlow was used to create our DDQN network [28].

Numpy allows the creation of n-dimensional arrays. Vectorization and indexing make this library fast and versatile compared to the standard Python library arrays [29].

Gym enables the development of RL algorithms. It allows the learning algorithm and the environment to communicate through a standard API. We use the library to facilitate a number of the agent's activities [5].

Pandas is a data analysis and manipulation tool. Like the numpy library, it is very fast and flexible [30]. In our code, we use pandas for the creation and manipulation of data frames.

Matplotlib allows the efficient creation of graphs and visualizations [31]. We combine the data frames made with the pandas library to make data visualizations, discussed further in the Results section.

References

[1] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, “The arcade learning environment: An evaluation platform for general agents,” *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[2] J. E. Moody, M. Saffell, Y. Liao, and L. Wu, “Reinforcement learning for trading systems and portfolios.,” in *KDD*, pp. 279–283, 1998.

[3] X. Wu, H. Chen, J. Wang, L. Troiano, V. Loia, and H. Fujita, “Adaptive stock trading strategies with deep reinforcement learning methods,” *Information Sciences*, vol. 538, pp. 142–158, 2020.

[4] D. Vezeris, I. Karkanis, and T. Kyrgos, “Adturtle: An advanced turtle trading system,” *Journal of Risk and Financial Management*, vol. 12, no. 2, p. 96, 2019.

[5] S. Jansen, *Machine Learning for Algorithmic Trading.* Packt, 2 ed., 2020.

[6] B. Lukanima, *Equity Index*, pp. 1956–1959. Dordrecht: Springer Netherlands, 2014.

[7] M. Hall, “Who or what is dow jones?,” Mar 2022.

[8] E. Chang, “How to choose between etfs and mutual funds,” Mar 2020.

[9] S. Sornmayura, “Robust forex trading with deep q network (dqn),” *ABAC Journal*, vol. 39, no. 1, 2019.

[10] B. Cattlin, “What moves g10 currency?,” May 2021.

[11] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, pp. 279–292, May 1992.

[12] B. Jang, M. Kim, G. Harerimana, and J. W. Kim, “Q-learning algorithms: a comprehensive classification and applications,” *IEEE Access*, vol. 7, pp. 133653–133667, 2019.

[13] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[14] B. Hambly, R. Xu, and H. Yang, "Recent advances in reinforcement learning in finance," *arXiv preprint arXiv:2112.04553*, 2021.

[15] F. Tan, P. Yan, and X. Guan, "Deep reinforcement learning: from q-learning to deep q-learning," in *International Conference on Neural Information Processing*, pp. 475–483, Springer, 2017.

[16] A. Merćep, L. Mrčela, M. Birov, and Z. Kostanjčar, "Deep neural networks for behavioral credit rating," *Entropy*, vol. 23, no. 1, p. 27, 2021.

[17] S. Carta, A. Ferreira, A. S. Podda, D. R. Recupero, and A. Sanna, "Multi-dqn: An ensemble of deep q-learning agents for stock market forecasting," *Expert systems with applications*, vol. 164, p. 113820, 2021.

[18] M. Ramicic and A. Bonarini, "Attention-based experience replay in deep q-learning," in *Proceedings of the 9th International Conference on Machine Learning and Computing*, pp. 476–481, 2017.

[19] W. Yuan, Y. Li, H. Zhuang, C. Wang, and M. Yang, "Prioritized experience replay-based deep q learning: Multiple-reward architecture for highway driving decision making," *IEEE Robotics & Automation Magazine*, vol. 28, no. 4, pp. 21–31, 2021.

[20] P. Lv, X. Wang, Y. Cheng, and Z. Duan, "Stochastic double deep q-network," *IEEE Access*, vol. 7, pp. 79446–79454, 2019.

[21] Y. Deng, F. Bao, Y. Kong, Z. Ren, and Q. Dai, "Deep direct reinforcement learning for financial signal representation and trading," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 3, pp. 653–664, 2016.

[22] J. Carapuço, R. Neves, and N. Horta, "Reinforcement learning applied to forex trading," *Applied Soft Computing*, vol. 73, pp. 783–794, 2018.

[23] "Yahoo finance - stock market live, quotes, business finance news," Mar 2022.

[24] "Financial technology, data, and expertise," Mar 2022.

[25] JetBrains, "Jupyter notebook support — pycharm," 2022.

[26] A. Ingargiola and contributors, “What is the jupyter notebook? — jupyter/ipython notebook quick start guide,” 2015.

[27] Google, “Google colab - what is google colab?,” 2022.

[28] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015. Software available from tensorflow.org.

[29] NumPy, “Numpy,” 2022.

[30] T. pandas development team, “pandas - python data analysis library,” Feb 2020.

[31] J. D. Hunter, “Matplotlib: A 2d graphics environment,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.