University of Twente

"Comptant plus de 9 000 étudiants toutes filières confondues, l'université est réputée comme étant une des meilleures universités des Pays-Bas, et figure en très bonne place dans les classements mondiaux, notamment pour son cursus en informatique (62e selon QS Ranking) et ingénierie (82e)."

Faculty of Electrical Engineering, Mathematics and Computer Science

Uncovering the Potential of Deep Learning in Algorithmic Trading

A Critique of: Deep reinforcement learning stock market trading, utilizing a CNN with candlestick images

Applied Mathematics - Master Data Science

Author:
Frits S. Tuininga
f.s.tuininga@student.utwente.nl

Supervisors:
Prof. Dr. Wouter M. Koolen
w.m.koolen@utwente.nl

Dr.Ir. Wouter van Heeswijk
w.j.a.vanheeswijk@utwente.nl

June 14, 2023

Abstract

Our study found that a specific form of technical analysis in combination with deep learning cannot be utilized to outperform the market. According to the research paper titled *Deep Reinforcement Learning Stock Market Trading, Utilizing a CNN with Candlestick Images* (Brim et al. 2022) , incorporating a Convolutional Neural Network (CNN) into a Double Deep Q-learning Network (DDQN) can help predict the best position to take on stock price movements. The study focuses on the 30 largest stocks in the S&P500 index. The approach employed in the study involves training and testing individual models for each stock. The performance of each model is measured in terms of geometric return, and the average performance across all stocks is calculated at the conclusion of the testing period. The study reveals that the proposed strategy of using a CNN in a DDQN framework outperforms the S&P500 in the period January 1st, 2020 to June 30th, 2020, by identifying advantageous positions using candlestick images. The utilization of visual representations, such as candlestick images, for making investment decisions is commonly referred to as technical analysis. According to prevailing financial theory (Hull 2003) , technical analysis is considered incapable of consistently generating above average returns. However, the study conducted by Brim et al. presents potentially contradictory findings. Our study aims to investigate and determine whether deep learning in combination with technical analysis can consistently outperform the market or that prevailing beliefs regarding the efficacy of technical analysis hold true. We aim to contribute to the ongoing discourse surrounding the effectiveness of technical analysis in the field of financial decision-making.

In our opinion, several modifications are warranted in the research methodology employed by Brim et al. Most importantly, their study did not examine whether their model consistently converged to the same solution during training regardless of the initial weight parameter settings. This raises an important question: Are the findings reported by Brim et al. the result of systematic factors or mere chance? To address this issue, we conducted a similar study with modifications to the research methodology. Our modifications encompassed training and testing exclusively on S&P500 data, evaluating a DDQN and a Prioritized Replay Dueling DDQN (PRD-DDQN), training the two models with 100 distinct initial weight parameters to assess convergence, incorporating a non-crisis test set to evaluate model performance in non-crisis periods (Brim et al. focused solely on a crisis period), and evaluating five strategies: daily long strategy in S&P500, trained DDQN, trained PRD-DDQN, untrained DDQN, and untrained PRD-DDQN. The inclusion of untrained models allows us to discern any significant differences in behavior compared to their trained counterparts. The PRD-DDQN model, which is an enhanced version of the DDQN, was included in our analysis with the specific aim of investigating its potential for superior performance compared to the DDQN. Our findings indicate that the trained models did not converge towards a similar solution. Moreover, the average geometric return achieved by each model type was found to be close to 0%. Notably, while the Prioritized Replay Dueling Double Deep Q-learning Network (PRD-DDQN) model demonstrated the ability to establish associations between features (input) and targets (desired output) during the training phase, it exhibited poor generalization and performance during testing. As a result, we were unable to obtain evidence that supports the claim of Brim et al. that a DDQN with an incorporated CNN can outperform the market.

Acknowledgements

I would like to express my gratitude to my academic supervisors, Prof. Dr. Wouter Koolen and Dr. Ir. Wouter van Heeswijk, whose invaluable guidance and expertise have been instrumental in the successful completion of this master thesis. Their unwavering support and technical insights on model creation were invaluable throughout this research process. In addition, I would like to express my gratitude to my corporate supervisor, Jaap Stolp, for his important guidance and insightful discussions throughout the course of this thesis. His input provided valuable direction and proved to be instrumental in the successful completion of this work.

I am also grateful to Bas van Tintelen and Ruben Lucas for their significant contribution to discussions on selecting the best course of action and selecting the most suitable artificial intelligence models in order to achieve the best results. Their insights and involvement were paramount to the quality of this thesis.

I would like to thank Annelies and Ype Tuininga for their unwavering support throughout my academic journey and for the discussions on linguistic details, which helped to enhance the clarity of this thesis.

Finally, I would like to extend my sincere appreciation to Mart Nijkamp for his unwavering support, encouragement, and for providing a conducive environment for a relaxed and productive working atmosphere. His insightful discussions on models and theses were instrumental in shaping the direction of this research.

Contents

List of Figures

List of Tables

List of Algorithms

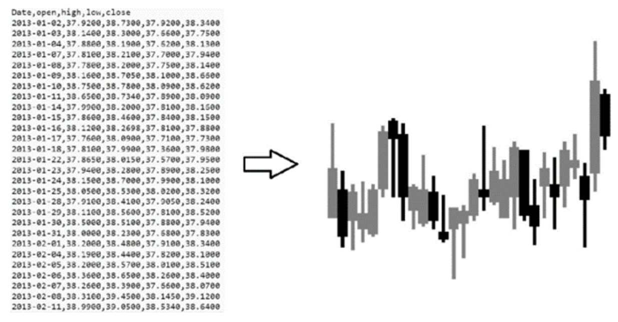
1 Introduction

The field of Deep Learning is currently undergoing rapid growth and development, as evidenced by its success in breakthrough applications such as *MuZero* (Schrittwieser et al. 2020) and *ChatGPT* (Downling et al. 2023; Aydin et al. 2022). In the financial sector, deep learning has garnered considerable attention due to its potential to improve the decision-making processes (Huang et al. 2020) . An example of this is presented in the paper *Deep Reinforcement Learning Stock Market Trading, Utilizing a CNN with Candlestick Images* (Brim et al. 2022) . In this paper, a new approach to stock market trading is introduced which combines deep Reinforcement Learning with a Convolutional Neural Network (CNN). The approach involves transforming financial time-series data into images, and use deep learning models to generate predictions regarding the best position to take given patterns within the image. This method is used to develop a trading policy that the authors claim outperformed the market with 17.2% during the start of the corona crisis (January 1st, 2020 to June 30th, 2020).

Utilizing images for investment purposes, as expounded by Brim et al., is a form of technical analysis. This technique involves the use of statistical patterns in historical data to predict future price movements. However, standard economic theory states that technical analysis historically failed to consistently generate above-average returns (Hull 2003) . In addition to the critiques put forward by Hull, there are additional points of improvement from the perspective of deep learning. If financial data is transformed into an image format, this transformation does not provide any additional information beyond what was already contained in the raw data. Furthermore, such a transformation may even introduce extraneous data, which could slow down the learning process and ultimately fail to enhance the predictive abilities of the model. Given these factors, it is reasonable to approach the research conducted by Brim et al. with a critical mindset, taking into account both the perspectives of financial analysis and deep learning.

The research of Brim et al., however, is interesting from an academic perspective. Their findings indicate the identification of a new area in which deep learning can excel. Conversely, if Brim et al. are mistaken, the standpoint that technical analysis does not work would be confirmed. For investors, this holds additional significance as it could lead to the discovery of a new way of trading or provide further support to investors who subscribe to the model as discussed by mathematical finance (Hull 2003) .

The aim of our study is to evaluate whether a CNN within a Deep Learning framework can use images to predict the best position to take regarding asset price movements. To achieve this, we conduct similar research as described by Brim et al. with a few adjustments to the research methodology including the introduction of an improved model. Before these adjustments are elaborated upon, it is important to discuss the research of Brim et al. in more detail. Their study combines Deep Reinforcement Learning with a CNN in which the latter is employed for image recognition. Reinforcement Learning is a deep learning field that involves an agent interacting with its environment to determine a good policy. In this context, the paper (Brim et al. 2022) utilizes candlestick charts as input for the CNN. Candlesticks represent price movements of an asset and display four price points for each time period: open, close, high and low price (Lee et al. 1999) . If the closing price is higher than the opening price during a given time period, candlestick color indicates a profitable period. Conversely, candlestick color indicates a loss when opening price is higher than closing price. The paper (Brim et al., 2022) employs a color scheme of gray and black to represent profit and loss respectively. The body of each candle represents the difference between the closing and opening price, whereas the wicks represent the high and low prices. By analyzing a time series of candlesticks in a single chart, a trader hopes to observe stock market trends and make informed investment decisions. An example of a candlestick chart (Brim et al. 2022) is given in Figure 1.



```
Date,open,high,low,close
2013-01-02,37.9200,38.7300,37.9200,38.3400
2013-01-03,38.1400,38.3000,37.6600,37.7500
2013-01-04,37.8800,38.1900,37.6200,38.1300
2013-01-07,37.8100,38.2100,37.7000,37.9400
2013-01-08,37.7800,38.2000,37.7500,38.1400
2013-01-09,38.1600,38.7050,38.1000,38.6600
2013-01-10,38.7500,38.7800,38.0900,38.6200
2013-01-11,38.6500,38.7340,37.8900,38.0900
2013-01-14,37.9900,38.2000,37.8100,38.1600
2013-01-15,37.8600,38.4600,37.8400,38.1500
2013-01-16,38.1200,38.2698,37.8100,37.8800
2013-01-17,37.7600,38.0900,37.7100,37.7300
2013-01-18,37.8100,37.9900,37.3600,37.9800
2013-01-22,37.8650,38.0150,37.5700,37.9500
2013-01-23,37.9400,38.2800,37.8900,38.2500
2013-01-24,38.1500,38.7000,37.9900,38.1000
2013-01-25,38.0500,38.5300,38.0200,38.3200
2013-01-28,37.9100,38.4100,37.9050,38.2400
2013-01-29,38.1100,38.5600,37.8100,38.5200
2013-01-30,38.5000,38.5100,37.8800,37.9400
2013-01-31,38.0000,38.2300,37.6800,37.8300
2013-02-01,38.2000,38.4800,37.9100,38.3400
2013-02-04,38.1900,38.4400,37.8200,38.1000
2013-02-05,38.2000,38.5700,38.0100,38.5100
2013-02-06,38.3600,38.6500,38.2600,38.4000
2013-02-07,38.2600,38.3900,37.5600,38.0700
2013-02-08,38.3100,39.4500,38.1450,39.1200
2013-02-11,38.9900,39.0500,38.5340,38.6400
```

Figure 1: Daily high, low, open, and close stock prices are converted to a candlestick image. Gray candles indicate upward price movement, and black candles indicate a downward price movement. (Brim et al. 2022)

Brim et al. transformed asset price data into candlestick charts, which were utilized as inputs to a CNN. The CNN is applied within the framework of a deep Reinforcement Learning model, which enables the identification of the most appropriate action to take based on the candlestick charts. The set of possible actions, denoted as A, is comprised of three distinct elements: *long position*, *no position*, and *short position*, which are represented by 1, 0, and -1 respectively. Actions are taken on a daily basis and the duration of asset holdings is limited to a single day. Specifically, if a long position is taken at the beginning of the day, then the asset is sold at the end of the day. Similarly, if a short position is taken at the beginning of the day, asset will be repurchased at the end of the day. In addition, trading is being restricted to a single asset and trading costs are not performed. Candlestick charts of historical asset price movements are utilized as input for a CNN, which generate a position to be taken at the commencement of the present day. At the end of the day, the position is terminated and a profit (positive, negative, or zero) is realized. Hence, each trading day starts with a neutral position. The underlying premise of this approach is that future asset price movements can be predicted by analyzing past asset price movements.

The model utilized (Brim et al. 2022) achieved impressive cumulative rewards and outperformed the Standard and Poor's 500 index (S&P500 index) by 17.2% during the start of the corona crisis (January 1st, 2020 - June 30th, 2020). The S&P500 index is a stock market index that measures the performance of the 500 largest publicly traded companies in the United States. In regard to the research methodology employed by Brim et al., our study contends that a sensitivity analysis should be conducted to assess the influence of initial weight parameters on the obtained solution. The extent to which the described model can identify a meaningful relationship between features and target is of utmost importance. In cases where such a relationship exists, the model's convergence to a consistent solution remains unaffected by the initial weight parameters. Conversely, when the model struggles to establish this relationship, its reliance on the initial weight parameters becomes more pronounced, resulting in a broader range of disparate solutions. If the described model is able to identify a relationship between features and target, then regardless of initial weight parameters the model would converge approximately to the same solution. In contrast, if the model is unable to identify a relationship between features and target, then the model becomes heavily reliant on initial weight parameters resulting in a wide range of different solutions. By incorporating a sensitivity analysis, a more comprehensive evaluation of the robustness and reliability of the findings of Brim et al. can be achieved. A CNN makes predictions based on its weight parameters. In the beginning, the CNN is initialized with random weight parameters and the weight parameters are adjusted during training in order to find a suitable solution. If the CNN fails to converge towards a suitable solution, and is unable to establish a relationship between the input and output, the initial weight parameters have a substantial impact on the final performance of the model. To enhance the validity of their findings, Brim et al. have trained and tested multiple Reinforcement Learning models with varying weight initialization parameters, thereby ensuring performance is either due to randomness or due to relations found between features and target. For this reason, our research incorporates a sensitivity analysis. In addition to the sensitivity analysis, we apply three adjustments to the research methodology of Brim et al.:

1. **Non-crisis test period:** It is possible that the model may demonstrate exceptional performance solely during periods of crisis due to large price swings, while underperforming during periods of relative stability. In order

to gain a more comprehensive understanding of model performance, it could be beneficial to evaluate model performance during periods of non-crisis as well.

2. **Comparing performances:** The study conducted by Brim et al. entailed training and testing their model on the 30 largest stocks in the S&P500 index, followed by comparing its performance against the overall S&P500 index performance during the early stages of the corona crisis (January 1st, 2020 - June 30th, 2020). While it is customary to benchmark investment strategies against a relevant index, it is more informative to assess model performance relative to the average asset returns on which the model was trained. During the start of the corona crisis, the 30 largest stocks in the S&P500 index exhibited a decline of -6.60%, while the overall S&P500 index experienced a decline of -4.04%. These results suggest that relative model performance might be even better than initially reported. To gain a better understanding of model performance, training and testing can be applied on S&P500 index data alone. To implement this approach, one would train and test a model on an Exchange Traded Fund (ETF) designed to emulate the price movements of the S&P500 index, rather than relying on training the model on each of the 500 stocks individually. It is worth noting that the S&P500 index is widely considered to be a benchmark, and as such, Brim's comparison is reasonable. Nevertheless, to obtain a more precise assessment of model performance, the adjustments mentioned earlier may be helpful.

3. **Improved model:** Brim et al. utilized a Reinforcement Learning model called a Double Deep Q-learning Network (DDQN) (Van Hasselt et al. 2016) . This model can be significantly improved by applying prioritized sampling methods (Schaul et al. 2015) and by separating the DDQN into two streams - one for estimating the state value and one for estimating the action value (Wang et al. 2016) . Hence, Brim's original model could potentially be improved through the integration of prioritized sampling techniques in conjunction with a dueling architecture. We refer to this model as the Prioritized Replay Dueling DDQN (PRD-DDQN).

In our study, we integrate the aforementioned sensitivity analysis and three areas of improvement into our experimental design, and subsequently evaluate whether similar outcomes to those reported by Brim et al. can be achieved. If this is the case, it would suggest that technical analysis has the potential to generate above-average returns consistently, in the context described by Brim et al. that is. To execute the sensitivity analysis, we train 100 models of the same type (either DDQN or PRD-DDQN) with varying initial weight parameter configurations. This allows us to evaluate the impact of initial parameter settings on the predictive capabilities of a given model type. A comprehensive account of the executed experiments is presented in the *Methodology* section. To enable a comparative analysis of our findings with those of Brim et al., Equation 1 can be used.

$$R_i = \Pi_{t=1}^{T}(1 + a_{i,t}y_t) - 1 \tag{1}$$

In this equation, T denotes the final day of testing and y_t is the daily percentage change in stock price from day $t-1$ to day t. The resulting cumulative return, R_i, is the geometric return on day T for model i. In order to gauge the overall performance per model type, we calculate the Average Geometric Return, $\bar{R}$, by taking the average of the geometric returns of all models on day T, as indicated in Equation 2, in which 100 denotes the total number of models tested.

$$\bar{R} = \frac{1}{100}\sum_{i=1}^{100} R_i \tag{2}$$

We selected the Average Geometric Return (Equation 2) as the performance metric for the experiments, which addresses the following research question:

To what extent can technical analysis be employed to achieve sustained above-average returns when converting stock data into candlestick charts and *subsequently utilizing it as input for a Reinforcement Learning model to forecast the percentage price movement of a stock from one trading day to the next?*

The original study by Brim et al. could be enhanced by incorporating a sensitivity analysis on parameter initialization, using a non-crisis test set, exclusively utilizing S&P500 index data for comparing performance, and improving the DDQN model by implementing Prioritized sampling and a Dueling architecture. The primary objective of our research is to contribute to the scientific body of knowledge by evaluating whether the specific form of technical analysis as described by Brim et al. can consistently generate above-average returns. Our research aims to evaluate the conclusion drawn in the original study (Brim et al. 2022) , which claims that a DDQN model outperforms the S&P500 index when tested during the start of the corona crisis (January 1st, 2020 to June 30th, 2020) and to evaluate whether a DDQN model is capable of outperforming the S&P500 index in a non-crisis period. It should be noted that in this context, outperforming the S&P500 index refers to generating returns superior to those obtained through the strategy of taking a long position in the S&P500 index on a daily basis and accumulating the resulting returns over time. The proposed methodology aims to extend and improve upon the work of Brim et al., thereby evaluating the feasibility of an alternative model that may provide superior results compared to the original model. If successful, this alternative model may have practical implications and could potentially be used in real-world algorithmic trading. The research may demonstrate the potential of deep learning methods in finance, encouraging further research in this area to explore untapped potential in the financial sector.

The remainder of work is structured as follows: in the *Theoretical Framework* section, a comprehensive literature search is conducted to identify related works and relevant studies, which are then discussed in detail. Furthermore, this section addresses the gaps in existing literature that we aim to fill. The *Methodology* section presents a detailed argumentation and layout of all experiments. The *Results* section presents a clear and concise overview of the observations made during research, using appropriate visual aids and statistical analyses where necessary. The *Discussion* section presents an in-depth interpretation of the results, along with a discussion of potential opportunities for future research. Furthermore, the Discussion discusses the possibility of replacing the CNN by other deep learning models. Finally, the *Conclusion* section summarizes the main findings of the study and draws appropriate conclusions.

2 Theoretical Framework

In this section, we first describe the various ways in which deep learning is applied within the financial sector in general. Subsequently, we focus on a specific deep learning algorithm known as a Double Deep Q-learning Network (DDQN) and its applications within finance. The aim is to demonstrate the ways in which deep learning in general and specifically a DDQN are relevant to the financial sector. Next, we examine the article *Deep Reinforcement Learning Stock Market Trading, Utilizing a CNN with Candlestick Images* (Brim et al. 2022) . We describe the research, its relevance to the literature, potential areas for improvement, and the value that such improvements would add to the existing body of knowledge. The mathematics behind the techniques used is discussed in the *Methodology* section.

2.1 Applications of Deep Learning in Finance

Deep learning is a subset of machine learning which revolves around deep neural networks. These neural networks are designed to process data and can learn to identify patterns on their own, thereby enabling them to provide accurate predictions (Goodfellow et al. 2016) . Their widespread applicability is seen in domains such as image recognition, natural language processing, and speech recognition. The article entitled *Sequence classification for credit-card fraud detection* (Jurgovsky et al. 2018) , employs a Long Short-Term Memory (LSTM) model to detect credit card fraud. The LSTM is a deep learning model which has the ability to analyze time series data. The dataset used in

this research is composed of real-world credit card transactions, which are classified as either fraudulent or legitimate. The study evaluated model performance on two types of transactions: online (e-commerce) and offline (point-of-sale or face-to-face) transactions. LSTM and Random Forest models were applied to the dataset to predict fraudulent transactions. The findings suggest that applying an LSTM greatly improves the accuracy of fraud detection in comparison to Random Forest for offline transactions. However, for online transactions, utilizing an LSTM does not enhance the predictive power of the model beyond that of Random Forest. Overall, the study contributes to the field of fraud detection by highlighting the importance of transaction history in identifying fraudulent credit card transactions, and emphasizes the potential of the LSTM in this area.
The paper titled *A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem* (Jiang et al. 2017) presents an approach to portfolio management by utilizing deep reinforcement learning. Reinforcement learning involves the iterative process of trial-and-error where the model interacts with an environment to learn a strategy that maximizes the accumulated reward. The proposed model employs a deep neural network to determine the optimal allocation of capital among 12 different cryptocurrencies in a portfolio. The objective is to compare the performance of the deep reinforcement learning-based model with traditional portfolio management techniques in terms of profitability. The results of the study indicate that the proposed model outperforms all surveyed traditional portfolio-selection methods in terms of profitability. This demonstrates the efficacy of deep reinforcement learning in finance and highlights the potential of this approach for optimizing portfolio management strategies.

2.2 Double Deep Q-learning Network

A specific application of deep reinforcement learning is the Double Deep Q-learning Network (DDQN) (Van Hasselt et al. 2016) . The DDQN algorithm is rooted in Q-learning and involves an agent that interacts with an environment. The environment can be seen as a video game and the agent can be seen as a player. The current situation in which the agent finds itself is called the state. At each state, the agent selects an action, and the environment generates a reward and a new state based on this action. Q-learning iteratively approximates a value assigned to each state-action pair to determine the best actions to take in each state. The value assigned to each state-action pair is called the Q-value and is determined by the expected cumulative reward. The set of the best actions given the states is called the policy. As the agent interacts with the environment, it learns from its previous experiences and adjusts its behavior to maximize its long-term reward. The Q-learning algorithm is guaranteed to converge to an optimal solution (Watkins et al. 1992) .
Q-learning becomes computationally expensive when the state and/or action space becomes too large, rendering the algorithm impractical. The Q-learning algorithm involves the storage of Q-values in a tabular format, with each individual state-action pair assigned a corresponding entry in the table. As the state or action space grows in size, the table storing the Q-values expands as well, ultimately resulting in a computational burden that renders it impractical for use. The DDQN overcomes the issue of a large state space by utilizing neural networks. The DDQN algorithm employs neural networks to approximate Q-values and generalize over states. The DDQN algorithm has been effectively applied in various financial applications, such as portfolio optimization and algorithmic trading. The paper *A deep Q-learning portfolio management framework for the cryptocurrency market* (Lucarelli et al. 2020)
presents an approach of using DDQNs to manage portfolios. In the context of finance, a portfolio refers to a collection of securities, such as stocks, held by an individual or a computer program. The objective of portfolio management is to allocate capital to investments that are anticipated to experience the greatest appreciation in value. The paper proposes a framework that integrates DDQNs with portfolio management. The framework is composed of two key components: (i) a set of local DDQN agents that are responsible for trading individual assets in the portfolio and (ii) a global agent that assigns rewards to each local agent based on a common reward function (i.e. global reward). While the local DDQN agents are tasked with determining an effective trading strat-

egy for a particular asset, the global DDQN agent assumes the responsibility of optimally allocating capital across all local DDQN agents to enhance returns while minimizing the potential risk of incurring significant losses. This Q-learning portfolio management approach is evaluated on a portfolio comprised of four popular cryptocurrencies: Bitcoin, Litecoin, Ethereum, and Ripple. The performance of the deep Q-learning agent is compared against several benchmark strategies (such as buy-and-hold strategy and a mean-variance optimization) to assess its efficacy in generating returns on investment and managing risk. The results of the analysis demonstrate that the deep Q-learning agent outperforms the benchmark strategies in both return on investment and risk-adjusted returns, thus providing a useful approach for financial portfolio management utilizing DDQNs for trading cryptocurrencies.
In addition, DDQNs have been utilized in the field of algorithmic trading to improve the performance of investment strategies. The paper entitled *Hybrid Deep Reinforcement Learning for Pairs Trading* (Cartea et al. 2021) investigates the use of DDQNs for pairs trading. Pairs trading is a statistical arbitrage strategy that exploits the short-term price divergences between two assets that have historically moved together. In the stock market, pairs traders simultaneously open a short position in an overvalued stock and a long position in an undervalued stock. Once the prices of the two stocks converge, the positions are closed by taking opposite positions, which results in a profit.
To evaluate the effectiveness of the proposed algorithm, the authors chose to test it on twenty stock pairs. The proposed algorithm utilizes one DDQN for trading actions, specifically for determining whether to take a long or short position, while the another DDQN was used to determine stop-loss boundaries, which dictate at which price stock positions will be closed. The authors compared the performance of the proposed algorithm with state-of-the-art reinforcement learning methods for pairs trading.
The results of the experiment showed that the proposed algorithm achieved an average return rate of 82.4%, which is 25.7% higher than that of the second-best method. Furthermore, the proposed algorithm yielded significantly positive return rates for all stock pairs. Therefore, the use of DDQNs in pairs trading can lead to more effective investment strategies that take advantage of short-term price divergences between closely related stocks.

2.3 DDQN for Stock Trading

In the article titled *Deep Reinforcement Learning Stock Market Trading, Utilizing a CNN with Candlestick Images* (Brim et al. 2022) , a Convolutional Neural Network (CNN) is employed in a DDQN framework. The approach involves transforming historical stock data into images, and
subsequently using a DDQN in conjunction with a CNN to identify meaningful patterns in these images, and using this information to predict the best action to take with regards to price movements from day t to day $t+1$. This method is used to develop a trading policy that the authors claim could outperform the market during the corona crisis (January 1st, 2020 to June 30th, 2020).

To confirm that Brim et al. identified a unique area of research in the application of a DDQN in conjunction with candlestick images for stock trading, a citation search we conducted. This revealed no prior relevant research was conducted in this particular domain.
Despite the impressive results achieved by Brim et al., there is a possibility that their findings are a product of chance. Notably, the absence of a sensitivity analysis to assess the impact of initial weight parameters makes it unclear whether the DDQN successfully identified meaningful relationships between features and the target. Hence, a sensitivity analysis on parameter initialization could be implemented to boost the credibility of results. After all, without a sensitivity analysis the findings could be the result of favorable initial parameter settings. Furthermore, it is possible that the model may demonstrate exceptional performance solely during periods of crisis. Hence, the model must be tested during periods of non-crises as well. In addition, the model is trained on a sub-set of S&P500 data while being compared to the entire S&P500 index. To obtain a more comprehensive understanding of model performance, the model can be exclusively trained and tested on the S&P500 index data set. Lastly, the DDQN

can be improved through the integration of prioritized sampling techniques in conjunction with a dueling architecture.
First, we will investigate the notion of prioritized sampling. In a standard DDQN, an agent actively interacts with its environment and captures four essential components of the exchange. These components are the current state, the action taken, the immediate reward, and the next state achieved. The agent then stores this information in its Replay Buffer for later use. Following the interaction with the environment, the agent randomly samples pieces of information from Replay Buffer. Random sampling allows the agent to learn from previous experiences, however, not all interactions are equally valuable for the agent's learning process regarding the identification of relationships between features and target. When an agent made a correct prediction regarding which action to take on a specific day, then that interaction may not offer significant learning opportunities, as there are no significant updates required to change the agents behavior in that situation. In contrast, when an agent makes incorrect predictions, the specific interaction offers valuable learning opportunities, as the agent can update its knowledge and improve its predictions. An incorrect prediction is characterized by the model taking a long position while the S&P500 index declines, a short position while the index increases or taking no position. In theory, the act of taking no position can be considered 'correct' if there is no change in the price of the S&P500 index from day t to day $t + 1$. However, in practice, this scenario is exceedingly rare. Notably, taking no position is preferable to taking a long position while the S&P500 index decreases or taking a short position while the index increases. If the model is uncertain about the optimal action to take, it may choose to take no position and minimize the magnitude of an erroneous prediction. In the paper *Prioritized experience replay* (Schaul et al. 2015) , the authors suggest ranking elements in Replay Buffer based on their prediction accuracy. When the agent interacts with its environment, each element in Replay Buffer is assigned a probability based on the quality of its prediction, with poorly predicted elements receiving higher probabilities and well-predicted elements receiving lower probabilities. Subsequently, when the DDQN finishes its interaction with the environment and begins sampling elements from Replay Buffer, elements with higher probabilities are more likely to be sampled than those with lower probabilities. Consequently, the model is updated mostly based on its worst predictions, where it stands to gain the most. The authors applied a DQN with Prioritized Sampling to Atari games and achieved superior results when compared to the standard DQN with uniform replay.
The introduction of Prioritized Sampling significantly enhances the performance of a DQN and, by extension, a DDQN (Schaul et al. 2015) . As a result, Prioritized Sampling has been acknowledged as a valuable extension to the conventional DDQN approach.
Let us now discuss the Dueling DDQN (Wang et al. 2016) . In comparison to a conventional DDQN, a Dueling DDQN evaluates the value of individual states and actions as opposed to merely assessing state-action pairs.

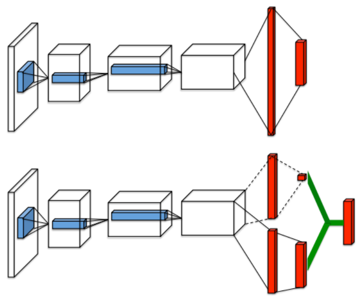


Figure 2: Illustration of a Dueling Architecture, which splits the final linear layer into two streams: one for estimating the action value and the other for estimating the state value (Wang et al. 2016)

Figure 2 shows the dueling architecture. The top architecture represents a conventional DDQN, whereas the bottom architecture represents a dueling DDQN. Note that the dueling architecture splits the linear layers in two streams of which the upper layers are dedicated to estimating individual state values, and the lower layers dedicated to estimating individual action values. The aggregation of state and action values, as described by Equation 3, is illustrated by the green lines.

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right) \quad (3)$$

In here, $V(s)$ and $A(s,a)$ refer to the individual state and action value learned. The results presented by the authors in *Dueling network architectures for deep reinforcement learning* (Wang et al. 2016) exhibit that the Dueling DDQN model surpasses other state-of-the-art models, including the conventional DDQN, in the Atari 2600 domain. Such findings suggest that the Dueling DDQN could potentially outperform a standard DDQN model in other contexts as well. Furthermore, the authors note that the dueling architecture is characterized by a more efficient learning process, as it allows for the independent evaluation of states and actions. As a consequence, the dueling architecture is a valuable extension to the conventional DDQN approach.

2.4 Contribution to Literature

This study contributes to the existing body of knowledge in two ways. First, we made improvements to the experimental setup as proposed by Brim et al. Specifically, our approach involves training and testing the algorithm on the S&P500 index and comparing its performance to the overall performance of the S&P500 index. Second, our modified algorithm is also applied to a non-crisis period, thus augmenting our understanding of its performance during both crisis and non-crisis periods. The implementation of these changes serves to enhance the credibility of the claims made by Brim et al. Secondly, the standard DDQN as described by Brim et al. is adjusted by implementing a Dueling architecture in combination with Prioritized Sampling. This Prioritized Replay Dueling DDQN (PRD-DDQN) is compared with a standard DDQN in terms of Average Geometric Return (see *Introduction*). The aim of this comparison is to ascertain whether the PRD-DDQN outperforms a standard DDQN. If so, the former algorithm can be seen as a viable alternative to the latter in this specific context.

Our study makes two contributions to the scientific body of knowledge. First, the research approach of Brim et al. is improved by conducting a sensitivity analysis, applying the models on non-crisis data, and training and testing on S&P500 index data instead of individual stocks. These improvements contribute to literature by addressing potential limitations in the research of Brim et al. and by providing a more comprehensive understanding of model performance. Second, the standard DDQN, as described by Brim et al., is compared to the PRD-DDQN. The goal is to examine whether or not the PRD-DDQN is able to systematically outperform the standard DDQN in the context as described by Brim et al. This contributes to literature by exploring the effectiveness of the PRD-DDQN in conjunction with candlestick images for stock trading.

3 Methodology

The research methodology comprises two distinct phases. In the first phase, the data is imported and subjected to preprocessing procedures. In the second phase, a series of four experiments, as outlined in Table 1, are carried out for both Brim's original Double Deep Q-learning Network (DDQN) (Brim et al. 2022) and the Prioritized Replay Dueling Double Deep Q-learning Network (PRD-DDQN). The comparative analysis of the outcomes is based on the Average Geometric Return.

Ex.	Validation	Test
1	-	[01/01/2020, 30/06/2020]
2	-	[01/01/2021, 31/12/2021]
3	[01/01/2012, 31/12/2012]	[01/01/2020, 30/06/2020]
4	[01/01/2012, 31/12/2012]	[01/01/2021, 31/12/2021]

Table 1: Summary of Experimental Design: Four experiments were conducted utilizing the identical training set (January 1st, 2013 to December 31, 2019) while employing varying test sets. Experiments 3 and 4 incorporate a validation set.

3.1 Phase 1: Data Transformation

The research commences with the first phase, which follows the approach as described by Brim et al. This phase involves the extraction of Standard and Poor's 500 index (S&P500 index) data from Yahoo Finance. The data is divided into four distinct time periods, which include the training set, test set 1, test set 2, and validation set. The following S&P500 index data is retrieved from Yahoo Finance for the respective time periods:

1. Train set: [01/01/2013, 31/12/2019]
2. Test set 1: [01/01/2020, 30/06/2020]
3. Test set 2: [01/01/2021, 31/12/2021]
4. Validation set: [01/01/2012, 31/12/2012]

The retrieved data contains several variables which are the same for all sets. To be more specific, the variables 'date', 'open', 'high', 'low', 'close', 'adj close', and 'volume' are included. The variables 'open', 'high', 'low', and 'close' refer to the price of the stock at the beginning of the day, the highest price reached during the day, the lowest price reached during the day, and the price at the end of the day, respectively. Notably, 'close' differs from 'adj close' in that the latter accounts for any dividend or capital gain distributions, while the former does not, although both adjust for stock splits. Lastly, 'volume' denotes the number of shares traded during the day. Candlestick charts represent price movements of a stock and display four price points for each time period: open, close, high and low price (Lee et al. 1999) . To generate candlestick charts and daily percentage changes, the 'date', 'adj close', and 'volume' columns are irrelevant and are therefore removed from the data set.

The graphical representation of financial data, known as candlesticks, provides insight into the direction and magnitude of price movements within a given time frame. Figure 3 illustrates the manner in which candlesticks convey both positive and negative price changes. Specifically, when the 'open' price exceeds the 'close' price, the associated stock experienced a negative movement during the designated time-period. Conversely, when the 'close' price exceeds the 'open' price, the corresponding stock underwent a positive movement during that same time-period. Utilizing candlestick patterns as the basis for an investment strategy represents a form of technical analysis, whereby statistical trends in past price movements are analyzed to predict future price movements. However, according to Hull, there is limited evidence that technical analysis can consistently generate above average returns. Instead, the book argues that stock price behavior is more accurately characterized by a stochastic process known as geometric Brownian motion (Hull 2003) , which is expressed in Equation 4.

$$S_t = S_0 \cdot e^{(\mu - \frac{\sigma^2}{2})t + \sigma^2 z(t)} \tag{4}$$

The variables in Equation 4, namely S_t, S_0, μ, σ, and $z(t)$, represent the stock price at time t, the stock price at the start of the period (i.e. $t = 0$), the drift rate, the volatility, and a standard normal random variable, respectively. The standard normal random variable represents the randomness of price movements. Note that asset prices are dependent on each other (i.e. S_{t+1} depends on S_t). Our study assesses the potential of a specific technical analysis method, as described by Brim et al., to consistently generate above-average returns. Our analysis seeks to either support this assertion or provide evidence that challenges Hull's claim.

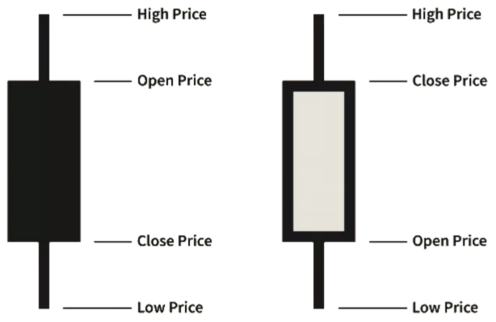
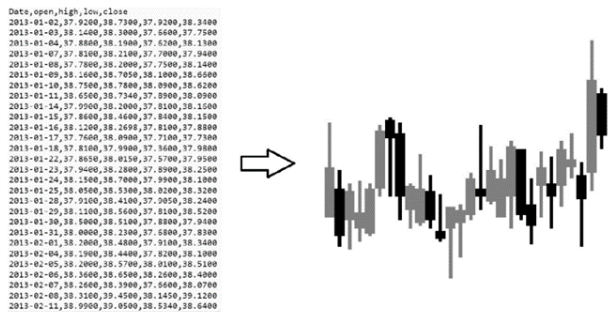


Figure 3: A visual representation of negative and positive candlesticks commonly used in financial analysis. The candlesticks are color-coded, with black and gray representing negative and positive stock movements, respectively. (Investopedia 2021)

Our study seeks to evaluate Hull's assertion regarding the efficacy of technical analysis in producing consistent above-average returns. To do so, we adopt the methodology put forth by Brim et al. for assessing the potential of candlestick charts combined with deep learning to generate such returns. Within the scope of our analysis, trades are limited to a single day, with each candlestick on the chart corresponding to the price variations of the S&P500 index within that one day. In line with the methodology outlined by Brim et al., the study incorporates 28 trading days for the candlestick chart. This implies that the previous 28 trading days are represented by a candlestick plot to determine the optimal position to take in order to maximize profit, given the percentage change in stock prices between the close of today and tomorrow. Consequently, the candlestick charts employed in this study conform to the structure delineated in Figure 4. The candle body is 3 pixels wide, while the wicks are 1 pixel wide. The height reflects price movements, and candle color represents upward and downward movements - gray indicates an up trend, and black denotes a downward trend.



```
Date,open,high,low,close
2013-01-02,37.9200,38.7300,37.9200,38.3400
2013-01-03,38.1400,38.3000,37.6600,37.7500
2013-01-04,37.8800,38.1900,37.6200,38.1300
2013-01-07,37.8100,38.2100,37.7000,37.9400
2013-01-08,37.7800,38.2000,37.7500,38.1400
2013-01-09,38.1600,38.7050,38.1000,38.6600
2013-01-10,38.7500,38.7800,38.0900,38.6200
2013-01-11,38.6500,38.7340,37.8900,38.0900
2013-01-14,37.9900,38.2000,37.8100,38.1600
2013-01-15,37.8600,38.4600,37.8400,38.1500
2013-01-16,38.1200,38.2698,37.8100,37.8800
2013-01-17,37.7600,38.0900,37.7100,37.7300
2013-01-18,37.8100,37.9900,37.3600,37.9800
2013-01-22,37.8650,38.0150,37.5700,37.9500
2013-01-23,37.9400,38.2800,37.8900,38.2500
2013-01-24,38.1500,38.7000,37.9900,38.1000
2013-01-25,38.0500,38.5300,38.0200,38.3200
2013-01-28,37.9100,38.4100,37.9050,38.2400
2013-01-29,38.1100,38.5600,37.8100,38.5200
2013-01-30,38.5000,38.5100,37.8800,37.9400
2013-01-31,38.0000,38.2300,37.6800,37.8300
2013-02-01,38.2000,38.4800,37.9100,38.3400
2013-02-04,38.1900,38.4400,37.8200,38.1000
2013-02-05,38.2000,38.5700,38.0100,38.5100
2013-02-06,38.3600,38.6500,38.2600,38.4000
2013-02-07,38.2600,38.3900,37.5600,38.0700
2013-02-08,38.3100,39.4500,38.1450,39.1200
2013-02-11,38.9900,39.0500,38.5340,38.6400
```

Figure 4: Daily high, low, open, and close stock prices are converted to a candlestick image. Gray candles indicate upward price movement, and black candles indicate a downward price movement. (Brim et al. 2022)

Candlestick charts are utilized as input data for a CNN in order to predict the best action to take given future price movements. The primary objective of the CNN is to analyze patterns between the input data and output data, where the output data represents price movements expressed as a percentage. This output data is calculated as the percentage change in the closing price of the next day in comparison to the current day's closing price. Equation 5 shows this principle, in which 'Close_i', 'Close_{i-1}' and 'y_i' refer to the close price of day i, the closing price of the previous day $(i-1)$, and the percentage change observed on day i, respectively.

$$y_i = \frac{\text{Close}_i - \text{Close}_{i-1}}{\text{Close}_{i-1}} \tag{5}$$

The preference for using percentages over absolute price movements arises from the exponential nature of stock market price trends, rendering the prediction of absolute price movement therefore considerably more challenging. In addition, investors prefer relative returns over absolute returns. In this regard, the approach of Brim is followed in which only percentage changes are considered.

The candlestick charts represent a historical depiction of price movements over a period of 28 days, and the CNN is tasked with interpreting these patterns to generate useful predictions.

The creation of charts involves the transformation of raw data into images. Resulting images exhibit a 3-pixel width for each candlestick body and 1-pixel width for each wick, a configuration that aligns with the study by Brim et al. This approach guarantees that the images remain as compact as possible while still preserving all the pertinent information. It is key to keep images as compact as possible to reduce computation time. As the method relies on the previous 28 trading days, and each candlestick body is 3 pixels wide, the resulting candlestick images are of dimensions $3 * 28 = 84$ pixels, both in height and width. To establish a consistent visual representation, candlesticks are scaled over 84 pixels in the vertical dimension which is in accordance with Brim et al. The relative highest and lowest positions of the chart are determined by the candle with the highest wick and the candle with the lowest wick, respectively. Furthermore, it is worth noting that a distinctive color scheme is employed in this study, wherein every positive candle is depicted using the color code $rgb(105, 105, 105)$, while negative candles are represented by the color code $rgb(0, 0, 0)$ and the background color is represented by $rgb(255, 255, 255)$ as in accordance with Brim et al.

To illustrate the process by example, consider Table 2 which focuses on S&P500 index data retrieved from Yahoo Finance. A candlestick chart is generated using 28 trading days, starting from row 1 up and until row 28. The daily change between rows 28 and 29 is then calculated based on the closing price, using Equation 5: $\frac{(3916.38-3909.88)}{3909.88} \approx 0.17\%$. Subsequently, the next candlestick chart is generated, spanning from row 2 up to 29, and the associated output value is calculated as 0.47%. Candlestick charts are constructed up and until row 30 and the percentage change is calculated up and until row 31. In this example, three candlestick charts with corresponding daily changes are constructed. It is important to observe that every candlestick chart displays a total of 28 candles corresponding to the price fluctuations of the preceding 28-day period.

The process of constructing the train, test 1, test 2, and validation sets follow a similar methodology as illustrated in the preceding example. The train, test 1, test 2, and validation sets comprise a total of 1764, 127, 253, and 250 candlestick charts, each accompanied by their subsequent daily change. The set of candlestick charts employed as the input for a CNN is denoted as $\mathbf{X}$, while the set of associated daily changes expressed in percentages is denoted as $\mathbf{y}$. The code employed to generate the candlestick charts, accompanied by the computation of daily changes, has been presented in Appendix A.

Day	Open	High	Low	Close
1	3,733.27	3,760.20	3,726.88	3,756.07
2	3,764.61	3,769.99	3,662.71	3,700.65
3	3,698.02	3,737.83	3,695.07	3,726.86
4	3,712.20	3,783.04	3,705.34	3,748.14
5	3,764.71	3,811.55	3,764.71	3,803.79
6	3,815.05	3,826.69	3,783.60	3,824.68
7	3,803.14	3,817.86	3,789.02	3,799.61
8	3,801.62	3,810.78	3,776.51	3,801.19
9	3,802.23	3,820.96	3,791.50	3,809.84
10	3,814.98	3,823.60	3,792.86	3,795.54
11	3,788.73	3,788.73	3,749.62	3,768.25
12	3,781.88	3,804.53	3,780.37	3,798.91
13	3,816.22	3,859.75	3,816.22	3,851.85
14	3,857.46	3,861.45	3,845.05	3,853.07
15	3,844.24	3,852.31	3,830.41	3,841.47
16	3,851.68	3,859.23	3,797.16	3,855.36
17	3,862.96	3,870.90	3,847.78	3,849.62
18	3,836.83	3,836.83	3,732.48	3,750.77
19	3,755.75	3,830.50	3,755.75	3,787.38
20	3,778.05	3,778.05	3,694.12	3,714.24
21	3,731.17	3,784.32	3,725.62	3,773.86
22	3,791.84	3,843.09	3,791.84	3,826.31
23	3,840.27	3,847.51	3,816.68	3,830.17
24	3,836.66	3,872.42	3,836.66	3,871.74
25	3,878.30	3,894.56	3,874.93	3,886.83
26	3,892.59	3,915.77	3,892.59	3,915.59
27	3,910.49	3,918.35	3,902.64	3,911.23
28	3,920.78	3,931.50	3,884.94	3,909.88
29	3,916.40	3,925.99	3,890.39	3,916.38
30	3,911.65	3,937.23	3,905.78	3,934.83
31	3,939.61	3,950.43	3,923.85	3,932.59

Table 2: Example S&P500 index Data in USD

3.2 Phase 2: Experiment Setup

The second phase of our study comprises of four experiments as illustrated in Table 1. Prior to the presentation of these experiments, an elaboration on several key concepts is provided, namely Convolutional Neural Networks (CNNs), Double Deep Q-learning Networks (DDQNs), and Prioritized Replay Dueling Double Deep Q-learning Networks (PRD-DDQNs).

3.2.1 Convolutional Neural Network

A Convolutional Neural Network (CNN) is a deep learning model which is used for image recognition. The fundamental building blocks of a CNN include three distinct types of layers: convolutional layers, pooling layers, and linear layers. These layers work together to identify important patterns in the provided image which are then subsequently leveraged to formulate predictions.

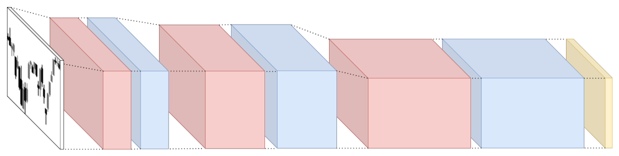


Figure 5: CNN Architecture as described by Brim et al., where the colors red, blue, and yellow indicate the convolutional, pooling, and linear layers, respectively. The first, second, and third convolutional and pooling layer each contain 8, 16, and 32 channels, respectively.

Figure 5 is a visualization of the CNN architecture as described by Brim et al. As shown, the CNN is initially presented with a candlestick image. Subsequently, as the image traverses through the CNN from left to right, it undergoes a series of transformations, progressively modifying the input. Upon reaching the final linear layer, the depth dimension is eliminated, and the CNN produces a set of three Q-values that correspond to each possible action.

3.2.2 Double Deep Q-learning Network

A Double Deep Q-learning Network (DDQN) is a reinforcement learning model utilized to determine good actions in a particular environment, yet before delving into the details of the DDQN, it is imperative to clarify the fundamental principles of Q-learning.

The Q-learning algorithm involves an agent and an environment. The agent can be seen as an investor and the environment can be seen as the stock market. The situation the agent finds itself in is called a state. The agent interacts with its environment by taking actions in a given state. The environment responds by generating a reward and a next state that the agent transitions to. This concept is visualized in Figure 6.

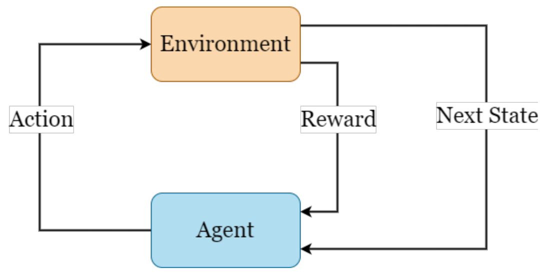


Figure 6: Interaction between Agent and Environment (LeCun et al. 2015)

The rewards accumulate over time and the objective of the agent is to find the best action to take in each state such that the discounted long-term reward is maximized. To achieve this, the Q-learning algorithm assigns a Q-value to each state-action pair and stores them in a Q-table. The Q-value represents the value of a state-action pair. The Q-learning algorithm is guaranteed to converge in the limit to an optimal solution (Watkins et al. 1992) .

In the context of Q-learning, an agent must effectively balance the competing goals of exploration and exploitation. Exploration is characterized by randomly selecting actions in order to avoid becoming trapped in local optima. As an off-policy method, Q-learning learns the value of a policy while following a different policy, in contrast to on-policy methods like SARSA that learn the value of a policy while following that policy. This difference implies that the impact of exploration on the policy discovered in Q-learning is more dampened than in on-policy methods. Exploitation involves choosing the best known action in a given state to optimize outcomes. Successful outcomes in an environment often require a balanced approach that combines both exploration and exploitation. The idea is to visit state-action pairs of high quality on a regular basis, in order to obtain accurate sample estimates, while also allowing for the discovery of even better state-action pairs.

A good approach involves utilizing exploration to discover new possibilities and exploitation to refine and optimize existing knowledge. Numerous methods exist to achieve this balance, but in our study, we employ an exponentially decaying epsilon-greedy approach. Let epsilon represent the probability that the agent takes a random action. An exponentially decaying epsilon (where $\epsilon > 0.5$) implies prioritizing exploration at the beginning of played games, with the epsilon decreasing exponentially as the number of played games progresses. Consequently, the agent increasingly emphasizes exploitation in later games. To ensure continued exploration, a minimum exploration rate of 10% was used. The base number used for exponentially decaying exploration rate was set at $c \approx 0.998$.

Let ϵ denote the exploration rate, where larger values imply a greater probability of exploration and smaller values imply a greater probability of exploitation. Additionally, let c be a real number between 0 and 1 representing the rate at which the model gradually shifts from exploration to exploitation, and let *epoch* represent the agent's current game. Equation 6 illustrates the calculation of epsilon when applying exponential decay.

$$\epsilon = \max\{c^{epoch}, 0.1\}, \qquad c \in \mathbf{R}_{(0,1)} \tag{6}$$

Algorithm 1 shows how the Q-learning algorithm works in detail.The symbol η herein denotes the learning rate, which characterizes the size of the steps taken by the algorithm towards a solution. Additionally, γ represents the discount rate, which serves to diminish the weight of future rewards and place greater emphasis on present rewards.

For all its advantages, Q-learning does have a significant drawback. As the state or action space increases, the computation time required for the Q-learning algorithm to converge increases significantly. As a result, if either the state or action space becomes too large, the Q-learning algorithm does not converge in reasonable time. As previously stated, our study employs candlestick images as input for a CNN. Candlestick images come in many forms, and are viewed as states in our study. As a consequence, the size of the state space becomes very large, rendering the use of simple Q-learning infeasible.

The DDQN algorithm was created to solve the problem of handling a large state space. This algorithm outputs the Q-values, but generalizes over

Algorithm 1 Q-learning (Watkins et al. 1992)

```
1: **Define:** State and action space $S$ and $A$, resp.
2: **Initialize**: Learning rate $\eta$, time step $t$, discount rate $\gamma$, reward matrix $R$, number of epochs $N$, base number used for exponentially decaying exploration rate $c$, terminal state $T = False$, and Q-table with zeros
3: for $epoch$ in $1, ..., N)$ do:
4:     Calculate: $\epsilon = max\{c^{epoch}, 0.1\}$
5:     Initialize current state: $s_0$
6:     while $T == False$ do:
7:         Generate a random number $p \sim U(0, 1)$
8:         if $p < \epsilon$ then:
9:             Explore: Take random action $a_t$
10:        else:
11:            Exploit: Take action $a_t$ associated
12:            with the greatest Q-value for this
13:            state-action pair
14:        end if
15:        Observe next state $s_{t+1}$
16:        Obtain reward $r_t(s_t, a_t)$ from reward
17:        matrix $R$
18:        Update Q-table:
19:            $Q^{\text{new}}(s_t, a_t) \leftarrow Q(s_t, a_t)+$
20:            $\alpha \cdot (r_t(s_t, a_t) + \gamma \cdot \max_a Q(s_{t+1}, a)$
21:            $-Q(s_t, a_t))$
22:        Reset current state: $s_t = s_{t+1}$
23:        $t = t + 1$
24:        if Final state is reached then:
25:            $T = True$
26:        end if
27:    end while
28: end for
```

the state space with the aid of neural networks. By utilizing a neural network the DDQN can calculate Q-values in reasonable time.
The DDQN comprises of two steps, namely the environment and update step. In the environment step, the agent interacts with its environment and collect valuable information. This information consists of the current state, action taken, obtained reward, and next state. This quadruple is stored in the Replay Buffer. After the agent has completed an epoch, the update step begins. During this step, quadruples are randomly sampled from the Replay Buffer, and neural networks are updated based on these experiences and errors in estimated and observed Q-values. This process facilitates learning from past experiences and enhances the agent's ability to make better decisions in future interactions with the environment. To learn from experience, the DDQN employs a loss function and update method. In the context of machine learning and optimization, a loss function evaluates the disparity between a model's predicted output and the desired output, serving as a measure of error or loss incurred by the model's predictions. The objective is to minimize the loss function during training by iteratively adjusting the model's parameters. The DDQN algorithm employs the Mean Squared Error (MSE) as loss function, as depicted in Equation 7.

$$\mathbb{L}(Q_\theta(s_t, a_t) = (Q_\theta(s_t, a_t) - Q^*(s_t, a_t))^2 \tag{7}$$

The update method leverages the loss function to iteratively update the model. In our investigation, we adopt the ADAM update method (Goodfellow et al. 2016) , renowned for its effectiveness in optimizing deep learning models. The DDQN algorithm, is outlined in detail in Algorithm 2.

Algorithm 2 DDQN (Van Hasselt et al. 2016)

```
1: **Define:** State and action space $S$ and $A$, resp.
2: **Initialize**: Learning rate $\eta$, time step $t$, discount rate $\gamma$, reward function $r_t(s_t, a_t)$, number of epochs $N$, base number used for exponentially decaying exploration rate $c$, terminal state $T$, Replay Buffer $D$, batch size $z$, number of updates UPDATES, primary network $Q_\theta$, and target network $Q_{\theta'}$
3: for $epoch$ in $1, ..., N$ do:
4:     Calculate: $\epsilon = max\{c^{epoch}, 0.1\}$
5:     Initialize current state: $s_0$
6:     Set terminal state: $T =$ False
7:     while T $== False$ do:
8:         Generate a uniform number $p \in (0, 1)$
9:         if $p < \epsilon$ then:
10:            Explore: Take random action $a_t$
11:        else:
12:            Exploit: Take action $a_t$ associated
13:            with the greatest Q-value for this
14:            state-action pair
15:        end if
16:        Observe next state: $s_{t+1}$
17:        Obtain reward: $r_t(s_t, a_t)$
18:        Save: $[s_t, a_t, r_t(s_t, a_t), s_{t+1}]$ to $D$
```

Algorithm 2 DDQN (Van Hasselt et al. 2016)

```
19:         if Final state is reached then:
20:             $T = True$
21:         end if
22:         $t = t + 1$
23:     end while
24:     for update in UPDATES do:
25:         Sample: $z$ quadruples from $D$
26:         Calculate:
27:             $Q^*(s_t, a_t) \approx r_t(s_t, a_t) + \gamma \cdot$
28:             $Q_{\theta'}(s_{t+1}, \text{argmax}_{a'} Q_\theta(s_{t+1}, a'))$
29:         Calculate: $\mathbb{L}(Q_\theta(s_t, a_t)$
30:         Apply ADAM to update $Q_\theta(s_t, a_t)$
31:         if (update+1) mod 100 == 0 then:
32:             Update $Q_{\theta'}$ by learning from the
33:             mistakes made
34:         end if
35:     end for
36: end for
```

Note that in the context of our study, rewards are calculated as shown in Equation 8.

$$r_t(s_t, a_t) = y_t \cdot a_t \cdot \text{NRM} \tag{8}$$

In this equation, y_t represents the percentage change in the S&P500 index price from today to tomorrow. The parameter a_t denotes the position taken by the model, which is either a long position ($a_t = 1$), no position ($a_t = 0$), or short position ($a_t = -1$). The Negative Rewards Multiplier (NRM) is an additional parameter introduced by Brim et al. to modify the impact of negative rewards. This constant is a positive integer value that is multiplied exclusively with negative rewards to encourage the model to take no position in uncertain situations. However, Brim et al. did not specify a particular value for the NRM in their research. We have assumed an NRM of 1, which implies that no penalty is applied to negative rewards.

3.2.3 Prioritized Replay Dueling DDQN

In contrast to the standard DDQN, the Prioritized Replay Dueling DDQN (PRD-DDQN) applies prioritized sampling and a dueling architecture. The discussion will focus on prioritized sampling followed by an examination of the dueling architecture.

The drawback of random sampling in a standard DDQN is that not all interactions are equally useful for learning. When the agent makes a correct prediction, the interaction does not offer significant learning opportunities, whereas an incorrect prediction provides valuable learning opportunities for the agent to update its knowledge and improve its predictions.

The PRD-DDQN approach represents a significant advancement over a standard DDQN, by introducing the notion of prioritized sampling. This technique enables the agent to focus on the most informative interactions, which in turn improve its overall learning process. To achieve this, the PRD-DDQN assigns a probability to each quadruple $(s_t, a_t, r_t(s_t, a_t), s_{t+1})$ in its Replay Buffer, based on the absolute difference between prediction and target. This probability distribution is used to selectively prioritize those interactions that provide the most valuable learning opportunities. Let priority p_i denote the absolute difference between the prediction and target of observation i, and let K represent the number of quadruples in Replay Buffer. The parameter $\alpha \in (0, 1)$ determines the degree of prioritization applied, with a low value indicating a uniform distribution, while a high value indicates highly prioritized sampling. Equation 9 was defined by Schaul et al. and shows how the probability for each piece of information in memory is calculated.

$$P(i) = \frac{p_i^\alpha}{\sum_{k=1}^{K} p_k^\alpha} \tag{9}$$

Brim et al. employed a CNN in their research, which culminated in a linear layer of three nodes. Each node in this layer corresponds to a specific action that can be undertaken by the neural network, namely a long position, no position, or a short position. The PRD-DDQN expands upon this linear layer by introducing a dueling architecture as illustrated in Figure 2. In here, the conventional DDQN is shown on top, whereas the dueling DDQN is shown below. As can be seen, the dueling DDQN splits the linear layer into two streams: a state-value stream (layers above) an action advantage stream (layers below). As the names suggest, the action advantage stream aims to allocate values to all possible actions, while the state-value stream assigns values to a specific state. By considering the values of the actions and states independently, the dueling architecture is able to analyze them more than when considering only state-action pairs. The aggregation of state and action values, as described by Equation 3, is illustrated by the green lines. This approach yields a deeper understanding of the interactions between states and actions, resulting in more meaningful insights. Algorithm 2 shows how a PRD-DDQN functions in which the changes with respect to Algorithm 2 are shown in blue.

Algorithm 3 DDQN (Van Hasselt et al. 2016)

```
1:  Define: State and action space $S$ and $A$, resp.
2:  Initialize: Learning rate $\eta$, time step $t$, degree of prioritization $\alpha$, discount rate $\gamma$, reward function $r_t(s_t, a_t)$, number of epochs $N$, base number used for exponentially decaying exploration rate $c$, terminal state $T$, Replay Buffer $D$, batch size $z$, number of updates UPDATES, primary network $Q_\theta$, and target network $Q_{\theta'}$
3:  for epoch in 1,...,N do:
4:      Calculate: $\epsilon = max\{c^{epoch}, 0\}$ (6)
5:      Initialize current state: $s_0$
6:      Set terminal state: $T$ = False
7:      while T == False do:
8:          Generate a uniform number $p \in (0,1)$
9:          if $p < \epsilon$ then:
10:             Explore: Take random action $a_t$
11:         else:
12:             Exploit: Take action $a_t$ associated
13:             with the greatest Q-value for this
14:             state-action pair
15:         end if
16:         Observe next state: $s_{t+1}$
17:         Obtain reward: $r_t(s_t, a_t)$
18:         Calculate:
19:             $Q^*(s_t, a_t) \approx r_t(s_t, a_t) + \gamma \cdot$
20:             $Q_{\theta'}(s_{t+1}, \text{argmax}_{a'} Q_\theta(s_{t+1}, a'))$
21:         Calculate priorities $p_t = |Q^*(s_t, a_t) -$
22:         $Q(s_t, a_t)|$
23:         Save: $[s_t, a_t, r_t(s_t, a_t), s_{t+1}, p_t]$ to $D$
24:         if Final state is reached then:
25:             $T = True$
26:         end if
27:     end while
28:     Calculate priority probabilities $P(i)$ (8)
```

Algorithm 2 PRD-DDQN (Schaul et al. 2015; Wang et al. 2016; Van Hasselt et al. 2016)

```
29:     for update in 1,...,UPDATES do:
30:         Sample (priority): $z$ quintuples from $D$
31:         with replacement
32:         Calculate: $Q^*(s_t, a_t)$
33:         Update: $Q_\theta$ batch-wise
34:         $t = t + 1$
35:         if (update+1) mod 100 == 0 then:
36:             Update $Q_{\theta'}$
37:         end if
38:     end for
39: end for
```

3.2.4 Research Approach

As for the scope of our study, Brim et al. have not provided a clear specification of model and experiment parameter settings. Consequently, certain assumptions have to be made in this regard. The first assumption is regarding the CNN architecture in which Brim et al. indicate that the CNN architecture consists of an input layer (84 x 84 pixels), followed by three convolutional layers with 128, 256, and 512 neurons, respectively. The 128 neurons in the first convolutional layer could suggest that a kernel size of 4 was utilized in conjunction with 8 channels ($4 \cdot 4 \cdot 8 = 128$). However, it could also indicate that a kernel size of 8 was used with 2 channels ($8 \cdot 8 \cdot 2 = 128$). Similar uncertainties apply to the other convolutional layers. We assume that each layer employs a kernel size of 4 with 8, 16, and 32 channels, respectively. Additionally, Brim et al. did not clarify other pertinent parameters. Consequently, it is assumed that the Adam optimizer, Mean Squared Error (MSE) loss, and a batch size of 64 were used (see *Terminology* for an explanation of these terms). Furthermore, Brim introduced a variable referred to as the Negative Rewards Multiplier (NRM), which is used to scale negative rewards. This strategy penalizes inaccurate predictions and encourages the model to exercise caution and sometimes choose "no position" when uncertain. Brim did not disclose the specific value used for the NRM during training. We assume an NRM of 1. In addition, it is suggested that an enhanced experiment setup could be achieved through the utilization of a sequential deep learning model instead of a convolutional neural network (CNN), which would eliminate the need for data transformation into image format and may produce superior results. In order to ensure comparability with the results obtained by Brim et al., it is imperative that we adhere to the data transformation techniques utilized by the aforementioned researchers. Hence, we make the decision to retain the CNN and the candlestick image input.

The study conducted by Brim et al. claimed that a Double Deep Q-learning Network (DDQN) could surpass the performance of the S&P500 index during the period of January 1st 2020 to June 30th 2020. The purpose of our research is to validate this claim and extend it to a non-crisis period. Additionally, we introduce a new model called Prioritized Dueling Replay DDQN (PRD-DDQN) and compare its performance with the standard DDQN.

We train both the standard DDQN and PRD-DDQN on candlestick images obtained from the S&P500 index data set spanning from January 1st 2013 to December 31st 2019, which is the same time period as used by Brim et al. The training procedure follows that of Brim et al. and entails 1,000 epochs. To assess the efficacy of the trained models in learning meaningful relationships between features and targets, we subject both trained and untrained PRD-DDQNs and DDQNs to testing. If the performance of the untrained models is comparable to that of the trained models, it may indicate a failure of the trained models to establish and exploit the necessary feature-target relationships during the training phase. Finally, we include the geometric returns of taking a long position in the S&P500 index as a benchmark in accordance with Brim et al.

Ex.	Validation	Test
1	-	[01/01/2020, 30/06/2020]
2	-	[01/01/2021, 31/12/2021]
3	[01/01/2012, 31/12/2012]	[01/01/2020, 30/06/2020]
4	[01/01/2012, 31/12/2012]	[01/01/2021, 31/12/2021]

Table 3: Summary of Experimental Design: Four experiments were conducted utilizing the identical training set (January 1st, 2013 to December 31, 2019) while employing varying test sets. Experiments 3 and 4 incorporate a validation set.

Our research methodology comprises four experiments, as illustrated in Table 3. The objective of each experiment is to undertake a sensitivity analysis and to assess the average performance of each model type with regard to the Average Geometric Return metric. The sensitivity analysis entails training and testing models with varying initial weight parameter configurations on S&P500 index data. If these models fail to converge towards a near optimal solution, it may be deduced that they have not successfully identified a meaningful relationship between the features and target. In each experiment, we consider a trained DDQN, an untrained DDQN, a trained PRD-DDQN, an untrained PRD-DDQN, and the geometric returns generated by taking a daily long position in the S&P500 index.

The trained DDQN and PRD-DDQN models are constructed based on S&P500 index data, with the training phase ranging between January 1st, 2013 and December 31st, 2019. Note that this is the same training set as used by Brim et al. in their study.

Experiments 1 and 3 utilize a testing set of S&P500 index data spanning from January 1st 2020 to June 30th 2020, which is the same testing period used by Brim et al. Experiments 2 and 4 utilize a testing set of S&P500 index data ranging from January 1st 2021 to December 31st 2021 in order to include a non-crisis period during testing.

Our study employs a four-experiment design, wherein 100 different initial weight parameter settings are utilized for each model type (namely, untrained
DDQN, trained DDQN, untrained PRD-DDQN, or trained PRD-DDQN). In Experiments 1 and 2, these 100 models are tested, and the average performance of each model type is compared.

In contrast, Experiments 3 and 4 entail a slightly different approach. Here, the 100 models of each type are applied to a validation set spanning from January 1st 2012 to December 31st 2012. From these models, the top 10 best performing ones are selected based on the realized geometric returns at December 31st 2012. Only the top 10 models per type are then tested, with the goal of providing a more focused and precise assessment of model performance.

This experimental design enables a comprehensive evaluation of the different types of models under varying initial parameter settings and validation conditions, contributing to a better understanding of their performance in different contexts. Tables 4 and 5 provide an overview of the number of models of each type employed during training, validation and testing.

	DDQN*	PRD-DDQN*	DDQN	PRD-DDQN	S&P500
Tr.	-	-	100	100	-
Ts.	100	100	100	100	1

Table 4: Summary of the quantity of trained and tested models for Experiments 1 and 2. DDQN* and PRD-DDQN* denote the untrained models. Additionally, Tr. and Ts. correspond to the Train and Test set, respectively.

	DDQN*	PRD-DDQN*	DDQN	PRD-DDQN	S&P500
Tr.	-	-	100	100	-
Val.	100	100	100	100	-
Ts.	10	10	10	10	1

Table 5: Summary of the quantity of trained and tested models for Experiments 3 and 4. DDQN* and PRD-DDQN* denote the untrained models. Additionally, Tr. and Ts. correspond to the Train and Test set, respectively.

Model types are compared based on Average Geometric Return (AGR). Hence, for each experiment five AGRs corresponding to each model type and geometric returns generated by a long position in the S&P500 index are calculated. To gain insight into the selection of the 10 models in Experiments 3 and 4, a daily long position in the S&P500 index has been included in the *Results* section. Additionally, the trained models are evaluated using the training set to observe to which extent the models established meaningful feature-target relationships during training.
In the following section, the results are presented and analyzed.

4 Results

Table 6 presents the results of four experiments that were carried out. The primary objective of these experiments is to assess whether the models converge towards a near-optimal solution. Additionally, the experiments aim to evaluate whether the Average Geometric Return of all models of the same type significantly exceeds that of adopting a daily long position in the S&P500 index.

Ex.	Validation	Test
1	-	[01/01/2020, 30/06/2020]
2	-	[01/01/2021, 31/12/2021]
3	[01/01/2012, 31/12/2012]	[01/01/2020, 30/06/2020]
4	[01/01/2012, 31/12/2012]	[01/01/2021, 31/12/2021]

Table 6: Summary of Experimental Design: Four experiments were conducted utilizing the identical training set (January 1st, 2013 to December 31, 2019) while employing varying test sets. Experiments 3 and 4 incorporate a validation set.

Figures 7-13 present the development of geometric return over time per model type. For each figure, the y-axis denotes the relative profitability in comparison to the initial capital invested at the starting day. The value 1.0 denotes neither profit nor loss, while values below 1.0 indicate losses and values above 1.0 indicate profits.
Figures 7 and 10 correspond to Experiments 1 and 3, which were conducted to evaluate the performance of models in the time-period January 1st 2020 to June 30th 2020. Both experiments include taking a daily long position in the S&P500 index. The profit development generated by a daily long position in the S&P500 index is the same for both experiments. The difference between the experiments lies in the number of models utilized. In Experiment 1, 100 models of the same type were applied to the test set (Table 4), and the average profitability was subsequently calculated. In contrast, Experiment 3 employed the top 10 models that demonstrated superior performance on the validation set (Table 5), subsequently testing and calculating their average profitability. This methodology facilitates a more nuanced evaluation of model efficacy. The hypothesis is that the ten selected models will outperform the 100 models on the same test dataset. This is due to the higher probability of the selected models having discovered meaningful relationships between features and target, which can be utilized during testing.
Figures 8 and 13 depict the outcomes of Experi-

ments 2 and 4, respectively, which were conducted to assess the performance of models during the time-period spanning January 1st, 2021 to December 31st, 2021. As can be seen, the S&P500 index performed exceptionally well during this year. Both experiments include a daily long position in the S&P500 index, and the resultant profit development generated by this strategy is the same for the two experiments. However, the experiments vary in terms of the number of models employed. Specifically, Experiment 2 utilized 100 models of the same type on the test set, and the average profitability was subsequently determined. In contrast, Experiment 4 selected the top 10 models that exhibited superior performance on the validation set, subsequently testing and calculating their average profitability.

Type	Ex.1	Ex.2	Ex.3	Ex.4
DDQN*	-0.99%	-1.66%	**6.55%**	26.09%
PRD-DDQN*	-3.70%	-0.14%	-3.29%	**28.00%**
DDQN	-4.84%	-0.76%	0.35%	22.64%
PRD-DDQN	**2.49%**	-1.82%	4.59%	1.36%
S&P500	-4.04%	**26.89%**	-4.04%	26.89%

Table 7: Average Geometric Return generated per model type and experiment. DDQN* and PRD-DDQN* refer to the untrained model types, whereas DDQN and PRD-DDQN refer to the trained model types. The S&P500 index refers to the strategy of a daily long position in the S&P500 index. The best results are displayed in bold.

Table 7 refers to the achieved results per strategy and experiment in terms of Average Geometric Return. For more information regarding the worst and best performing models per experiment, see Appendix D. Figures 15 and 16 demonstrate the geometric returns realized by 100 trained DDQNs and PRD-DDQNs, which were applied to the train set spanning January 1st, 2013 to December 31st, 2019.

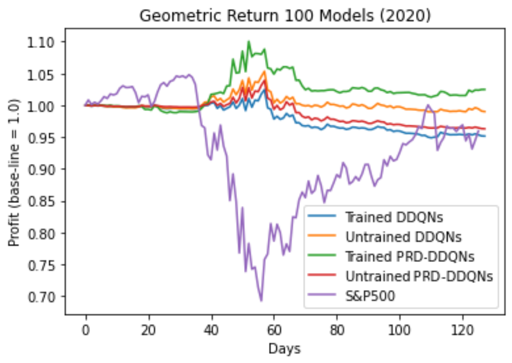


Figure 7: Experiment 1 - Geometric returns generated by trained and untrained DDQNs, trained and untrained PRD-DDQNs, and S&P500 index between January 1st 2020 and June 30th 2020.

Figure 7 shows that adopting a daily long position in the S&P500 index during the corona crisis might not be a viable strategy, since the model experienced a loss of approximately -30% on day 55, followed by a swift recovery, ultimately ending in a loss of -4.04%. This approach exhibited significant volatility. In contrast, the average performance of 100 model types exhibited relatively lower volatility. The untrained models performed relatively well, with both the untrained DDQNs and untrained PRD-DDQNs outperforming the trained DDQN in terms of Average Geometric Return.
Among the models, the trained PRD-DDQN realized the greatest Average Geometric Return of 2.49%. However, these results fall short of the Average Geometric Return of 13.16% achieved by Brim et al.

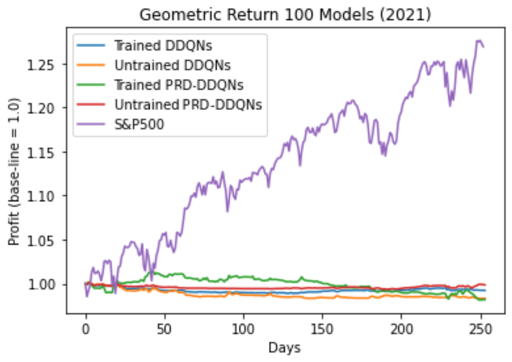


Figure 8: Experiment 2 - Geometric returns generated by trained and untrained DDQNs, trained and untrained PRD-DDQNs, and S&P500 index between January 1st 2021 and December 31st 2021.

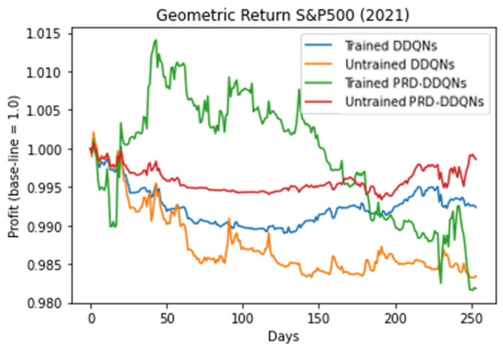


Figure 9: Experiment 2 - Geometric returns generated by trained and untrained DDQNs, trained and untrained PRD-DDQNs, without S&P500 index between January 1st 2021 and December 31st 2021.

Figure 8 shows that adopting a daily long position in the S&P500 index by far outperforms all other strategies, as it achieved an impressive Average Geometric Return of 26.89%, whereas all other strategies resulted in losses. Interestingly, there seems to be no substantial difference in performance between the trained and untrained models. The experimental results, presented in Figure 9, were obtained by comparing the profit development of multiple models during Experiment 2 in the absence of S&P500 index. The purpose of this comparison is to provide a clearer visual representation of the disparities between the models.

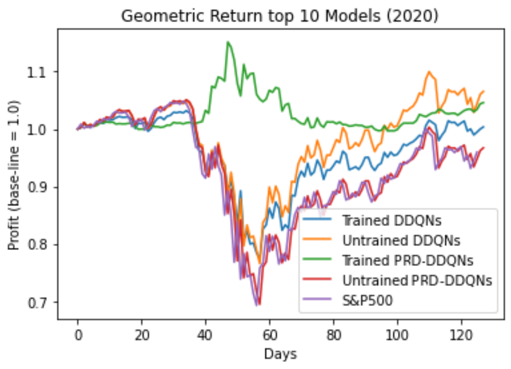
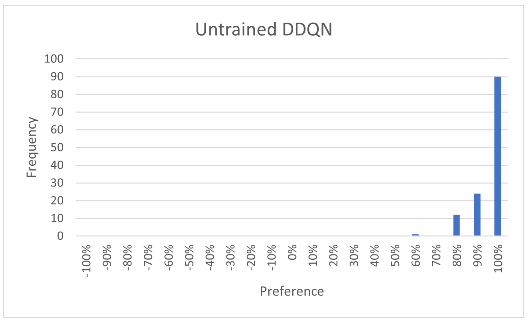
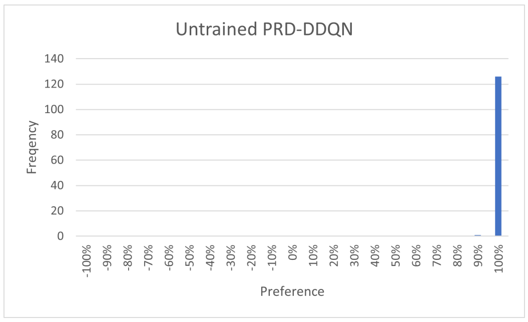


Figure 10: Experiment 3 - Geometric returns generated by top 10 trained and untrained DDQNs, top 10 trained and untrained PRD-DDQNs, and S&P500 index between January 1st 2020 and June 30th 2020.

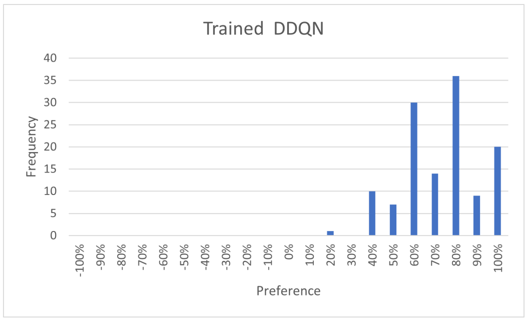
Figure 10 shows that all models surpass the Average Geometric Return generated by the daily long strategy in the S&P500 index at the final day (30th of June). The trained and untrained models exhibit comparable behavior with respect to profit development. This indicates that the selection process of the validation phase leads to the identification of untrained models that show comparable behavior. The untrained DDQNs exhibit the greatest Average Geometric Return in comparison to all other models on June 30th, 2020. Conversely, the trained PRD-DDQN demonstrate impressive performance during the sharp downturn observed in the market. Taken as a whole, there does not seem to be large differences in the Average Geometric Return achieved by each strategy on June 30th, 2020.
In order to evaluate the daily decision-making process of each model type, a histogram was constructed for Experiments 3 and 4. The x-axis of the histogram represents the model's preference for going long, neutral, or short. A preference of -100% denotes that all models have opted for a short position in a specific state, while 100% indicates a unanimous decision for a long position in a specific state. The y-axis displays the frequency among days of a specific position chosen. For instance, Figure 11b illustrates that all models unanimously agreed on a long position for 125 days during testing. For Experiment 3, the histograms are presented in Figures 11a-11d.



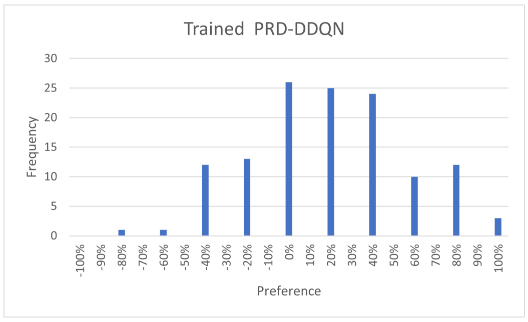
(a)



(b)

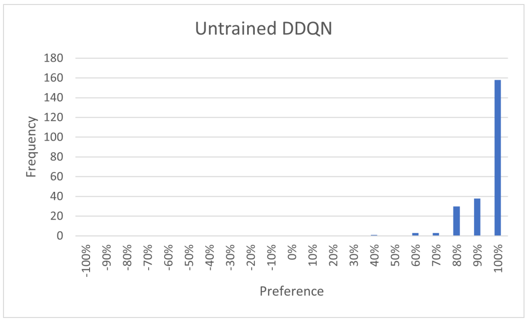


(c)

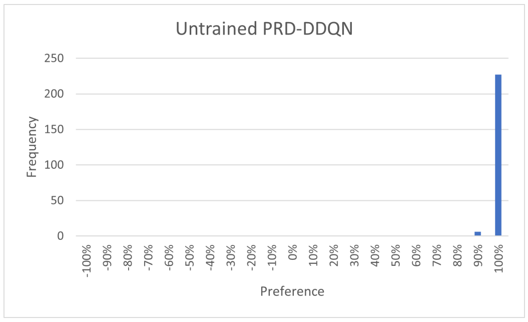


(d)

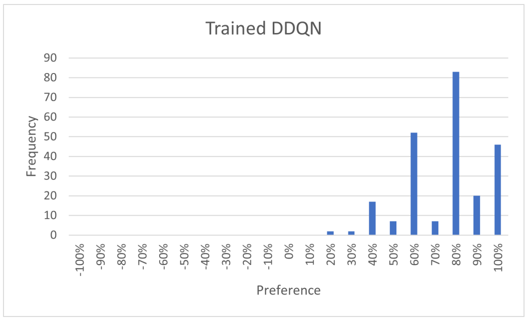
Figure 11: Experiment 3 - A Histogram Analysis of Model Preference for Long, Neutral, and Short Positions, where -100% implies all Models went Short, whereas 100% implies all Models went Long.



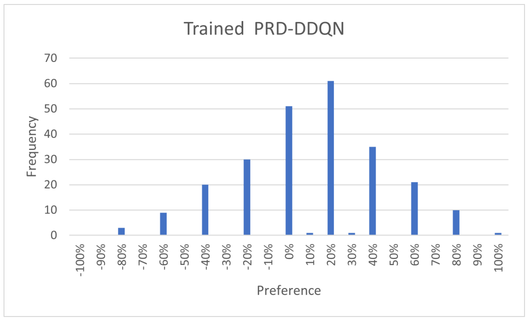
(a)



(b)



(c)



(d)

Figure 12: Experiment 4 - A Histogram Analysis of Model Preference for Long, Neutral, and Short Positions, where -100% implies all Models went Short, whereas 100% implies all Models went Long.

The figures show that the trained PRD-DDQN model slightly favors long positions. On the other hand, the trained DDQN, untrained DDQN, and untrained PRD-DDQN models show a strong preference for long positions.

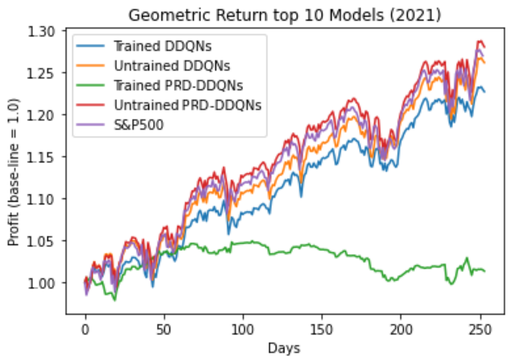


Figure 13: Experiment 4 - Geometric returns generated by top 10 trained and untrained DDQNs, top 10 trained and untrained PRD-DDQNs, and S&P500 index between January 1st 2021 and December 31st 2021.

Figure 13 shows that most models behave similar in terms of profit development, with trained DDQNs, untrained DDQNs, untrained PRD-DDQNs, and daily long positions in the S&P500 index demonstrating similar patterns. However, a notable deviation is observed in the performance of the trained PRD-DDQN at December 31st, which shows the worst Average Geometric Return. In contrast, the untrained PRD-DDQNs achieve the highest Average Geometric Return.

Figures 11a-11d displayed the preference per model type for long, neutral, or short positions in Experiment 3. Similarly, Figures 12c-12d illustrate the corresponding preferences observed in Experiment 4. In Figure 12d, a normal distribution is observed, suggesting that when models exhibit uncertainty regarding position selection, there is a likelihood of long-term cancellation (i.e., an approximate equal number of long and short positions taken on average). This phenomenon, depicted in Figure 12d, indicates that the models lack clear direction in determining their positions within Experiment 4.

Figure 14 showcases the geometric return obtained from a daily long position in the S&P500 index during the validation period from January 1st, 2012 to December 31st, 2012.

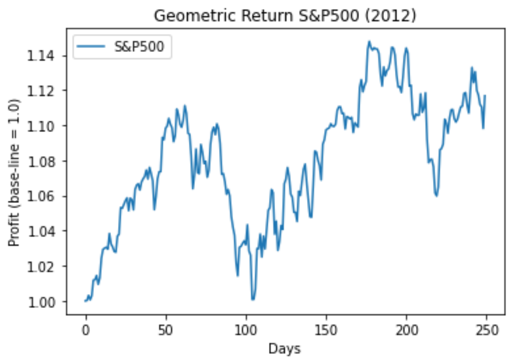


Figure 14: Geometric returns derived from a daily long position in the S&P 500 index spanning from January 1st, 2012 to December 31st, 2012.

Figure 14 illustrates that the application of a daily long position strategy in the S&P500 index during the validation period results in a geometric return of 11.16%. Consequently, this outcome leads to the selection of models that favor a long position during the validation phase.

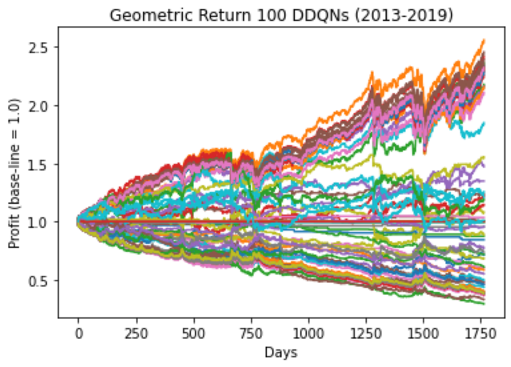


Figure 15: Geometric returns generated during training by 100 DDQNs between January 1st 2013 and December 31st 2019.

Figure 15 shows the development of geometric returns generated by each trained DDQN model when applied to the train set. The figure demonstrates that the trained DDQN attains a maximum geometric return of approximately 2.5, while several trained models generate a loss (i.e., a geometric return below 1). Notably, consistently taking a long position in the S&P 500 on a daily basis yields an approximate geometric return of 2.26.

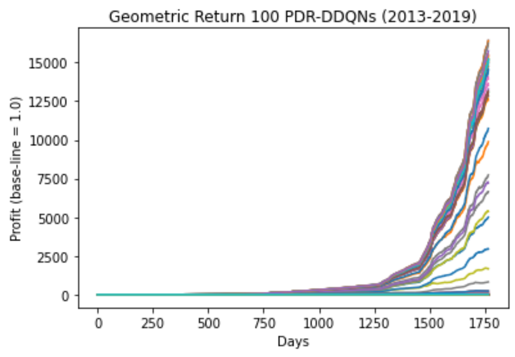


Figure 16: Geometric returns generated during training by 100 PRD-DDQNs between January 1st 2013 and December 31st 2019.

Figure 16 shows the development of geometric returns generated by each trained PRD-DDQN model when applied to the train set. The figure displays that certain trained PRD-DDQNs achieved exceptionally high geometric returns, with the highest recorded geometric return exceeding 1,500,000%. While this does not imply a complete overfit, it does approach a state of substantial overfitting. This observation implies that a small subset of the trained PRD-DDQN models were able to effectively leverage the relationships between the features and target during the training process.

5 Discussion

The experimental results presented in Figures 7-13 indicate that the trained PRD-DDQN exhibits distinct behavior from the remaining strategies. This difference in behavior becomes especially evident in Figures 8-13, where the trained PRD-DDQN frequently moves in different directions compared to the other strategies. In contrast, the trained DDQN displays similar behavior to the untrained models. This observation suggests that the PRD-DDQN was able to learn a relationship between the input features and target variable during training, while the DDQN was not. This theory is supported by the training performances of both trained models. A model that correctly predicts every action during training has the potential to achieve a geometric return of 21,446 times its initial investment. This is, of course, an outcome based on perfect foresight, but it remains a possibility during the training process. Figure 16 clearly demonstrates that several PRD-DDQN models were able to attain an exceptionally high geometric return when applied to the train set, with a few even achieving a geometric return of more than 15,000. Conversely, Figure 15 illustrates that the DDQN failed to generate geometric returns of comparable magnitude. These observations imply that certain PRD-DDQN models may converge to a policy that exhibits outstanding performance during training and, unlike the DDQN, that the PRD-DDQN may, in some instances, establish relationships between features and target. However, these results do not generalize to testing, indicating that some of the PRD-DDQN models overfit. As can be seen in Figure 16 and Table 7, the PRD-DDQN models that did not perform exceptionally well during training did not overfit but also did not generate impressive results during testing. Therefore, it seems that the PRD-DDQN was only able to establish useful relationships between features and target during training, while the DDQN was not able to establish notable relationships between features and target during training. As demonstrated in Table 7, both the trained PRD-DDQN and trained DDQN were incapable of consistently outperforming the other strategies during testing. These findings indicate that the PRD-DDQN and DDQN were unable to consistently leverage useful patterns in the candlestick plots to accurately predict target values.

Given the exceptional performance of the PRD-DDQN model during training, it is plausible to argue that the deep learning architecture effectively captured the intricate relationship between the features and target. The DDQN model was unable to achieve similar performance and, as a result, did not establish useful relationship between features and target. However, one could posit that by modifying the model's parameters, the DDQN framework could also acquire the ability to discern and incorporate this relationship. Moreover, expanding the size of the dataset could potentially enhance the performance of the DDQN model, as it is conceivable that the existing dataset might have been insufficient for establishing robust relationships between features and target. As illustrated in Figure 16, a subset of the PRD-DDQN models displayed signs of overfitting, prompting the exploration of regularization techniques to alleviate this propensity and ultimately yield improved outcomes on the test data sets. Notably, since all models failed to effectively generalize the acquired knowledge from training to the test set, it is worth considering alternative data representations to enhance the predictive capacity, as candlestick images may not be the most optimal form of data representation. The exploration of alternative data representations holds promise for potentially attaining more favorable results.

As presented in Table 7, all models performed better in Experiments 3 and 4 in comparison to Experiments 1 and 2, respectively. Given that all models, except for the PRD-DDQN model, exhibit a consistent inclination towards assuming long positions in Experiments 3 and 4, the outcomes obtained align closely with those derived from exclusively adopting a long position in the S&P500 index on a daily basis. In the event that the trained DDQN fails to identify a policy that performs quite well during training, its predictions become highly reliant on the initial parameter settings. Given the use of 100 different initial parameter settings, the DDQN may follow several distinct paths when applied to the validation set. It is important to note that a daily long position in the S&P500 index yields a geometric return of 11.16% during validation. Thus, if the DDQN cannot identify useful relationships between features and target (as suggested by Figure 15), it would be prudent to prefer a daily long position. Consequently, the top 10 DDQN models selected during validation prefer a long position and exhibit behavior that is highly similar to taking a daily long position in the S&P500 index during testing. This theory is supported by the results found in Figures 11c-11b, Figures 11c-12b. This shows that the trained DDQN far more often prefers adopting a long position in Experiments 3 and 4, as compared to the trained PRD-DDQN. Similar reasoning applies to the untrained models. The trained PRD-DDQN exhibited distinct behavior, lending support to the notion that the model was able to establish a relationship between features and target during training, while the DDQN could not. It is important to note that increasing the number of epochs during training may lead to the DDQN converging to a policy. Nevertheless, in order to explore the potential benefits of increasing the number of epochs, we conducted additional training with two DDQN models using 10,000 epochs. Subsequently, we evaluated the performance of these models in the period January 1st, 2020 to June 30th, 2020. Surprisingly, both models exhibited the same action on only 54 out of the 127 testing days, indicating a lack of convergence to a similar solution. Naturally, this observation does not offer sufficient empirical evidence to definitively support or refute the proposition that increasing the number of epochs for the DDQN leads to convergence. However, this does provides us with an indication of the potential consequences associated with such an increase.

As stated in *Theoretical Framework*, Prioritized Sampling ensures that the PRD-DDQN would converge faster than the standard DDQN. This is exactly what is observed.

In our study, we were unable to achieve similar results as obtained by Brim et al. in Experiments 1 and 3. There are several potential reasons for this outcome. First, Brim et al. trained their DDQN on the 30 largest stocks in the S&P500 index (the DDQN was trained and tested on individual stocks), which may have led to better performance. Second, Brim et al. employed a different set of parameters that resulted in better outcomes. Third, it is possible that the models were not able to identify a meaningful relationship between features and target during training because there is none, and that the results achieved by Brim et al. were simply the result of chance. Since the 30 largest stocks in the S&P500 are correlated with the S&P500 index overall, the first reason seems unlikely.

The assertion that model parameters alone are responsible for the observed unsatisfactory performance cannot accepted with absolute certainty. However, Brim et al. did not carry out a sensitivity analysis to investigate the impact of initial weight parameter configurations. As a result, the impressive outcomes found by Brim et al. may be attributable to favorable initial weight parameter settings. Consequently, there is uncertainty as to whether the assertion made by Brim et al. that a DDQN trained on candlestick plots outperforms the S&P500 index during the corona crisis is valid. To exclude the possibility that the impressive results achieved by Brim et al. are a consequence of favorable initial parameter settings, it is recommended to perform a sensitivity analysis. Since

Brim and colleagues did not include a sensitivity analysis and we were unable to achieve comparable outcomes, our findings do not provide support for the notion that technical analysis can consistently generate above-average returns. This further reinforces the perspective put forth by Hull.

In addition to the primary recommendations of utilizing the parameters as employed by Brim et al. and performing a sensitivity analysis, we propose several additional recommendations with regard to conducting similar research. Firstly, extending the training period to provide the DDQN with more data to establish a reliable policy is recommended. According to the paper *Revisiting the duration dependence in the US stock market cycles* (Zakamulin 2022) , the average duration of a market cycle in the US stock market is 3.5 years. Therefore, to accurately capture the returning patterns in these cycles, it is recommended to train the model on multiple market cycles. For instance, a suitable approach would be to expand the training dataset to 14 years, which would encompass four market cycles.

Secondly, it is advisable to apply a deep learning model directly to the raw data, rather than employing a methodology that involves transforming the data into a candlestick image and then utilizing a CNN for analysis. Throughout our research, we adhered mainly to the methodology outlined by Brim et al., which involves transforming data into candlestick images and feeding it to a CNN. Although this approach was essential for validating the work of Brim et al., it may not be the optimal method, as the raw data contains the same information as the image and the image includes a great deal of extraneous data, such as large white spaces. This makes the model computationally more expensive and does not add to the predictive capabilities of a model. Therefore, we recommend directly applying a CNN to the raw data or using a sequential model on the raw data. A suitable sequential model, such as the Transformer model described in the paper *Attention is All You Need* (Vaswani et al. 2017) , could significantly reduce computation time and potentially enhance performance, or at least not negatively impact performance.

The third recommendation focuses on the fact that Brim et al. concentrated on the corona crisis, but for an investor, a model that is confirmed to perform well only during a crisis may be insufficient. An investor desires to know if the model performs well during both crisis and non-crisis periods. Therefore, it is advisable to include a non-crisis period during testing.

The fourth recommendation is to train and test exclusively on the S&P500 index. This facilitates direct comparisons with taking a daily long position in the S&P500 index.

The fifth recommendation is including the PRD-DDQN during experiments. If a standard DDQN can consistently outperform the S&P500 index, then the PRD-DDQN should be capable of achieving similar or even better outcomes for reasons discussed in *Theoretical Framework*. In our study, however, we failed to uncover substantial evidence supporting the assertion that the DDQN model consistently outperforms the S&P500 index benchmark. Moreover, our experimental results indicate that the PRD-DDQN model, in the conducted experiments, was unable to surpass the performance of the DDQN model.

The sixth recommendation is to take risk into account as investors are not exclusively interested in Average Geometric Returns. The Sharpe Ratio (SR), defined in Equation Equation 10, is a tool to achieve this.

$$\mathrm{SR} = \frac{E[R_a - R_f]}{\sigma_a} \tag{10}$$

In the equation, R_a, R_f, and σ_a represent the realized return of the asset, risk-free rate of return (e.g., 10Y US-Treasury bond rate), and volatility of returns, respectively. However, it should be noted that the Sharpe Ratio assumes normality of asset returns, which may not always be the case. For instance, when R_a and R_f exhibit exponential growth over time, the differences between them may become exceedingly large towards the end of the time-period. This phenomenon implies that the returns realized at the end of the time period will have a significant impact on the expected value, leading to a distortion of the Sharpe Ratio. The research question was:

To what extent can technical analysis be employed to achieve sustained above-average returns when converting stock data into candlestick charts and *subsequently utilizing it as input for a Reinforcement Learning model to forecast the percentage price movement of a stock from one trading day to the next?*

Our study could not obtain similar results as suggested by the study titled *Deep Reinforcement Learning Stock Market Trading, Utilizing a CNN with Candlestick Images* (Brim et al. 2022) . Since Brim et al. did not report a sensitivity analysis, there is reasonable doubt if the results accurately reflect model performance. Hence, we could not find evidence to disprove the claim of Hull that technical analysis cannot achieve consistent above-average returns. Moreover, our analysis revealed that both the DDQN and the PRD-DDQN models failed to outperform a simple strategy of taking a daily long position in the S&P500 index, suggesting neither model could identify useful relationships between features and target. The PRD-DDQN model established a relationship that is solely advantageous during training, as it only recognizes distinctive features that only exist in the training dataset. Hence, the model could not generalize and discern useful patterns in the candlestick plot. Due to its poor performance in both training and testing, the DDQN model was unable to detect any meaningful relationships between features and targets too.

6 Conclusion

The study *Deep Reinforcement Learning Stock Market Trading, Utilizing a CNN with Candlestick Images* (Brim et al. 2022) proposes an approach to stock market trading by combining a Double Deep Q-learning Network (DDQN) with a Convolutional Neural Network (CNN). Specifically, the approach involves the transformation of financial data into candlestick images, which are subsequently used as inputs for the DDQN. The premise is that the DDQN can identify patterns in the candlestick images, which enables it to anticipate the best course of action for the succeeding day. The action space consists of three actions: long position, no position or short position. The aim is to maximize profit.
Our objective was to replicate the results obtained by Brim et al. while introducing several modifications to the experimental setup. First, an additional testing period was introduced to assess model performance during non-crisis periods. Second, the model was trained and tested solely on the S&P500 index to enable direct comparisons with a daily long position in the S&P500 index. Third, we introduced a second model, the Prioritized Replay Dueling DDQN (PRD-DDQN). Lastly, we conducted a sensitivity analysis to evaluate the impact of the initial weight parameter configuration on the model's predictive capabilities.

To comprehensively evaluate model performance measured with the Average Geometric Return, we utilized five different strategies, including a trained DDQN, a trained PRD-DDQN, an untrained DDQN, an untrained PRD-DDQN, and a daily long position in the S&P500 index. The inclusion of untrained models was intended to assess the extent to which trained models outperformed their untrained counterparts, thereby indicating the degree to which the models had learned an effective policy during training. Regarding the trained models, the train set consists of S&P500 index data spanning from January 1st, 2013 to December 31st, 2019.
In our study, four experiments are conducted. These experiments are designed to ascertain whether similar results can be obtained under the similar conditions as those described in the research of Brim et al. Experiments 1 and 3 utilize a test set of S&P500 index data spanning from January 1st 2020 to June 30th 2020, which is the same test period used by Brim et al. who view this period as a crisis. Experiments 2 and 4 utilize a test set of S&P500 index data ranging from January 1st 2021 to December 31st 2021 in order to include a non-crisis period during testing. In Experiments 1 and 2, 100 models are tested, and the average performance of each model type is compared. In contrast, Experiments 3 and 4 entail a slightly different approach. Here, 100 models of each type are applied to a validation set spanning from January 1st 2012 to December 31st 2012. From these models, the top 10 best performing ones are selected based on the realized geometric returns at December 31st 2012. Solely the best-performing 10 models of each type undergo testing. The idea is that a subset of the 100 models might have the ability to recognize recurring patterns in the candlestick chart, enabling them to make informed decisions about which position to adopt on the succeeding

day. By selecting the top 10 models based on performance, the testing procedure can maximize the possibility of identifying the most accurate predictive models. For each experiment, a daily long position in the S&P500 index is taken into account. What becomes apparent from the experiment outcomes is that the trained DDQN exhibited comparable behavior to its untrained counterparts across all experiments. In contrast, the PRD-DDQN demonstrated distinct behavior compared to all other models. Figure 16 illustrates that certain PRD-DDQNs achieved extremely high geometric returns during training by identifying the correlations between features and target. Nonetheless, these models overfitted during training because they were unable to produce similar results during testing. Furthermore, PRD-DDQNs that did not perform well during training also performed poorly during testing. Additionally, in Experiments 3 and 4 (see Figures 10 and 13), the trained DDQN, untrained DDQN, and untrained PRD-DDQN demonstrated similar behavior.

As suggested by Figure 15, the DDQN did not discover meaningful relationships between features and target during training. This indicates that the DDQN might be heavily reliant on the initial weight parameter settings, and thus, when a set of 100 trained DDQNs are employed on the validation dataset, each model's trajectory differs considerably. Some DDQNs could opt for an overall short position regardless of the state they find themselves in, while others might opt for an overall long position. Given that taking a daily long position in the S&P500 index during validation results in a geometric return of 11.16%, the DDQNs that tend towards an overall long position are selected for testing. Consequently, the top 10 trained DDQN models roughly adhere to a daily long position in the S&P500 index in Experiments 3 and 4. Similar reasoning applies to the untrained models. Note that the trained PRD-DDQN did not follow an overall long position in Experiments 3 and 4. As indicated by Figure 16, this is probably due to the fact that the PRD-DDQN did learn a policy which worked well during training and did identify relationships between features and target. As a result, the 100 PRD-DDQNs applied to the validation set do not favor an overall long position regardless of input. Once again this underlines the hypothesis that the PRD-DDQN was able to learn an effective policy during training whereas the DDQN was not. As discussed in *Theoretical Framework*, the PRD-DDQN might converge faster than the DDQN due to the usage of Prioritized Sampling. Note that the DDQN might converge as the number of epochs is increased. Nevertheless, we opted to preserve the parameter of 1,000 epochs as defined by Brim et al. in their research. This decision was motivated by our aim to closely align our experimental approach with theirs, in order to obtain similar outcomes.

In our study, we were unable to replicate the results obtained by Brim et al. in Experiments 1 and 3. There are several potential reasons for this outcome. First, Brim et al. trained their DDQN on the 30 largest stocks in the S&P500, which may have led to better performance. Second, Brim et al. may have employed a different set of parameters that resulted in better outcomes. Third, it is possible that the models were not able to identify a meaningful relationship between features and target during training, and that the results achieved by Brim et al. were simply the result of chance. Since the 30 largest stocks in the S&P500 are highly correlated with the S&P500 index overall, the first reason seems unlikely.

In order to rule out the possibility that unsatisfactory performance was due to model parameters, such as kernel size or Negative Rewards Multiplier (NRM), we attempted to clarify which parameters were employed by seeking contact with Dr. Brim. We were unable to retrieve the original parameter settings, and as a result, we cannot dismiss the probability that incorrect model parameters contributed to the suboptimal performance observed. However, Brim et al. did not report that a sensitivity analysis was conducted to examine the influence of initial weight parameter configurations. Therefore, the impressive outcomes reported by Brim et al. could be attributed to favorable initial weight parameter settings. This uncertainty raises questions about the validity of the claim that a DDQN trained on candlestick plots outperforms the S&P500 index during the corona crisis. To eliminate the possibility that the remarkable results attained by Brim et al. are due to advantageous initial parameter settings, a sensitivity analysis is recommended. This analysis would aid

in excluding any possibility that the exceptional outcomes achieved are a consequence of favorable initial parameter settings.
The research question was:

To which extent can technical analysis be used to generate consistent above-average returns when following the approach as described in "Deep Reinforcement Learning Stock Market Trading, Utilizing a CNN with Candlestick Images"?

In our research, we were unable to obtain outcomes comparable to those reported by Brim et al. Notably, the lack of a sensitivity analysis brings into question the reliability of the conclusions
reached by Brim and colleagues. As a result, we have found no grounds to challenge Hull's assertion that technical analysis cannot consistently produce superior returns.

References

Brim, A., & Flann, N. S. (2022). Deep reinforcement learning stock market trading, utilizing a cnn with candlestick images. *Plos one*, *17*(2), e0263181. https://doi.org/10.1371/journal.pone.0263181

Cartea, Á., Jaimungal, S., & Sánchez-Betancourt, L. (2021). Deep reinforcement learning for algorithmic trading. *Available at SSRN 3812473.*

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning* [http://www.deeplearningbook.org]. MIT Press.

Huang, J., Chai, J., & Cho, S. (2020). Deep learning in finance and banking: A literature review and classification. *Frontiers of Business Research in China*, *14*(1), 1–24. https://doi.org/10.1186/s11782-020-00082-6

Hull, J. C. (2003). *Options futures and other derivatives.* Pearson Education India.

Investopedia. (2021). *Candlestick charting: What is it?* Investopedia. https://www.investopedia.com/trading/candlestick-charting-what-is-it/

Jiang, Z., Xu, D., & Liang, J. (2017). A deep reinforcement learning framework for the financial portfolio management problem. *arXiv preprint arXiv:1706.100*

Jurgovsky, J., Granitzer, M., Ziegler, K., Calabretto, S., Portier, P.-E., He-Guelton, L., & Caelen, O. (2018). Sequence classification for credit-card fraud detection. *Expert Systems with Applications*, *100*, 234–245. https://doi.org/j.eswa.2018.01.037

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436–444. https://doi.org/10.1038/nature14539

Lee, K., & Jo, G. (1999). Expert system for predicting stock market timing using a candlestick chart. *Expert systems with applications*, *16*(4), 357–364. https://doi.org/10.1016/S0957-4174(99)00011-1

Lucarelli, G., & Borrotti, M. (2020). A deep q-learning portfolio management framework for the cryptocurrency market. *Neural Computing and Applications*, *32*, 17229–17244. https://doi.org/10.1007/s00521-020-05359-8

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. *arXiv preprint* *arXiv:1511.05952*. https://doi.org/10.48550/arXiv.1511.05952

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., et al. (2020). Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, *588*(7839), 604–609. https://doi.org/10.1038/s41586-020-03051-4

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. *Proceedings of the AAAI conference on artificial intelligence*, *30*(1). https://doi.org/10.48550/arXiv.1509.06461

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, *30*. https://doi.org/10.48550/arXiv.1706.03762

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., & Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. *International conference on machine learning*, 1995–2003.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*, 279–292. https://doi.org/10.1007/BF00992698

Zakamulin, V. (2022). Revisiting the duration dependence in the us stock market cycles. *Applied Economics*, 1–12.

A Candlestick Chart Generation (Python Code)

```python
import torch
import numpy as np
import yfinance as yf
import torchvision.transforms as transforms
from pandas.tseries.offsets import BDay
from datetime import datetime
from joblib import dump
from tqdm import tqdm
from PIL import Image

def normaliser(data):

    #get maximum and minimum of highs and lows in 28 days
    max_val = data["High"].max()
    min_val = data["Low"].min()

    #normalise outputs and multiply with 84 (to scale candles in figure)
    output = ((data[["High", "Low", "Open", "Close"]]-min_val) / (max_val-min_val))
    *84

    #round all values to nearest integer
    output = output.round(0).astype("int")

    #reset window index
    return output.reset_index(drop=True)

#parameter selection
SYMBOLS = ["^GSPC"]
TYPE = "Test"
START_DAY = 1
START_MONTH = 1
START_YEAR = 2020
END_DAY = 30
END_MONTH = 6
END_YEAR = 2020

#define path
path = f"X\\{TYPE}"

#create 2d numpy array (84x84) with only white pixels
fig = np.zeros((84,84))
fig.fill(255)

#define torch transformations
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Grayscale()
])

for symbol in tqdm(SYMBOLS):

    #import data
    df = yf.download(
            symbol,
            start=datetime(START_YEAR, START_MONTH, START_DAY)-BDay(29),
            end=datetime(END_YEAR, END_MONTH, END_DAY)+BDay(1),
            interval='1d',
            progress=False
            ).reset_index()

    #calculate number of images to generate
    N = len(df) - 28

    #initialisation
    images = []
    labels = []

    for i in tqdm(range(N)):

        #create window
        window = normaliser(df[i:i+28].copy())

        #copy white canvas
        canvas = fig.copy()

        for j in range(28):

            #calculate body of the candle
            body = window["Close"][j] - window["Open"][j]

            if body < 0:

                #calculate start and end body (negative body implies Open > Close)
                start_body = window["Close"][j]
                end_body = window["Open"][j]

                #to generate images body must by nonnegative
                body = abs(body)

                #black color is assigned to negative candles
                color = 0

            else:

                #calculate start and end body (positive body implies Close > Open)
                start_body = window["Open"][j]
                end_body = window["Close"][j]

                #gray color is assigned to positive candles
                color = 105

            #add body to canvas for corresponding day
            canvas[start_body:end_body,3*j:3*j+3] = color

            #add high to canvas for corresponding day
            canvas[end_body:window["High"][j],3*j+1] = color

            #add low to canvas for corresponding day
            canvas[window["Low"][j]:start_body,3*j+1] = color

        #horizontal flip to ensure correct form
        canvas = np.flip(canvas, axis=0)

        #convert numpy array to PIL image (RGB)
        canvas = Image.fromarray(canvas)
        canvas = canvas.convert('RGB')

        #append images with transformed canvas (now a grayscale tensor)
        images.append(transform(canvas.copy()))

        #append labels with percental change of this day to the next
        labels.append(torch.tensor([(df["Close"][i+28]-df["Close"][i+27])/df["Close"
    ][i+27]]))

    dump(torch.stack(images, dim=0), f"{path}\\{symbol}_X.joblib")
    dump(torch.stack(labels, dim=0), f"{path}\\{symbol}_y.joblib")
```

B Standard Double Deep Q-learning Network

```
import copy
import math
import torch
import numpy as np
import pandas as pd
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import matplotlib.pyplot as plt
from joblib import load, dump
from tqdm import tqdm
import time as time

#set random seed
torch.manual_seed(42)
torch.cuda.manual_seed(42)

def plot(x, title="MSE Loss", label="Error", xlabel="Epochs", ylabel="Error"):

    #define figure and axis
    fig, ax = plt.subplots()

    #define x- and y-axis (+label)
    ax.plot([i for i in range(1,len(x)+1)], x)

    #define title
    plt.title(title)

    #define labels
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)

    #show plot
    plt.show()

class CNN(nn.Module):
    def __init__(self,
                 img_size,
                 conv_layers=(1,8,16,32),
                 conv_kernel=4,
                 conv_padding=0,
                 conv_stride=1,
                 pool_kernel=2,
                 pool_padding=0,
                 pool_stride=2):
        super(CNN, self).__init__()

        #initialisation
        self.img_size = img_size
        self.conv_kernel = conv_kernel
        self.conv_padding = conv_padding
        self.conv_stride = conv_stride
        self.pool_kernel = pool_kernel
        self.pool_padding = pool_padding
        self.pool_stride = pool_stride
        self.conv_len = len(conv_layers) - 1
        self.conv_output = conv_layers[-1]

        #declare convolutional layers
        self.conv_layers = nn.ModuleList([nn.Conv2d(in_channels=conv_layers[i],
    out_channels=conv_layers[i + 1], kernel_size=conv_kernel, stride=conv_stride,
    padding=conv_padding) for i in range(len(conv_layers) - 1)])

        #declare last linear layer
        self.fc = nn.Linear(self.conv_to_lin(), 3)

    def conv_to_lin(self):

        #initialise size
        size = self.img_size

        #iterate over all convolutional layers
        for i in range(self.conv_len):

            #calculate size after convolutional layer (i.e. rounddown((size - k + 2p)
    /stride) + 1)
            size = math.floor((size-self.conv_kernel+2*self.conv_padding)/self.
    conv_stride) + 1

            #calculate size after pooling layer (i.e. rounddown((size - k + 2p)/
    stride) + 1)
            size = math.floor((size-self.pool_kernel+2*self.pool_padding)/self.
    pool_stride) + 1

        #calculate number of required neurons (i.e. width*height*channels)
        size = (size*size)*self.conv_output

        return size

    def forward(self, x):

        #pass data through conv layers and apply max. pooling
        for layer in self.conv_layers:
            x = F.max_pool2d(input=F.relu(layer(x)), kernel_size=self.pool_kernel,
    stride=self.pool_stride, padding=self.pool_padding)

        #ensure correct dimension is used
        if len(x.shape) == 3:
            s = 0
        else:
            s = 1

        #flatten image for linear layers
        x = torch.flatten(x, start_dim=s)

        #pass x trough last linear layer
        x = self.fc(x)

        #get final prediction
        return x

class ExperienceReplay():
    def __init__(self):

        #initialization
        self.quadruple = ['states', 'actions', 'rewards', 'next_states']
        self.buffer = {key:[] for key in self.quadruple}

    def size(self):
        return len(self.buffer['actions'])

    def add(self, x):

        #if buffer size exceeds maximum, then remove oldest items
        if self.size() >= 1763:

            #get batch size to delete old items
            del_len = len(x[0])

            #remove all oldest batches with size del_len from buffer
            for key in self.quadruple:
                self.buffer[key] = self.buffer[key][del_len:]

        #add new items
        for i, key in enumerate(self.quadruple):
            self.buffer[key].extend(x[i])

    def sample(self):

        #initialise output
        output = []

        #sample replay indices
        idx = torch.randint(low=0, high=1763, size=(64,))

        #apply sampling and select 64 quadruples
        for elem in self.quadruple:
            output.append(torch.stack([self.buffer[elem][i] for i in idx], dim=0))

        return output

class Agent():
    def __init__(self):

        #initialization
        self.gamma = 0.95

    def take_actions(self, states_batch):

        #initialization
        batch_size = len(states_batch)
        actions = torch.randint(0,3,(batch_size,)).to(device)
        p = torch.rand((batch_size,))

        #get state indices to exploit based on epsilon
        exploit_indices = (p>epsilon).nonzero().flatten()

        #get actions
        actions[exploit_indices] = self.predict(primary_model, states_batch[
    exploit_indices]).argmax(dim=1)

        return actions

    def predict(self, model, states_batch):

        #get prediction and do not accumulate gradients
        with torch.no_grad():
            return model(states_batch)

class Fit():
    def __init__(self):

        #intialization
        self.nrm = 1
        self.optimizer = optim.Adam(primary_model.parameters(), lr=1e-3)

    def environment_step(self, y, primary_model, target_model):

        for i in range(0, 1763, 64):

            #get states batch
            states_batch = states[i:i+64]

            #get next states batch
            next_states_batch = next_states[i:i+64]

            #get y batch
            y_batch = y[i:i+64]

            #get actions
            actions_batch = agent.take_actions(states_batch)

            #get rewards
            rewards_batch = y_batch*(actions_batch-1)

            #multiply rewards by Negative Rewards Multiplier (NRM)
            rewards_batch[rewards_batch<0] = rewards_batch[rewards_batch<0]*self.nrm

            #add quadruples to replay buffer
            replay.add([states_batch, actions_batch, rewards_batch, next_states_batch
])

    def update_step(self, primary_model, target_model):

        #initialization
        error = []

        for update in range(UPDATES):

            #sample from replay buffer
            states_batch, actions_batch, rewards_batch, next_states_batch = replay.
sample()

            #get primary model predictions for next states
            primary_pred = primary_model(next_states_batch)

            #identify best actions
            primary_best_actions = primary_pred.argmax(dim=1)

            #get target model predictions for next states
            target_pred = target_model(next_states_batch)

            #select Q-values based on previously calculated actions
            target_best_qs = target_pred[torch.arange(64), primary_best_actions]

            #calculate Q* (target)
            q_star = rewards_batch + agent.gamma*target_best_qs

            #get primary model predictions for current states
            q_pred = agent.predict(primary_model, states_batch)

            #get Q* in the correct form
            q_star_matrix = q_pred.clone()
            q_star_matrix[torch.arange(64), actions_batch] = q_star

            #set gradient equal to zero to prevent unwanted accumulation of gradients
            self.optimizer.zero_grad()

            #get loss
            loss = loss_function(q_pred, q_star_matrix)

            #append error
            error.append(loss)

            #calculate gradients for each layer in model (backwards fashion)
            loss.backward()

            #update model with specified update method
            self.optimizer.step()

            #update target network
            if update%99 == 0:

                #copy primary model to target model
                target_model = copy.deepcopy(primary_model)

        #return average error as tensor and as a percentage
        return torch.stack(error, dim=0).mean()*100

#PARAMETER SELECTION-----------------------------------------------------------

RUNS = 100
EPOCHS = 1000
UPDATES = 100

EPSEXP = 0.15

USE_CUDA = True

#PARAMETER SELECTION-----------------------------------------------------------

#use either GPU or CPU
if USE_CUDA and torch.cuda.is_available():
    device = torch.device('cuda')
    print("GPU is used!")
else:
    device = torch.device('cpu')
    print("CPU is used!")

#import X and y
X_train = load("C:\\Users\\Frits\\OneDrive\\Desktop\\University of Twente\\AM Thesis
    \\Zanders\\Prediction Program\\Data\\Train\\^GSPC_X.joblib").float().to(device)
y_train = torch.squeeze(load("C:\\Users\\Frits\\OneDrive\\Desktop\\University of
    Twente\\AM Thesis\\Zanders\\Prediction Program\\Data\\Train\\^GSPC_y.joblib")).
    float().to(device)*100
X_test = load("C:\\Users\\Frits\\OneDrive\\Desktop\\University of Twente\\AM Thesis\\
    Zanders\\Prediction Program\\Data\\Test\\^GSPC_X.joblib").float().to(device)
y_test = torch.squeeze(load("C:\\Users\\Frits\\OneDrive\\Desktop\\University of
    Twente\\AM Thesis\\Zanders\\Prediction Program\\Data\\Test\\^GSPC_y.joblib")).
    float().to(device)*100

#initialize loss function
```

```python
loss_function = nn.MSELoss()

#calculate base used for exponential decay
eps_base = EPSEXP**(1/math.floor(EPOCHS/2))

#truncate X- and y-train for calculation (the last state can never transition into a
    new state)
states = X_train[:-1]
next_states = X_train[1:]
y = y_train[:-1]

#initialize dataframe to store rewards
df_rewards = pd.DataFrame(np.arange(127), index=np.arange(127), columns=["Index"])

#register start time for iteration
stime = time.time()

for run in range(RUNS):

    #initialization
    primary_model = CNN(84).to(device)
    target_model = copy.deepcopy(primary_model)
    replay = ExperienceReplay()
    agent = Agent()
    fit = Fit()
    train_errors = []
    test_rewards = []

    for epoch in tqdm(range(EPOCHS)):

        #get exponentially decaying epsilon with minimum exploration rate of 10%
        epsilon = max(eps_base**epoch, 0.1)

        #take environment step
        fit.environment_step(y, primary_model, target_model)

        #take update step and store train errors
        train_errors.append(fit.update_step(primary_model, target_model).cpu().detach
    ().numpy())

        #initialization
        rewards = []

        #do not accumulate gradients
        with torch.no_grad():
            for x_, y_ in zip(X_test, y_test):

                #get action given X-test
                action = primary_model(x_).argmax(dim=0) - 1

                #get reward
                reward = y_*action

                #append rewards
                rewards.append(reward.cpu().detach().item())

        #clear memory
        torch.cuda.empty_cache()

        #store test rewards
        test_rewards.append(np.mean(rewards))

    #store model
    dump(primary_model, f'Standard\\model_{run}.joblib')

    #store rewards
    df_rewards[f"RUN_{run}"] = rewards

#plot results
plot(train_errors)
plot(test_rewards, title="Average Rewards", label="Average Reward", xlabel="Epochs",
    ylabel="Average Rewards (%)")
plot(rewards, title="Rewards (01-01-2020, 30-06-2020)", label="Reward", xlabel="Days"
    , ylabel="Rewards (%)")

#write df to excel
df_rewards.to_excel('df_Standard_Rewards.xlsx')

print(f"Total computation time: {time.time()-stime}s")
```

C Prioritized Replay Dueling Double Deep Q-learning Network

```python
import copy
import math
import torch
import numpy as np
import time as time
import pandas as pd
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import matplotlib.pyplot as plt
from joblib import load, dump
from tqdm import tqdm

#set seed
torch.manual_seed(42)
torch.cuda.manual_seed(42)

def plot(x, title="MSE Loss", label="Error", xlabel="Epochs", ylabel="Error"):

    #define figure and axis
    fig, ax = plt.subplots()

    #define x- and y-axis (+label)
    ax.plot([i for i in range(1,len(x)+1)], x)

    #define title
    plt.title(title)

    #define labels
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)

    #show plot
    plt.show()

class CNN(nn.Module):
    def __init__(self,
                 img_size,
                 conv_layers=(1,8,16,32),
                 conv_kernel=4,
                 conv_padding=0,
                 conv_stride=1,
                 pool_kernel=2,
                 pool_padding=0,
                 pool_stride=2):
        super(CNN, self).__init__()

        #initialisation
        self.img_size = img_size
        self.conv_kernel = conv_kernel
        self.conv_padding = conv_padding
        self.conv_stride = conv_stride
        self.pool_kernel = pool_kernel
        self.pool_padding = pool_padding
        self.pool_stride = pool_stride
        self.conv_len = len(conv_layers) - 1
        self.conv_output = conv_layers[-1]

        #declare convolutional layers
        self.conv_layers = nn.ModuleList([nn.Conv2d(in_channels=conv_layers[i], out_channels=conv_layers[i + 1], kernel_size=conv_kernel, stride=conv_stride, padding=conv_padding) for i in range(len(conv_layers) - 1)])

        #value stream for states
        self.fc_h_v = nn.Linear(self.conv_to_lin(), 3)
        self.fc_z_v = nn.Linear(3, 1)

        #advantage stream for actions
        self.fc_h_a = nn.Linear(self.conv_to_lin(), 3)
        self.fc_z_a = nn.Linear(3, 3)

    def conv_to_lin(self):

        #initialise size
        size = self.img_size

        #iterate over all convolutional layers
        for i in range(self.conv_len):

            #calculate size after convolutional layer (i.e. rounddown((size - k + 2p)/stride) + 1)
            size = math.floor((size-self.conv_kernel+2*self.conv_padding)/self.conv_stride) + 1

            #calculate size after pooling layer (i.e. rounddown((size - k + 2p)/stride) + 1)
            size = math.floor((size-self.pool_kernel+2*self.pool_padding)/self.pool_stride) + 1

        #calculate number of required neurons (i.e. width*height*channels)
        size = (size*size)*self.conv_output

        return size

    def forward(self, x):

        #pass data through conv layers and apply max. pooling
        for layer in self.conv_layers:
            x = F.max_pool2d(input=F.relu(layer(x)), kernel_size=self.pool_kernel, stride=self.pool_stride, padding=self.pool_padding)

        #ensure correct dimension is used
        if len(x.shape) == 3:
            s = 0
        else:
            s = 1

        #flatten image for linear layers
        x = torch.flatten(x, start_dim=s)

        #get state layer values
        v = self.fc_h_v(x)
        v = self.fc_z_v(v)

        #get action layer values
        a = self.fc_h_a(x)
        a = self.fc_z_a(a)

        #combine streams
        q = v + a - a.mean()

        #get final prediction
        return q

class ExperienceReplay():
    def __init__(self):

        #initialization
        self.alpha = 0.95
        self.quintuple = ['states', 'actions', 'rewards', 'next_states', 'priorities'
    ]
        self.buffer = {key:[] for key in self.quintuple}

    def size(self):
        return len(self.buffer['actions'])

    def add(self, x):

        #if buffer size exceeds maximum, then remove oldest items
        if self.size() >= 1763:

            #get batch size to delete old items
            del_len = len(x[0])

            #remove all oldest batches with size del_len from buffer
            for key in self.quintuple:
                self.buffer[key] = self.buffer[key][del_len:]

        #add new items
        for i, key in enumerate(self.quintuple):
            self.buffer[key].extend(x[i])

    def sample(self, probs):

        #initialise output
        output = []

        #sample replay indices
        idx = torch.multinomial(probs, 64, replacement=True)

        #apply priority sampling and select 64 quintuples
        for elem in self.quintuple[:-1]:
            output.append(torch.stack([self.buffer[elem][i] for i in idx], dim=0))

        return output

    def get_probs(self):

        #calculate priority probabilities
        prob = [el**self.alpha for el in replay.buffer['priorities']]
        probsum = sum(prob)
        probs = [el/probsum for el in prob]

        #convert to tensor
        return torch.stack(probs, dim=0)

class Agent():
    def __init__(self):

        #initialization
        self.gamma = 0.95

    def take_actions(self, states_batch):

        #initialization
        batch_size = len(states_batch)
        actions = torch.randint(0,3,(batch_size,)).to(device)
        p = torch.rand((batch_size,))

        #get state indices to exploit based on epsilon
        exploit_indices = (p>epsilon).nonzero().flatten()

        #get actions
        actions[exploit_indices] = self.predict(primary_model, states_batch[
    exploit_indices]).argmax(dim=1)

        return actions

    def get_preds(self, states_batch, actions_batch, rewards_batch, next_states_batch
    ):

        #get batch size
        batch_size = len(states_batch)

        #get primary model predictions for next states
        primary_pred = self.predict(primary_model, next_states_batch)

        #identify best actions
        primary_best_actions = primary_pred.argmax(dim=1)

        #get target model predictions for next states
        target_pred = self.predict(target_model, next_states_batch)

        #select Q-values based on previously calculated actions
        target_best_qs = target_pred[torch.arange(batch_size), primary_best_actions]

        #calculate Q* (target)
        q_star = rewards_batch + self.gamma*target_best_qs

        #get primary model predictions for current states and current actions
        q_pred = self.predict(primary_model, states_batch)[torch.arange(batch_size),
    actions_batch]

        return [q_pred, q_star]

    def predict(self, model, states_batch):

        #get prediction and do not accumulate gradients
        with torch.no_grad():
            return model(states_batch)

class Fit():
    def __init__(self):

        #intialization
        self.nrm = 1
        self.optimizer = optim.Adam(primary_model.parameters(), lr=1e-3)

    def environment_step(self, y, primary_model, target_model):

        for i in range(0, 1763, 64):

            #get states batch
            states_batch = states[i:i+64]

            #get next states batch
            next_states_batch = next_states[i:i+64]

            #get y batch
            y_batch = y[i:i+64]

            #get actions
            actions_batch = agent.take_actions(states_batch)

            #get rewards
            rewards_batch = y_batch*(actions_batch-1)

            #multiply rewards by Negative Rewards Multiplier (NRM)
            rewards_batch[rewards_batch<0] = rewards_batch[rewards_batch<0]*self.nrm

            #get predictions and targets
            y_pred, target = agent.get_preds(states_batch, actions_batch,
    rewards_batch, next_states_batch)

            #calculate loss
            priorities = abs(target - y_pred) + (1-epsilon)

            #add quintuple to replay buffer
            replay.add([states_batch, actions_batch, rewards_batch, next_states_batch
    , priorities])

    def update_step(self, primary_model, target_model):

        #initialization
        error = []

        #get probabilities
        probs = replay.get_probs()

        for update in range(UPDATES):

            #sample from replay buffer
            states_batch, actions_batch, rewards_batch, next_states_batch = replay.
    sample(probs)

            #get primary model predictions for next states
            primary_pred = agent.predict(primary_model, next_states_batch)

            #identify best actions
            primary_best_actions = primary_pred.argmax(dim=1)

            #get target model predictions for next states
            target_pred = target_model(next_states_batch)

            #select Q-values based on previously calculated actions
            target_best_qs = target_pred[torch.arange(64), primary_best_actions]

            #calculate Q* (target)
            q_star = rewards_batch + agent.gamma*target_best_qs

            #get primary model predictions for current states
            q_pred = primary_model(states_batch)

            #get Q* in the correct form
            q_star_matrix = q_pred.clone()
            q_star_matrix[torch.arange(64), actions_batch] = q_star

            #set gradient equal to zero to prevent unwanted accumulation of gradients
            self.optimizer.zero_grad()

            #get loss
            loss = loss_function(q_pred, q_star_matrix)

            #append error
            error.append(loss)

            #calculate gradients for each layer in model (backwards fashion)
            loss.backward()

            #update model with specified update method
            self.optimizer.step()

            #update target network
            if update%99 == 0:

                #copy primary model to target model
                target_model = copy.deepcopy(primary_model)

        #return average error as tensor and as a percentage
        return torch.stack(error, dim=0).mean()*100

#PARAMETER SELECTION-----------------------------------------------------------

RUNS = 1
EPOCHS = 1000
UPDATES = 1000

EPSEXP = 0.15

USE_CUDA = True

#PARAMETER SELECTION-----------------------------------------------------------

#use either GPU or CPU
if USE_CUDA and torch.cuda.is_available():
    device = torch.device('cuda')
    print("GPU is used!")
else:
    device = torch.device('cpu')
    print("CPU is used!")

#import X and y
X_train = load("C:\\Users\\Frits\\OneDrive\\Desktop\\University of Twente\\AM Thesis
    \\Zanders\\Prediction Program\\Data\\Train\\^GSPC_X.joblib").float().to(device)
y_train = torch.squeeze(load("C:\\Users\\Frits\\OneDrive\\Desktop\\University of
    Twente\\AM Thesis\\Zanders\\Prediction Program\\Data\\Train\\^GSPC_y.joblib")).
    float().to(device)*100
X_test = load("C:\\Users\\Frits\\OneDrive\\Desktop\\University of Twente\\AM Thesis\\
    Zanders\\Prediction Program\\Data\\Test\\^GSPC_X.joblib").float().to(device)
y_test = torch.squeeze(load("C:\\Users\\Frits\\OneDrive\\Desktop\\University of
    Twente\\AM Thesis\\Zanders\\Prediction Program\\Data\\Test\\^GSPC_y.joblib")).
    float().to(device)*100

```

```
#initialize loss function
loss_function = nn.MSELoss()

#calculate base used for exponential decay
eps_base = EPSEXP**(1/math.floor(EPOCHS/2))

#truncate X- and y-train for calculation (the last state can never transition into a
    new state)
states = X_train[:-1]
next_states = X_train[1:]
y = y_train[:-1]

#initialize dataframe to store rewards
df_rewards = pd.DataFrame(np.arange(127), index=np.arange(127), columns=["Index"])

#register start time for iteration
stime = time.time()

for run in range(RUNS):

    #initialization
    primary_model = CNN(84).to(device)
    target_model = copy.deepcopy(primary_model)
    replay = ExperienceReplay()
    agent = Agent()
    fit = Fit()
    train_errors = []
    test_rewards = []

    for epoch in tqdm(range(EPOCHS)):

        #get exponentially decaying epsilon with minimum exploration rate of 10%
        epsilon = max(eps_base**epoch, 0.1)

        #take environment step
        fit.environment_step(y, primary_model, target_model)

        #take update step and store train errors
        train_errors.append(fit.update_step(primary_model, target_model).cpu().detach
    ().numpy())

        #initialization
        rewards = []

        #do not accumulate gradients
        with torch.no_grad():
            for x_, y_ in zip(X_test, y_test):

                #get action given X-test
                action = primary_model(x_).argmax(dim=0) - 1

                #get reward
                reward = y_*action

                #append rewards
                rewards.append(reward.cpu().detach().item())

        #clear memory
        torch.cuda.empty_cache()

        #store test rewards
        test_rewards.append(np.mean(rewards))

    #store model
    #dump(primary_model, f'PDN\\model_{run}.joblib')

    #store rewards
    df_rewards[f"RUN_{run}"] = rewards

#plot results
plot(train_errors)
plot(test_rewards, title="Average Rewards", label="Average Reward", xlabel="Epochs",
    ylabel="Average Rewards (%)")
plot(rewards, title="Rewards (01-01-2020, 30-06-2020)", label="Reward", xlabel="Days"
    , ylabel="Rewards (%)")

#write df to excel
df_rewards.to_excel('df_PDN_Rewards.xlsx')

print(f"Total computation time: {time.time()-stime}s")
```

D Average Geometric Return with Error Bars

Type	Experiment 1 (%)	Experiment 2 (%)	Experiment 3 (%)	Experiment 4 (%)
DDQN*	[-49.20, -0.99, 81.44]	[-25.92, -1.66, 44.20]	[-4.41, 6.55, 65.83]	[19.30, 26.09, 35.49]
PRD-DDQN*	[-14.02, -3.70, 11.72]	[-23.04, -0.14, 30.09]	[-3.45, -3.29, -3.27]	[27.71, 28.00, 30.09]
DDQN	[-45.11, -4.84, 54.12]	[-26.53, -0.76, 31.24]	[-25.53, 0.35, 54.12]	[2.88, 22.64, 30.87]
PRD-DDQN	[-44.04, 2.49, 103.67]	[-27.39, -1.82, 33.87]	[-44.04, 4.59, 75.85]	[-17.93, 1.36, 27.70]
S&P500	-4.04	26.89	-4.04	26.89

Table 8: Geometric Returns per Model Type and Experiment: the triplet format denotes the lowest, average, and highest realized returns across all models of the same type. DDQN* and PRD-DDQN* represent untrained model types, while DDQN and PRD-DDQN represent trained model types. S&P500 refers to the daily long position strategy on the S&P500.