UNIVERSITEIT LEIDEN

MASTER THESIS

Deep Reinforcement Learning in Portfolio Allocation

Author:
Jonathan IJBEMA

Supervisors:
Zhao YANG
Aske PLAAT

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Reinforcement Learning Group
Leiden Institute of Advanced Computer Science

May 8, 2025

Universiteit Leiden
The Netherlands

UNIVERSITEIT LEIDEN

Abstract

Faculty of Science
Leiden Institute of Advanced Computer Science

Master of Science

Deep Reinforcement Learning in Portfolio Allocation

by Jonathan IJBEMA

This thesis evaluates the application of Deep Reinforcement Learning (DRL) algorithms, specifically DDPG, PPO, and SAC, to the complex task of portfolio allocation. Instead of focusing on achieving optimal performance, this work offers a comprehensive analysis of key elements that influence and can enhance DRL agents in this domain, including the incorporation of lookback windows, contextual market information, advanced feature engineering, exploration strategies, and exposure to transaction costs. We have established a framework for assessing training stability and performance with statistical significance, a practice that is, in our experience, not yet standard in DRL applied in finance, and we stress the importance of mitigating biases in data and training methodology for unbiased experimentation. Through a series of carefully designed experiments, we address fundamental challenges inherent to employing DRL in finance, such as navigating high-dimensional state spaces, capturing long-term dependencies, ensuring sample efficiency, mitigating training instability, and achieving robust generalization. The findings reveal the learning capacity and generalization to out-of-sample data of off-policy algorithms (DDPG and SAC) when handling diverse portfolios. We also highlight the persistent challenge of outperforming simple benchmarks like BuyAndHold. In addition, this thesis emphasizes the significant challenge of handling order-invariant inputs using standard network architectures, together with ways to address this problem in the context of portfolio allocation. Ultimately, this thesis contributes with an empirical evaluation of DRL for portfolio optimization, providing valuable insights into its current capabilities and providing a roadmap for future research — most importantly an improved neural network architecture and reward function — aimed at enhancing the performance, robustness, and practical applicability of DRL-driven investment strategies.

Acknowledgements

First and foremost, I would like to share my deep appreciation for the guidance of my supervisors Zhao Yang and Aske Plaat during the writing process of this thesis. I have come to know both of them during the seminar on deep reinforcement learning, where they have instilled great curiosity in me for the field. Zhao has really aided me in shaping this thesis into its current form, always showing an interest in what was really going on. His high standards and meticulous vision pushed me to write something complete, something that truly demonstrates our shared knowledge and experience in the field of deep reinforcement learning and finance, while abiding by the norms of scientific research. Through his unpretentious expertise, Aske has guided us through the process by asking the right questions and ensuring that our ambitions remained focused and purposeful. I would also like to express my appreciation towards Thomas Moerland, who has connected me with Zhao in the initial stages of this project and provided me, together with Aske, with the necessary preliminary scholarship about reinforcement learning through their detailed courses at the faculty. I am also deeply grateful for the unequivocally valuable input from Koen Ripping and Jasper Kousen during this project, as it would not have been possible to write this thesis without them. Apart from providing me with the necessary data and resources, they have challenged my thinking and beliefs and given numerous pieces of advice and possible research directions along the way. Lastly, I can not express enough my sincere gratitude towards my parents, sister, and girlfriend for helping me become the person I am today. Without their support, I would not have been able to write a thesis at all, let alone a master thesis.

Contents

Chapter 1

Introduction

The world of finance has many applications that are critical to both individual investors and large financial institutions. Some of the key areas are risk and wealth management, portfolio allocation and optimization, market making, algorithmic trading, derivative pricing and hedging, and quantitative analysis. In many of these areas, machine learning algorithms are helping to make financial markets or investment strategies more stable, profitable and efficient. Machine learning models are able to process vast amounts of financial data, beyond any human capacity, and are better at extracting complex information and dependencies from that data than traditional methods (Rundo et al., 2019).

Machine learning is generally classified into three paradigms: Supervised learning, unsupervised learning, and reinforcement learning (RL). Supervised learning already dominates in financial settings and is used to tackle various problems, such as predicting stock prices, credit scoring, fraud detection and risk management (Dixon, Halperin, and Bilokon, 2020). Unsupervised learning is generally less popular, but can be very valuable, for example in financial forecasting (Corchado, Fyfe, and Lees, 1998). Meanwhile, reinforcement learning is gaining significant traction due to its ability to learn complex patterns from interacting with an environment in sequential decision-making problems, for example in market making and portfolio optimization (Osterrieder and GPT, 2023). A key approach within this domain is deep reinforcement learning (DRL), which uses deep neural networks for function approximation, allowing it to handle more complex tasks and environments effectively.

RL, like DRL, gained much of its popularity through successes in playing games. A breakthrough application of DRL was demonstrated by Mnih et al., 2015, where they trained a Deep Q-Network to successfully play multiple Atari 2600 games. Building on this success, many researchers continued to solve other games, like chess, shogi, and go with state-of-the-art algorithms like AlphaZero (Silver et al., 2018), Heads-up No-Limit Hold'em Poker (Moravčík et al., 2017), and to solve video games such as Starcraft (Vinyals et al., 2019).

RL has found numerous applications beyond gaming. Robotics is another prominent field (Plaat, 2022), where RL algorithms learn to control robot movements and achieve objectives. RL is also used in diverse domains such as education (Fahad Mon et al., 2023), healthcare (Yu et al., 2021), smart grid (Li et al., 2023), computer systems, business management and natural language processing (NLP) (Li, 2017). For example, ChatGPT and other large language models (LLMs) now extensively utilize reinforcement learning from human feedback (RLHF) to align them with human preferences.

While the finance sector is also adopting RL, its application remains in a relatively premature and exploratory phase, especially in published literature. Furthermore, the application of DRL to financial problems presents challenges such as high-dimensional state spaces, long-term dependencies, and limited data availability (Osterrieder and GPT, 2023). RL-specific challenges, such as poor generalization and high training variance, are particularly prevalent in financial applications, where the absence of rigorous statistical frameworks makes it difficult to determine whether the results represent a significant improvement over the state-of-the-art (Henderson et al., 2018).

This thesis explores the application of DRL algorithms to the portfolio optimization task using stocks from the S&P1500, which is an index comprising the 1,500 largest U.S. equities (S&P Dow Jones Indices, 1995). We propose an effective methodology for training DRL agents by employing a wide range of financial features and using random portfolio sampling. This approach addresses the challenge of limited data availability and establishes a statistically sound framework for evaluating DRL performance. Additionally, a consolidation layer is introduced to integrate these features with contextual market data and previous allocations.

By sampling a diverse range of assets, we mitigate the risk of cherry-picking a specific portfolio and reduce potential bias in our results. Our goal is to assess the training stability and generalization of DRL algorithms with continuous action spaces, specifically in their ability to learn profitable portfolio weights from inherently noisy financial data and adapt to evolving market dynamics. To this end, we focus on three widely used DRL methods: Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2015), Proximal Policy Optimization (PPO) (Schulman et al., 2017), and Soft Actor-Critic (SAC) (Haarnoja et al., 2018).

During the thesis, we aim to answer the following questions regarding DRL in portfolio allocation.

1. What is the impact of varying portfolio size on the learning stability and investment performance of DRL agents (DDPG, PPO, and SAC) in a financial setting? How well do these algorithms scale under the growing complexity in action and observation spaces?

2. How do different exploration strategies within DRL algorithms affect the learning and generalization capabilities of portfolio allocation agents when trained on deterministic historical financial data?

3. To what extent do transaction costs, contextual macroeconomic information, and historical feature lookback windows influence the behavior, performance, and robustness of deep reinforcement learning agents in portfolio optimization?

4. What are the key architectural considerations, particularly regarding permutation equivariance, for designing effective and robust DRL models for portfolio allocation when dealing with randomly sampled sets of assets? What is the influence of an attention mechanism (Vaswani, 2017) on permutation equivariance?

5. How do DRL algorithms compare to traditional benchmark strategies, such as the buy and hold strategy, an equal-weighted portfolio, and the global minimum variance portfolio (Markowitz, 1952)?

Through these questions, we hope to provide a comprehensive and thorough view of the current standing of DRL in portfolio allocation. In the remainder of this

introduction, we discuss the concept of reinforcement learning, with its applications and challenges, in Section 1.1. In Section 1.2, we introduce the specific finance application this thesis tries to tackle, namely the portfolio allocation problem. In Section 1.3, we discuss the existing literature on applying DRL to the portfolio allocation task. Lastly, in Section 1.4, we present our contribution to this field of research.

1.1 Reinforcement Learning

Reinforcement learning (RL) is a framework to teach algorithms to solve decision-making problems, closely resembling the way humans learn. It uses principles from behavioral psychology (Sutton, 2018), rewarding desirable actions and punishing unwanted behavior. More formally, it is a paradigm in machine learning (ML), besides supervised and unsupervised learning, concerned with making sequential decisions in an uncertain environment with the view of realizing some kind of objective.

In RL, an agent is the decision maker. It learns to interact with its environment by taking actions based on the currently available information (the state), so as to maximize some notion of cumulative rewards. The states and rewards are provided by the environment. This interaction is visualized in Figure 1.1. Over time, the agent learns a decision-making rule, or a policy, that maps the available information to the desirable behaviors in order to maximize its expected cumulative reward over a span of time.

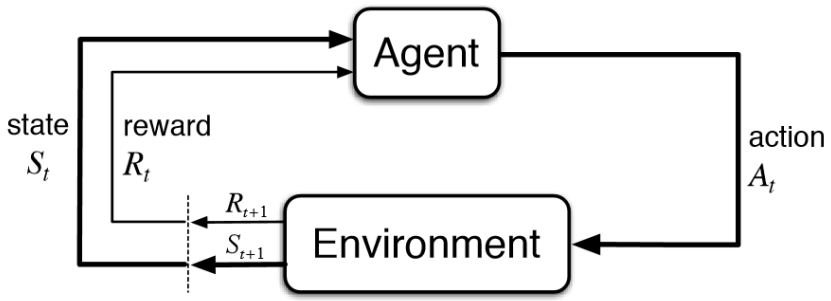


FIGURE 1.1: The agent–environment interaction in reinforcement learning (Sutton, 2018).

Initially, RL algorithms focused on solving problems with discrete state and action spaces, often represented in tabular forms. For instance, in a discrete state setup, the weather could be either sunny or rainy, and temperature might be categorized into one of ten distinct ranges. These tabular RL methods, such as Q-learning (Watkins and Dayan, 1992), provide a strong basis for the field.

However, tabular methods suffer from the curse of dimensionality (Bellman, 1957). This means that as the complexity of the environment grows, with larger or continuous state and action spaces, these methods become computationally infeasible. Categorizing temperature into discrete bins, for example, is only practical up to a certain point.

To overcome these limitations and tackle more complex, real-world problems, researchers have turned to function approximation techniques, and in particular, deep neural networks. This has led to the rise of deep reinforcement learning (DRL) algorithms. Instead of a table, DRL algorithms employ a deep neural network to

map states to actions. One of the major DRL algorithms is Deep Q-Networks (DQN) (Mnih et al., 2015). DQN combines Q-learning with deep convolutional neural networks to handle high-dimensional sensory inputs, achieving human-level performance in playing Atari games. However, DQN still uses discrete actions.

Value-based algorithms, such as DQN, aim to learn how promising each action is in a given state. Another significant branch of DRL algorithms constitutes the policy gradient methods, with REINFORCE (Williams, 1992) as one of the more prominent versions. Instead of estimating how good each action is, these algorithms learn by directly adjusting their behavior based on the rewards they receive. Algorithms like REINFORCE prove especially valuable for problems with continuous actions, where precisely controlling things like robot joint angles or steering adjustments is crucial. However, the learning process of REINFORCE can be quite unstable.

To mitigate these issues and combine the benefits of both value-based and policy-based methods, Actor-Critic algorithms have been developed (Sutton et al., 1999). Actor-Critic methods utilize two networks: an actor network that learns the policy and a critic network that estimates the value function, which helps with more stable learning. Prominent examples of Actor-Critic algorithms are A2C (Wu et al., 2017), A3C (Mnih et al., 2016), and later advancements like PPO (Schulman et al., 2017) and SAC (Haarnoja et al., 2018). These algorithms can handle increasingly complex and high-dimensional state and action spaces, and have subsequently been applied in difficult tasks in game AI, robotics (Tang et al., 2024), healthcare (Yu et al., 2021) and finance (Hambly, Xu, and Yang, 2023).

1.1.1 Applications

RL algorithms are probably most famous for their applications in game AI. DeepMind's AlphaGo, and its successors AlphaGo Zero and AlphaZero (Silver et al., 2018), have defeated world champions in the game of Chess and Go. These systems have demonstrated the ability of RL agents to master complex strategic reasoning and long-term planning. OpenAI Five (Berner et al., 2019) achieved superhuman performance in the complex multi-agent game of Dota 2. Vinyals et al., 2019 introduce AlphaStar, an agent which achieved a grandmaster status in the multiplayer game Starcraft. RL has also shown competence in Heads-up No-Limit Hold'em Poker (Moravčík et al., 2017). These breakthroughs show RL's capability in environments with large state and action spaces, with limited information and with other (competing) agents.

Beyond games, RL has made significant progress in robotics (Kober, Bagnell, and Peters, 2013). RL enables robots to learn complex motor skills, navigate intricate environments, and perform manipulation tasks. As gathering real-world data is expensive and RL algorithms are often quite sample inefficient, most agents are trained in simulated environments. Research in this area is consequently also focused on transferring learned policies in simulations to the real world (Zhao, Queralta, and Westerlund, 2020). In autonomous systems, particularly in autonomous driving, RL is also being explored to develop intelligent agents that can make driving decisions (Kiran et al., 2021).

Other prominent areas benefiting from RL are recommendation systems (Afsar, Crump, and Far, 2022), healthcare (Yu et al., 2021), energy systems (Zhang, Zhang, and Qiu, 2019), logistics and supply chain (Yan et al., 2022) and finance (Hambly, Xu, and Yang, 2023), where it has been applied to portfolio optimization and risk management.

1.1.2 Challenges

Despite its successes, RL faces several significant challenges, which are also very relevant to this thesis. One of the major challenges is their sample inefficiency, as RL algorithms often require a vast amount of interaction data with the environment to learn effectively (Sutton, 2018). In real-world scenarios this is most problematic, as gathering data can be expensive or dangerous.

Another challenge is dealing with sparse rewards (Ng, Harada, and Russell, 1999). In many realistic scenarios, rewards are either infrequent or delayed, making it difficult for the agent to learn which actions are truly contributing to achieving the goal. It comes down to designing effective and informative reward functions to tackle this challenge.

Training stability is another practical challenge. RL algorithms, particularly deep RL algorithms, can be sensitive to hyperparameter settings and prone to instability during training (Henderson et al., 2018). Furthermore, due to the absence of robust statistical frameworks and significance tests, it remains challenging to assess whether the observed improvements over the state-of-the-art are truly meaningful. This lack of statistical rigor is also evident in RL applications within finance.

Lastly, generalization remains a key challenge. RL agents trained in one environment may not generalize well to new, unseen environments or slightly different tasks (Cobbe et al., 2019). Developing RL algorithms that can generalize effectively is very important for real-world deployment where environments are often non-stationary and unpredictable. In finance, generalization is especially useful, as agents are trained, validated, and tested on different sections of the dataset and market dynamics might change over time.

Current research aims to improve upon these challenges. For example, sample efficiency is tackled through techniques like model-based RL (Moerland et al., 2023), meta-learning (Huisman, Van Rijn, and Plaat, 2021; Hospedales et al., 2021), and transfer learning (Zhu et al., 2023). Model-based RL builds a model of the environment to learn from, meta-learning aims to make algorithms better at learning in general, and transfer learning involves training an agent on one task and then adapting its skills for another, allowing it to leverage previously acquired knowledge in new settings, such as the real-world robot that has learned from a simulation. Research on addressing sparse rewards is exploring methods like intrinsic motivation, hierarchical RL, and curriculum learning. These methods improve reward signals or break up learning into multiple, more manageable tasks.

To summarize, while RL still faces challenges such as sample inefficiency, sparse rewards, training instability, and generalization, ongoing research is continually engaged in addressing them. In this thesis, we contend with these challenges through our framework for training and evaluating agents in the context of portfolio allocation in finance. Specifically, we address the challenge of sample inefficiency by using a comprehensive dataset of engineered financial features and enhancing data efficiency through random portfolio sampling from a large universe of stocks. To address training instability and promote generalization, we use a separate validation set and test on a substantial set of unseen assets to ensure robust out-of-sample performance (Packer et al., 2018). Our evaluation is conducted within a rigorous statistical framework, allowing for a clear analysis of the implications and impacts of particular design choices.

With our current understanding of RL, we can turn to the task on which we will experiment, namely the problem of portfolio allocation.

1.2 Portfolio Allocation

Portfolio allocation is a fundamental problem in finance, focused on profitably allocating investment capital across a set of available assets. These assets can be bonds, stocks, derivatives, currencies, or any other financial instrument. The primary goal is to construct a portfolio that aligns with an investor's specific financial objectives, typically aiming to maximize returns while taking into account risk.

In financial literature, researchers have tried to approach the portfolio allocation problem from different angles. Classic approaches represented by Modern Portfolio Theory (MPT) (Markowitz, 1952) search for the optimal allocation that maximizes expected returns for a given level of risk, typically measured by the variance of the portfolio. Although MPT provides the groundwork for understanding the risk-return trade-off, it builds on assumptions that are not necessarily valid in practice. Some of these assumptions include normally distributed returns, that investors are rational, and that market conditions are stationary. Furthermore, MPT does not incorporate transaction costs.

Later advances in portfolio optimization involve dynamic strategies that adapt to shifting market conditions. Merton's continuous-time portfolio problem, for example, introduced stochastic control for dynamic rebalancing (Merton, 1969). With this framework, we can model changes in wealth and time-varying risk.

RL is a relatively recent approach to portfolio allocation, but it appears very promising (Hambly, Xu, and Yang, 2023). The portfolio allocation task, which aims to optimize the risk-return trade-off sequentially, closely aligns with the RL objective of maximizing cumulative rewards by optimizing a sequence of returns (Sato, 2019). Model-free RL methods enable agents to learn adaptive policies from market interactions without predefined models of return distributions. Since deep RL algorithms can support highly dimensional state and action spaces, we can build significantly large portfolios with hundreds of financial features to characterize the health and potential of assets, along with market regimes. We can also model high-order nonlinear relationships between different assets using neural networks. Another key advantage of DRL is its potential to support multiple objectives and constraints simply by shaping the reward function. This enables us to optimize any risk-reward trade-off while including transaction costs, risk management measures, and liquidity constraints. The following section examines the current state of (D)RL algorithms in finance and explores the various approaches used to train them in this domain.

1.3 Literature Review

The portfolio allocation problem has been approached with various methods ranging from classical financial theory to advanced computational techniques. Also, machine learning (ML) is being adopted to enhance portfolio management strategies. This comes mainly from its ability to discern complex, nonlinear patterns within often large financial datasets. Alongside supervised and unsupervised learning as ML paradigms, (deep) reinforcement learning (DRL) is quickly gaining traction in financial applications (Hambly, Xu, and Yang, 2023).

However, applying RL to such complex problems as determining optimal portfolio allocations comes with various challenges and considerations. The most common challenges include high-dimensional state and action spaces (especially when the number of assets in the portfolio and the features per asset grow), long-term

dependencies, and limited data availability (Osterrieder and GPT, 2023). To address those problems, researchers have to consider the differences between various (D)RL algorithms, state spaces, action spaces, reward functions, and model architectures. Another crucial step in this process involves selecting suitable financial assets, including stocks, bonds, derivatives, or (crypto-)currencies, and curating historical data for training the algorithms.

In this section, we provide a comprehensive overview of design choices when applying DRL to the portfolio allocation task. Table 1.1, summarizes the most important aspects discussed in this section.

Both value-based, policy-based, and actor-critic algorithms have been applied to the portfolio optimization task, using various asset classes. In particular, value-based methods, such as Q-learning (Du, Zhai, and Lv, 2016; Pendharkar and Cusatis, 2018), SARSA (Pendharkar and Cusatis, 2018), and DQN (Zhang, Zohren, and Roberts, 2019; Chakraborty, 2019; Park, Sim, and Choi, 2020; Carta et al., 2021; Zejnullahu, Moser, and Osterrieder, 2022; Li and Hai, 2024), have shown promising results. These algorithms typically employ discrete action spaces, where actions correspond to distinct investment decisions. Common examples include "Buy", "Hold", and "Sell" actions or target orders (Huang, 2018) that directly translate to specific portfolio positions. Given their nature as tabular solutions, Q-learning and SARSA necessitate the use of discrete state spaces, where states are distinct entries within the Q-table. As DQN can handle continuous high-dimensional state spaces, we can directly input asset prices and other continuous features.

With policy-based and actor-critic algorithms, we can use continuous action spaces. This allows the algorithm to directly output the portfolio weight vector. Several studies have successfully applied policy-based methods (Jiang, Xu, and Liang, 2017; Zhang, Zohren, and Roberts, 2019; Benhamou et al., 2021) and actor-critic methods, such as A2C (Zhang, Zohren, and Roberts, 2019; Yang et al., 2020; Lu, 2023), DDPG (Liu et al., 2018; Yang et al., 2020; Jang and Seong, 2023; Lu, 2023), PPO (Yang et al., 2020; Lu, 2023; Sood et al., 2023; Jama, 2023), TD3 (Lu, 2023), and SAC (Lu, 2023) to the portfolio optimization task.

Various reward functions have been adopted in the literature. Among the most common variants are direct profits and losses (PnL) (Du, Zhai, and Lv, 2016; Liu et al., 2018; Chakraborty, 2019; Zhang, Zohren, and Roberts, 2019), portfolio returns (Li and Hai, 2024; Pendharkar and Cusatis, 2018), the (differential) Sharpe ratio (Benhamou et al., 2021; Sood et al., 2023; Jama, 2023), logarithmic portfolio returns (Jiang, Xu, and Liang, 2017; Jang and Seong, 2023), the Kelly criterion (Lu, 2023), and the value-at-risk (Jama, 2023). Some of these reward functions directly optimize the total return, while others also take risk into account. Although direct comparisons of identical agents across various reward functions are scarce in the literature, the observed performance differences remain minimal, with a slight tendency favoring risk-adjusted reward functions.

In financial settings, where the environment is inherently complex and noisy, the formation of high-quality observations is crucial for effective RL. Although most articles utilize open, high, low, close (OHLC) pricing data (Jiang, Xu, and Liang, 2017; Liu et al., 2018; Lu, 2023; Jama, 2023; Li and Hai, 2024), extracting valuable signals from raw prices can be challenging. To address this, Chakraborty, 2019 propose the use of feature engineering to derive technical indicators, such as the MACD, RSI, Williams %R, and asset return series, which can facilitate the extraction of meaningful information from the asset price series by (D)RL algorithms (Zhang, Zohren, and Roberts, 2019; Benhamou et al., 2021; Jang and Seong, 2023). Others also include

TABLE 1.1: Overview of methods and techniques in DRL for portfolio allocation.

Aspect	Details
ALGORITHMS	- **Value-based methods**: Q-learning (Du, Zhai, and Lv, 2016; Pendharkar and Cusatis, 2018), SARSA (Pendharkar and Cusatis, 2018), DQN (Zhang, Zohren, and Roberts, 2019; Chakraborty, 2019; Park, Sim, and Choi, 2020; Carta et al., 2021; Zejnullahu, Moser, and Osterrieder, 2022; Li and Hai, 2024). - **Policy-based methods**: Direct output of portfolio weight vector (Jiang, Xu, and Liang, 2017; Zhang, Zohren, and Roberts, 2019; Benhamou et al., 2021). - **Actor-critic methods**: A2C (Zhang, Zohren, and Roberts, 2019; Yang et al., 2020; Lu, 2023), DDPG (Liu et al., 2018; Yang et al., 2020; Jang and Seong, 2023; Lu, 2023), PPO (Yang et al., 2020; Lu, 2023; Sood et al., 2023; Jama, 2023), TD3 (Lu, 2023), SAC (Lu, 2023).
REWARD FUNCTIONS	- Profit and Loss (PnL) (Du, Zhai, and Lv, 2016; Liu et al., 2018; Chakraborty, 2019; Zhang, Zohren, and Roberts, 2019) - Portfolio returns (Li and Hai, 2024; Pendharkar and Cusatis, 2018) - (Differential) Sharpe ratio (Benhamou et al., 2021; Sood et al., 2023; Jama, 2023) - Logarithmic returns (Jiang, Xu, and Liang, 2017; Jang and Seong, 2023) - Kelly criterion (Lu, 2023) - Value-at-risk (Jama, 2023).
OBSERVATIONS	- Use of OHLC pricing data (Jiang, Xu, and Liang, 2017; Liu et al., 2018; Lu, 2023; Jama, 2023; Li and Hai, 2024). - Technical indicators (e.g., MACD, RSI) (Chakraborty, 2019; Zhang, Zohren, and Roberts, 2019; Benhamou et al., 2021; Jang and Seong, 2023). - Asset correlations (Benhamou et al., 2021; Benhamou, 2023; Jang and Seong, 2023; Li and Hai, 2024) - Previous weights (Jiang, Xu, and Liang, 2017; Liu et al., 2018; Chakraborty, 2019; Benhamou et al., 2021; Benhamou, 2023; Sood et al., 2023; Lu, 2023; Jama, 2023). - Contextual market data (e.g., indices, macroeconomic variables) (Benhamou et al., 2021; Benhamou, 2023; Sood et al., 2023; Li and Hai, 2024).
NORMALIZATION TECHNIQUES	- Min-max scaling or Z-score normalization to ensure stability and robustness (Patro, 2015).
MODEL ARCHITECTURES	- Time series models: RNNs, LSTMs (Chakraborty, 2019; Zhang, Zohren, and Roberts, 2019), GRUs. - CNNs for temporal pattern extraction (Benhamou et al., 2021). - Ensemble strategies combining multiple agents (e.g., PPO, A2C, DDPG) or classifiers (Yang et al., 2020; Carta et al., 2021).
FINANCIAL ASSETS	- Stocks in various markets (e.g., S&P500 (Sood et al., 2023), SSE (Li and Hai, 2024), DJIA (Liu et al., 2018)). - Forex market (currency conversions) (Chakraborty, 2019). - Cryptocurrencies (Jiang, Xu, and Liang, 2017). - Futures contracts in commodity and equity index markets (Zhang, Zohren, and Roberts, 2019).

asset correlations to include dependencies between elements in the portfolio (Benhamou et al., 2021; Benhamou, 2023; Jang and Seong, 2023; Li and Hai, 2024). To give agents a sense of the impact of their previous actions on the reward, especially when dealing with transaction costs, many researchers include the previous weights in the observations (Jiang, Xu, and Liang, 2017; Liu et al., 2018; Chakraborty, 2019; Benhamou et al., 2021; Benhamou, 2023; Sood et al., 2023; Lu, 2023; Jama, 2023).

Other features try to capture the state and risk of the market by including contextual market data, such as market indices, macroeconomic variables, and volatility indices (Benhamou et al., 2021; Benhamou, 2023; Sood et al., 2023; Li and Hai, 2024). Normalization of input data is important to ensure learning stability and robustness in machine learning settings (Patro, 2015), and most articles do this using min-max scaling or a Z-score.

To effectively capture the inherent complexities of financial data, researchers often employ sophisticated model architectures and training techniques. A frequent strategy involves utilizing time series of pricing or feature data as input and leveraging recurrent neural networks (RNNs), such as vanilla RNNs, Long Short-Term Memory Units (LSTMs), or Gated Recurrent Units (GRUs), or convolutional neural networks (CNNs) to extract temporal patterns. Chakraborty, 2019 and Zhang, Zohren, and Roberts, 2019 use LSTMs to extract sequential data relations from the time series of several technical indicators and pricing series. Benhamou et al., 2021 also use LSTMs and compare them with CNNs, showing that CNNs perform the best. Jiang, Xu, and Liang, 2017 introduce the Ensemble of Identical Independent Evaluators (EIIE) topology, where independent networks, sharing parameters, are employed for each asset. The authors compare the performance of RNNs, CNNs, and LSTMs within this framework, observing that CNNs and RNNs perform particularly well.

Other articles use more advanced setups, combining several agents into an ensemble. Yang et al., 2020 introduce an ensemble strategy that combines three actor-critic-based algorithms (PPO, A2C, and DDPG) to create a robust automated stock trading system. Carta et al., 2021 develop a sophisticated stock trading model that integrates multiple layers of deep neural networks and an ensemble of meta-learner classifiers to generate trading signals and improve trading decisions. Li and Hai, 2024 introduce CMPS, a model for managing a portfolio of stocks that uses a three-agent structure with a self-attention mechanism to make informed trading decisions.

Finally, these algorithms are applied to various financial assets and markets. Chakraborty, 2019 focus on the Forex market, focusing on currency conversion data, applying their algorithm to 10 different currency conversions. Jiang, Xu, and Liang, 2017 train and evaluate their agents on the cryptocurrency market. Zhang, Zohren, and Roberts, 2019 focus on futures contracts in the commodity, Forex, fixed income, and equity index markets. When focusing on stocks as a financial instrument, researchers cover a wide variety of markets. Li and Hai, 2024 focus on the China Shanghai Stock Exchange (SSE), comprising 34 stocks. Liu et al., 2018, Guan and Liu, 2022, and Jang and Seong, 2023 train and test their agents on the constituents of the Dow Jones 30 (DJIA) index, which consists of 30 US stocks. Sood et al., 2023 focus on stocks in the S&P500, which are the 500 largest companies in the US. Jama, 2023 analyzes DRL agents on the Swedish stock market, specifically on the OMXS30 index. Although all these algorithms are trained on real historical data, Lu, 2023 employ a price simulator based on correlated geometric Brownian motion to generate stock prices, along with the Bertsimas-Lo model to simulate market impact. This approach provides greater flexibility in evaluating algorithm performance, as it is not constrained by the availability of historical data.

Most of the papers reviewed use a single model for each configuration and perform backtesting on one portfolio over a span of several years. An exception to this approach is Sood et al., 2023, who use five agents and a sliding window technique to generate ten independent backtests. When evaluating a single trained model on

only one or a few portfolios, researchers are prone to selection bias, where a well-performing model and portfolio may be unintentionally favored. This makes it difficult to determine whether a proposed method genuinely adds value or if the results are merely due to random fluctuations or noise in the data, especially in the inherently volatile and noisy financial markets. Without statistical significance tests, it is hard to gauge the genuine performance of DRL on the portfolio allocation task.

To address this challenge Wang, 2019 sample thousands of portfolios of size $n =$ 20-100 from the S&P 500. Their approach provides the foundation for a robust statistical framework for assessing the performance of financial models more reliably.

In the next section, where we introduce our contribution to the literature, we also discuss how we build on the work of Wang, 2019 and aim to provide a real statistical foundation for evaluating DRL algorithms on the portfolio allocation task.

1.4 Our Contribution

Most reinforcement learning (RL) tasks use a simulator that simulates an environment, providing an endless stream of diverse and high-quality signals for an agent to train on. Games, for example, are an essentially bounded environment where the agent is free to take actions and states are produced organically.

In domains without an inherent data-generating environment, such as finance, healthcare, energy grid management, or supply chain optimization, curating relevant data for training machine learning models is essential. Some resort to simulate the dynamics of the real-world data, but this severely reduces the applicability in real-world situations, as we do not know the true data-generating process. For instance, in financial settings, simulating price series using Brownian motion can provide a game-like environment for training agents, but it will never take into account economic crises.

To achieve competitive performance in real financial markets, training on historical data currently seems the only option. But despite having only a limited and fixed dataset at our disposal, we can still successfully train our agent on real-world data. Offline reinforcement learning, for example, allows agents to learn from previously collected datasets without interacting with the environment in real-time (Levine et al., 2020). We can also create our own environment following the Gymnasium API (Towers et al., 2024) using historical data, much the same way as for more standard RL environments, allowing algorithms to learn online (Liu et al., 2020). Essentially, the environment acts as a wrapper around the financial market, simulating trading by sliding over historical data, while handling the algorithm's actions and calculating portfolio returns (Sood et al., 2023). We favor this simulated online approach over purely offline RL, because it enables the agent to interactively generate experience based on its evolving policy, reducing the need to curate transitions. Furthermore, it enables the use of standard online algorithms without requiring modifications to their core implementation.

Unfortunately, RL algorithms are not particularly efficient with data, often requiring datasets with millions of (state, action, reward, next state) tuples to properly learn a policy. Lu, 2023 have trained several deep reinforcement learning (DRL) agents on simulated data using geometric Brownian motion with the Bertsimas-Lo market impact model. They show that a PPO agent takes about 2M steps to learn the optimal policy, which translates to roughly 8000 years of daily data points.

To facilitate the extraction of valuable signals from raw financial data, experts often turn to feature engineering. This involves calculating the momentum and moving averages of the pricing data series, along with trading volume and other technical indicators such as volatility, relative strength index (RSI) (Wilder, 1978), and Bollinger Bands (Bollinger, 1992). The modeling of stock shares can be enhanced by incorporating features related to company fundamentals, analyst predictions, and market sentiment. These engineered features provide a deeper understanding of company potential, reducing the number of data points necessary for training (Chakraborty, 2019).

Furthermore, Wang, 2019 introduce another interesting avenue to improve data efficiency. They propose an RL framework for continuous-time mean-variance portfolio selection, proving the optimality of a Gaussian policy with time-decaying variance and establishing a policy improvement theorem. Their EMV algorithm, based on this theorem, demonstrates superior performance compared to traditional and DRL methods on S&P500 data. To improve data efficiency, they sample a portfolio of size $d = 50, 75$, and 100 from the total pool of 500 stocks in the S&P500, essentially providing them with $\binom{n}{d}$ different portfolios for training. They show that their EMV algorithm significantly outperforms model-free RL alternatives, such as DDPG.

Following the methodology proposed by Wang, 2019, we can benchmark the performance of RL algorithms on a diverse set of test portfolios. In contrast to studies that optimize a single, deterministic portfolio, the framework in Wang, 2019 enables the evaluation of RL algorithms across a wide range of portfolios. This approach improves statistical significance and mitigates the risk of data mining, backtest overfitting (Liu et al., 2020), and cherry-picking results.

Building on these foundational studies, our contribution can be summarized in five key areas:

- To address the problem of limited data availability in financial applications, we combine feature engineering and portfolio sampling from the works of Chakraborty, 2019 and Wang, 2019 into a new powerful training framework. Specifically, we sample portfolios of size 10 from the S&P1500, giving us $\binom{1500}{10} \approx$ 1.54e25 different portfolios. This allows our agents to learn across a diverse range of market conditions and asset combinations, leading to more robust and generalizable policies. Additionally, we craft 116 virtually uncorrelated features for each company in the S&P1500 [1], ranging from company fundamentals, technical indicators, analyst predictions, and market sentiment for over 20 years, providing us with almost a billion high-quality data points.

- Building on the EIIE topology from Jiang, Xu, and Liang, 2017, we train a single deep neural network to extract small embedding vectors, sharing weights across all companies. Unlike Jiang, Xu, and Liang, 2017, which handles portfolio company embeddings separately, we process them simultaneously in a consolidation layer alongside the previous allocation and contextual market data, capturing complex inter-company relationships and integrating them with the macroeconomic context.

- By randomly sampling portfolios, we evaluate RL agents across a wide range of portfolios, ensuring statistically significant results. To address RL training instability, we train multiple agents on the same configuration (Henderson et al., 2018). This approach allows us to analyze performance and behavioral

[1] In preliminary studies, we observed that the features are nearly uncorrelated, with most correlation values consistently below 0.1 in magnitude.

aspects such as portfolio turnover, stock diversity, lookback window length, market data incorporation, and training efficiency. We assess whether agents genuinely learn by comparing their performance with random and optimal policies, a factor often overlooked in existing literature. Using train, validation, and test sets, we also evaluate generalization to new data. In all of this, we uphold rigorous standards for processing financial data to minimize look-ahead and survivorship bias.

- Since different permutations of stocks within a portfolio do not convey additional information, we aim to make the company embedding vectors context-aware by employing an attention mechanism (Vaswani, 2017), which is inherently permutation-equivariant. This design ensures that the refined embedding vectors facilitate information processing in the consolidation layer, eliminating the need for it to learn that 10! possible portfolio permutations are fundamentally identical. This method draws inspiration from Tang and Ha, 2021, where RL agents are trained to process randomly permuted inputs from various RL tasks effectively.

- Using our framework, we benchmark our DRL agents against several strong baselines, including the buy and hold strategy, the equal-weighted portfolio, and the minimum variance portfolio (Markowitz, 1952). Although these algorithms are simple, they are often very effective and hard to outperform in the long run.

1.5 Thesis Outline

Before we dive into DRL algorithms in the portfolio allocation task, we will briefly introduce the rest of the chapters and their contents.

In Chapter 2, we will provide the necessary theoretical background on portfolio optimization, reinforcement learning, and deep learning techniques relevant to the following experiments. In Chapter 3, we will detail the DRL algorithms (DDPG, PPO, SAC), the financial environment and dataset used for training and evaluation, the network architecture, and the experimental design for each set of experiments. Chapter 4 presents the core experimental results. It begins by assessing the learning capabilities of the algorithms across varying portfolio sizes. It then examines the influence of exploration strategies on performance and the out-of-sample performance of the trained agents against several baseline strategies. The experiments thereafter investigate the permutation equivariance of the network, the impact of different transaction cost regimes, and the effect of incorporating contextual market information and a lookback window into the observation space. Building upon the findings of Chapter 4, Chapter 5 will summarize the key findings of the thesis, discuss their implications for the field of DRL-based portfolio allocation, and outline potential directions for future research based on the limitations and insights gained from this thesis.

Chapter 2

Background

This chapter provides the theoretical foundation for applying Deep Reinforcement Learning (DRL) to portfolio allocation. We begin by introducing reinforcement learning (RL) within the framework of a Markov Decision Process (MDP), detailing essential components and outlining how policies are formulated to maximize cumulative rewards via the Bellman equations. Next, we review the principal RL algorithms, including value-based, policy-based, and hybrid actor-critic methods.

We then transition to deep learning fundamentals, discussing neural network architectures, training via backpropagation, and optimization strategies, with a focus on specialized models such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) for processing financial data.

Finally, we discuss the portfolio allocation problem by presenting its mathematical formulation, introducing objective functions, utility measures, and constraints like transaction costs. We hope to establish a robust foundation for exploring DRL-based strategies in the following chapters.

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm that focuses on training agents to make sequences of decisions in an environment. In RL, an agent navigates an environment by taking actions, transitioning through states, and receiving rewards. The agent's objective is to learn an optimal strategy (a policy) that maximizes expected cumulative rewards over time.

In the following sections, we formally define RL using the Markov Decision Process (MDP), discuss policies that guide agent behavior, and introduce value functions that quantify the value of different states and actions. We also discuss the foundational types of RL algorithms, such as value-based algorithms, policy-based algorithms and actor-critic algorithms.

2.1.1 Markov Decision Process

We can express the agent-environment interaction, introduced in Section 1.1, more formally, using the definitions and notations introduced by Sutton, 2018. At each discrete time step $t = 1, 2, 3, \ldots$ in the environment, the agent observes a state s_t, performs actions a_t, and is given rewards r_t. The state s_t is a full representation of the environment at step t. In chess, for example, this would be the complete observable board position and the player to make the next move. The states s_t come from the set $\mathcal{S}$, the agent chooses its actions a_t from the total set of possible actions $\mathcal{A}$, and the rewards r_t that the environment produces in return come from a set $\mathcal{R}$.

Just like humans favor an ice-cream given now more than an ice-cream given after two years of waiting, RL has a way of balancing the importance of immediate

rewards versus future rewards. For this, we use a discount factor $\gamma \in [0, 1]$. Lower values of γ put a greater emphasis on immediate rewards, whereas values towards 1 put a higher value on future rewards. In the extremities, a discount factor of 0 considers only an immediate reward, and a value of 1 gives all rewards equal weight. In financial contexts, it is standard to set the discount factor to reflect the risk-free rate, so that at any time step, rewards received in the future are discounted by the opportunity cost of not investing them in a risk-free asset, such as a government bond.

Given these sets and the discount factor γ, we can define a mathematical process that formalizes the sequences of states, actions, and rewards. For this, we use the Markov Decision Process (MDP), and we can define it as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$, where:

- $\mathcal{S}$ represents the state space,
- $\mathcal{A}$ is the action space,
- $\mathcal{T}$ defines the transition function, or the set of probabilities $\mathcal{P}$ of moving from one state to the next state given an action,
- $\mathcal{R}$ denotes the reward function,
- $\gamma \in [0, 1]$ is the discount factor.

By definition, this process follows the Markov property. This means that the next state, denoted as s', is a function of only the current state (s) and the action (a), and not of previous states or actions. In that case, we are able to predict the next state and expected next reward given the current state and action (Sutton, 2018). This is an important concept in RL and sequential modeling, as we have to be able to model how our current behavior affects the future. If this transition would be random, we would never know how our actions shape the future.

2.1.2 Policies

Using the former terminology, we can formulate the behavior of an agent using a policy, or decision-making rule. A policy maps states to actions, such that the agent learns what action to take in each state. In RL, the goal of the agent is to find an optimal policy, i.e. one where the agent takes the most lucrative action in each state.

Formally, the agent needs to determine an optimal policy $\pi^*(a|s)$, that maximizes the expected cumulative discounted reward G_t at each step t,

$$\pi^* = \arg\max_{\pi} \mathbb{E}[G_t|\pi], \tag{2.1}$$

for a discrete time MDP with an infinite horizon, and where $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ is referred to as the return. In other words, the agent needs to find the behavior that ensures that we reach the highest attainable reward in the long run, discounted by γ.

Policies can be deterministic or stochastic. Deterministic policies map each state s to a fixed action a, ensuring that the same action is chosen for a given state every time. Stochastic policies map s to a probability distribution over actions a, allowing for variability in the actions taken, even for the same state. This randomness can be beneficial in exploration, enabling the agent to discover more diverse strategies without inducing randomness at the level of the actions.

2.1.3 Value Function

Maximizing the return G_t entails achieving the highest cumulative reward over a sequence of actions. To effectively maximize this return, we need to relate the return to the states and actions that contribute to it. This connection can be further refined by linking the return to the current state or action through the state value function and the action value function.

The state value function $V^\pi(s)$ denotes the expected return when starting in state s and subsequently following policy π. It is defined as

$$V^\pi(s) = \mathbb{E}_\pi[G_t|s_t = s]. \tag{2.2}$$

The action-value function, or Q-function, $Q^\pi(s,a)$, on the other hand, denotes the expected return when starting in state s, taking action a, and following policy π thereafter. It is defined as

$$Q^\pi(s,a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a]. \tag{2.3}$$

These functions measure how lucrative the current state and action are. For example, if we still have the queen on the board, the value of $V^\pi(s)$ will be higher than in a state where the queen is just blundered away. Also, $Q^\pi(s,a)$ of the action that inadvertently sacrificed the queen will be lower than one that leads to a checkmate in three.

The true applicability of the value function and the Q-function emerges when we recognize their recursive nature. As there exists a relationship between subsequent values of these functions, we can more easily estimate and optimize long-term rewards. Formally, the value function and Q-function can be recursively defined through the Bellman equations (Bellman, 1957). Through Equations 2.2 and 2.3, we can define the Bellman equation for the value function as

$$V^\pi(s) = \mathbb{E}_\pi[r_t + \gamma V^\pi(s_{t+1})|s_t = s], \tag{2.4}$$

and the Bellman equation for the Q-function as

$$Q^\pi(s,a) = \mathbb{E}_\pi[r_t + \gamma Q^\pi(s_{t+1}, a_{t+1})|s_t = s, a_t = a]. \tag{2.5}$$

In essence, these equations express the relationship between the value of a state (or state-action pair) and the values of the states coming after. This recursive relationship allows us to iteratively update our function estimates with observed actions, states, and rewards, gathered from experience by interacting with the environment. This phenomenon is useful for dynamic programming and forms the foundation of most RL algorithms.

Using these functions, we can define an optimal policy π^*, which achieves the optimal value function and optimal Q-function for all states and actions. The optimal value function is defined as

$$V^*(s) = \max_\pi V^\pi(s), \tag{2.6}$$

which represents the highest expected return of being in state s, given that the agent behaves optimally according to policy π^*. In the same way, the optimal Q-function represents the maximum achievable expected cumulative reward from taking action

a, while starting from state s,

$$Q^*(s,a) = \max_\pi Q^\pi(s,a). \tag{2.7}$$

Hence, the optimal policy π^* effectively maximizes the expected return in the given MDP, as it does so in each state and for each action. To find π^*, the RL algorithm needs to implicitly learn the transition probabilities from $\mathcal{T}$ and the reward function $\mathcal{R}$.

This foundation of MDPs, value functions, and Bellman equations provides the mathematical framework for understanding and developing RL algorithms. An RL algorithm can focus either on calculating the (action-)value function, which gives rise to value-based RL algorithms, or on optimizing the policy directly, as done in policy-based methods. Actor-Critic methods combine these two elements. We will explore these methods further in the upcoming sections.

Before we discuss value-based and policy-based algorithms, we have to introduce some important RL concepts. While learning the policy, RL algorithms can perform updates on-policy or off-policy. On-policy algorithms update the policy with transitions from the current policy iteration, while off-policy algorithms update the policy also with older, previously gathered transitions, for example from a replay buffer.

Another important distinction can be made between model-free and model-based methods, where the latter tries to build a model of the environment in order to improve learning. This thesis focuses exclusively on model-free reinforcement learning algorithms, as we hypothesize that building a model of financial markets is too complex. Model-based approaches, while relevant to the broader field, fall outside the scope of this particular research.

2.1.4 Value-Based Reinforcement Learning

Value-based algorithms aim to learn an optimal value function $V^*(s)$ or Q-function $Q^*(s,a)$ to guide decisions. By estimating the expected return G_t for each state or state-action pair, the agent can choose actions to maximize long-term (discounted) rewards for any given state. In other words, the agent's policy is derived from the learned (action-)value function and therefore makes decisions considering the expected cumulative rewards.

A classic value-based algorithm is Q-learning, which is an off-policy algorithm (updating the policy using older, previously gathered transitions) that learns iteratively given observed rewards and the maximum estimated Q-value of the next state (Watkins and Dayan, 1992). Q-learning relies on a table, often called a Q-table, for storing and updating the Q-values for each state-action pair. This process uses the relationship in Equation 2.5 to learn optimal policies in a model-free manner. It begins with a random initialization of the Q-table and iteratively updates it with the observed rewards, states, and actions gathered in the environment.

Using the relationship in Equation 2.5, we can formulate the update rule for Q-learning as

$$Q^{\text{new}}(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \left[r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t) \right], \tag{2.8}$$

where α is the learning rate and the term in the square brackets is coined the temporal difference (TD) error. The TD error measures the difference between the current Q-value and the TD target, which is the reward r_t attained after taking action a_t in

state s_t plus the best possible future value ($\max_{a'} Q(s_{t+1})$. The new Q-value is calculated by adjusting the current value with a weighted difference between the current Q-value and an estimate derived from the observed reward and the maximum Q-value of the next state s_{t+1} after taking action a_t.

Another popular value-based method is State-Action-Reward-State-Action (SARSA), an on-policy algorithm that updates the Q-function based on the current policy and the rewards observed (Rummery and Niranjan, 1994). The update rule for SARSA can be formulated as

$$Q^{\text{new}}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right], \tag{2.9}$$

where α is the learning rate, r_t is the reward received after taking action a_t in state s_t, and a_{t+1} is the next action chosen according to the current policy. The term within the square brackets again represents the TD error. In SARSA, the TD target is defined by the Q-value corresponding to the subsequent state-action pair (s_{t+1}, a_{t+1}), rather than the maximum Q-value over all possible actions. This update rule ensures that the learning process remains consistent with the current policy, making SARSA an on-policy algorithm.

However, the problem with a table is that it can not capture continuous states, as the dimensions grow too large. This caused the advent of Deep Q-Networks (DQNs), which extend Q-learning to high-dimensional state spaces with the use of deep neural networks to approximate the Q-function (Mnih et al., 2015). RL algorithms combined with neural networks from deep learning are called Deep Reinforcement Learning (DRL) algorithms. DRL is especially useful when states become high-dimensional and tabular solutions grow too complex. DQN's breakthrough has enabled significant advances in the application of RL to complex tasks, such as playing Atari games directly from high-dimensional pixel inputs, which would be impossible to model with a Q-table.

Value-based RL algorithms are particularly well-suited for environments featuring discrete action spaces, where actions fall into distinct categories, such as "Left" and "Right" or "Buy" and "Sell".

2.1.5 Policy-Based Reinforcement Learning

Instead of relying on value functions, policy-based algorithms directly optimize the policy. These methods optimize the policy parameters to maximize the expected return $J(\pi_\theta)$, which is defined as

$$J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \gamma^t r_t \right], \tag{2.10}$$

where τ represents a trajectory sampled from the policy π_θ. A trajectory is simply a sequence of states, actions, and rewards, and when it terminates it is called an episode.

The core idea behind policy-based methods lies in adjusting the policy parameters in a direction that increases the probability of selecting actions leading to higher returns. This optimization process, commonly referred to as the policy gradient, consists of an iterative update of the policy parameters proportionally to the gradient of the expected return:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\pi_\theta), \tag{2.11}$$

where θ represents the policy parameters, α is the learning rate, and $\nabla_\theta J(\pi_\theta)$ is the gradient of the expected return with respect to the policy parameters.

Using the policy gradient theorem (Sutton et al., 1999), we can express the gradient of $J(\pi_\theta)$ as

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)\hat{R}_t\right], \tag{2.12}$$

where $\hat{R}_t$ is an estimator of the return from time step t onward, often equal to G_t. The REINFORCE algorithm (Williams, 1992) estimates $\hat{R}_t$ using Monte Carlo simulations.

Policy-based methods have several advantages. First, they can handle continuous action spaces. Since the output of the policy network directly corresponds to the actions, which can be continuous, these methods are not limited to discrete action selections through Q-values, for example. The second advantage is that policy-based methods can learn stochastic policies. By injecting variability into the policy network itself, they offer a more nuanced approach towards exploration compared to adding stochasticity at the level of the actions via action noise. However, they can be sensitive to hyperparameter settings and can have high variance in the gradient estimator $\nabla_\theta J(\pi_\theta)$.

2.1.6 Actor-Critic Methods

To address the high variance issue, which slows down learning, actor-critic methods (Konda and Tsitsiklis, 1999) combine the strengths of value-based and policy-based approaches. They use an actor network to represent the policy $\pi_\theta(a|s)$, which outputs (continuous) actions, while a separately learned critic network approximates either the value function $V^\pi(s)$ or the action-value function $Q^\pi(s,a)$. We can then compute the advantage function using the estimated value function, and use the advantage in place of the raw return in updating the policy network. This mechanism reduces the variance in gradient updates.

To illustrate this, we can slightly modify the policy gradient representation from Equation 2.10 to

$$\nabla_\theta J(\pi_\theta) \approx \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^\pi(s_t,a_t)\right], \tag{2.13}$$

where we use the advantage function $A^\pi(s_t,a_t)$ instead of the return estimator $\hat{R}_t$. The advantage function can be approximated as $A^\pi(s_t,a_t) = Q^\pi(s_t,a_t) - V^\pi(s_t)$ or $R_t - V^\pi(s_t)$. Here, the critic estimates $V^\pi(s_t)$ or $Q^\pi(s_t,a_t)$, reducing the variance of the policy gradient as it centers the gradient estimates around zero. Now, the actor updates his policy parameters θ based on the feedback from the critic.

Actor-critic methods are the golden standard in most RL tasks nowadays. One of the first successful implementations was the Asynchronous Advantage Actor-Critic (A3C) (Mnih et al., 2016), which introduced an actor-critic architecture combined with asynchronous updates. Unlike its asynchronous counterpart, A2C (Wu et al., 2017) is a synchronous, deterministic variant of A3C. It uses multiple workers to avoid the use of a replay buffer.

Later, many more algorithms have been introduced, improving robustness and sample-efficiency in the process. Trust Region Policy Optimization (TRPO) (Schulman et al., 2015) provides constraints on policy updates, such that the new policy does not diverge from the old one too much. This improves stability significantly.

Proximal Policy Optimization (PPO) (Schulman et al., 2017) is another very popular RL algorithm. It generally provides a much more computationally efficient and scalable alternative to TRPO while maintaining stability. Soft Actor-Critic (SAC) (Haarnoja et al., 2018) incorporates entropy maximization into the objective to encourage exploration. This helps overcome premature convergence to suboptimal policies.

Other popular actor-critic algorithms are Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2015) and its improvement Twin Delayed DDPG (TD3) (Fujimoto, Hoof, and Meger, 2018). These algorithms rely on the deterministic policy gradient theorem and are effective in high-dimensional continuous action spaces. TD3 improves on many RL tasks by mitigating the Q-value overestimation bias (Van Hasselt, Guez, and Silver, 2016) through employing a pair of critics instead of one.

We will discuss some of these algorithms, specifically DDPG, PPO and SAC in more detail in the following chapters. In the next sections, we introduce deep learning, which is vital for function approximation in DRL algorithms.

2.2 Deep Learning

Since most state-of-the-art RL algorithms rely on deep learning networks to approximate value or policy functions, we provide a brief introduction to the concept in this section. Deep learning draws inspiration from the workings of the brain and neural networks, mimicking their layered structure and learning mechanisms (LeCun, Bengio, and Hinton, 2015). The primary motivation behind deep learning is to approximate any mathematical function through a model f, which maps inputs $\mathcal{X}$ to outputs $\mathcal{Y}$, parameterized by a large set of trainable parameters θ (Goodfellow, Bengio, and Courville, 2016). In this framework, deep neural networks serve as universal approximators, representing any function with an arbitrary number of nodes, layers, and non-linear transformations (Hornik, 1991).

In Figure 2.1, we present an illustration of a basic neural network with an input layer of dimension 5, several hidden layers, and an output layer of dimension 3. The lines between node j in layer $i-1$ and k in layer i have a weight $w_{i,jk}$ [1]. Not represented in the figure, but very common in neural networks, is a bias term $b_{i,k}$ for each node k in the layer i in the network, which essentially serves the same role as the intercept in a linear regression.

If we combine weights $w_{i,jk}$ into a matrix W_i and biases $b_{i,k}$ into a vector b_i, each layer i in the network represents a multivariate linear regression $W_i x + b_i$. On top, in most neural networks, each layer has an activation function A_i. Common activation functions, include tanh, softmax, and the rectified linear unit (ReLU) (Nair and Hinton, 2010), which enable the networks to model complex, non-linear relationships in data effectively. Using the activation function A_i, the output h_i of each layer can be formulated as

$$h_i = A_i(W_i x + b_i), \tag{2.14}$$

which is a non-linear relationship between inputs x and output h_i. If we stack n layers together, the output h_n of the final layer can be expressed as the composition of all intermediate layers:

$$h_n = A_n(W_n(A_{n-1}(W_{n-1}(\dots A_1(W_1 x + b_1)\dots) + b_{n-1}) + b_n). \tag{2.15}$$

[1] In this section, w or W specifically refer to neural network weights. In all subsequent sections and chapters, unless stated otherwise, $\mathbf{w}$ or w will denote portfolio weights.

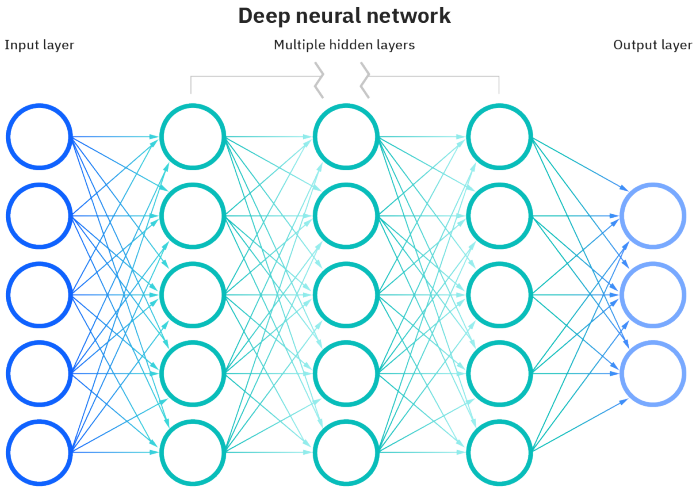


FIGURE 2.1: A basic neural network architecture with an input layer, multiple hidden layers, and an output layer (Lotfinejad, 2023). The input layer consists of 5 nodes and the output layer of 3 nodes. Each line between nodes has a trainable weight $w_{i,jk}$.

In deep learning, finding the optimal weights W, or, equivalently, the optimal parameters θ, is done by the backpropagation algorithm (Rumelhart, Hinton, and Williams, 1986). Backpropagation calculates the gradient of the loss function with respect to each parameter by applying the chain rule of differentiation, propagating errors backward through the network. These gradients are then used to update the weights in the direction that minimizes the loss, denoted as

$$\theta_{k+1} = \theta_k - \alpha \nabla_\theta L, \tag{2.16}$$

where θ_k are the parameters at step k, α is the learning rate, and $\nabla_\theta L$ is the gradient of the loss function L with respect to the parameters θ. Intuitively, we change the parameters each step with size α in the direction that minimizes some loss function.

In practice, this update step is done by optimization algorithms, such as stochastic gradient descent (SGD) (Robbins and Monro, 1951; Ruder, 2016), where parameters are updated iteratively using the gradient of the loss function computed on random subsets (batches) of the training data. Variants of SGD, such as momentum (Polyak, 1964), add a moving average of past gradients to the current update to accelerate convergence and avoid oscillations. Adaptive methods like AdaGrad (Duchi, Hazan, and Singer, 2011), RMSProp (Tieleman and Hinton, 2012), and Adam (Kingma and Ba, 2014) dynamically adjust the learning rate for each parameter based on historical gradient information, enabling faster convergence in complex, high-dimensional optimization landscapes.

2.2.1 Convolutional and Recurrent Neural Networks

Up to this point, we have only discussed fully connected architectures. However, deep learning offers more specialized network architectures for different data types and tasks. Two prominent examples are Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), which excel in processing spatial and sequential data, respectively.

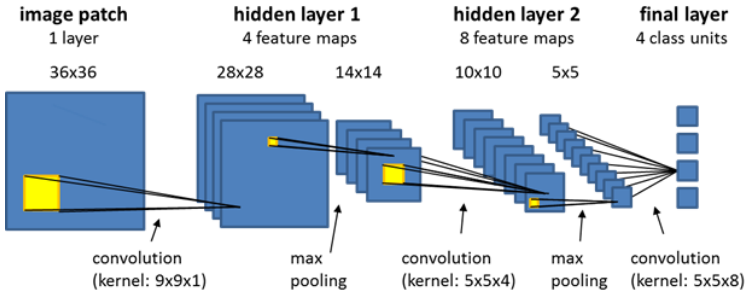


FIGURE 2.2: Schematic representation of a convolutional neural network (CNN) architecture designed for image classification.

Convolutional Neural Networks are particularly effective for analyzing grid-structured data, due to their ability to extract hierarchical spatial features. CNNs are known for their capabilities in analyzing images (2D), but they are also adapt at learning spatial temporal features in videos (3D) and recognizing patterns in time series (1D). Instead of connecting each neuron in a layer to every neuron in the next layer, as in fully connected networks, CNNs apply convolutional filters over local regions of the input. This approach reduces the number of trainable parameters, making CNNs well-suited for image recognition, object detection, and related tasks (LeCun et al., 1998; Krizhevsky, Sutskever, and Hinton, 2012). Furthermore, CNNs are translational invariant. This means that a CNN can still identify an object in an image, such as a chair, if its position is changed.

Key components of CNNs include convolutional layers, pooling layers for down-sampling, and fully connected layers for classification. Max-pooling, for example, is a permutation invariant operation which outputs the maximum value of a matrix of arbitrary size, effectively reducing the dimensionality of the processed data. In Figure 2.2, a basic CNN architecture with the aforementioned elements is depicted. The network processes a 36×36 input image patch through successive layers of convolution and max-pooling. The two hidden layers extract 4 and 8 feature maps, respectively. The final fully connected layer outputs 4 class units, which could represent the different object classes on the processed image that the algorithm should detect.

In financial applications, 1D CNNs are widely and successfully used to extract temporal patterns from the time series of features (Jiang, Xu, and Liang, 2017; Benhamou et al., 2021). Some also employ 2D or 3D convolutions, also capturing relationships between assets and financial features in the portfolio (Benhamou, 2023; Li and Hai, 2024). However, it is crucial to ensure that there is an intuitive spatial hierarchy in the data before applying a CNN. If the order of assets and features in the input data can be changed without altering the fundamental meaning and dynamics of the portfolio optimization task, then a spatial CNN might not be the most suitable feature extractor.

There exist other architectures to extract temporal features from time series data. Recurrent Neural Networks are specifically designed to do this. RNNs introduce feedback loops, enabling information from previous time steps to influence the current state. At each time step t, the hidden state h_t is updated based on the current input x_t and the previous hidden state h_{t-1}. We can formally denote this as

$$h_t = A(W_h h_{t-1} + W_x x_t + b), \tag{2.17}$$

where W_h and W_x are trainable weight matrices. This allows RNNs to effectively model sequences of data, such as pricing series.

However, vanilla RNNs suffer from vanishing or exploding gradients when learning long-term dependencies. To address this, architectures such as Long Short-Term Memory (LSTM) networks (Hochreiter, 1997) and Gated Recurrent Units (GRUs) (Cho, 2014) introduce gating mechanisms to control the flow of information and gradients more effectively.

Attention mechanisms, foundational to the Transformer architecture, have further improved processing of sequential data (Vaswani, 2017). They enable models to selectively focus on input parts. At its core, attention is a weighted sum, calculated in Transformers via scaled dot-product attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.18}$$

Here, queries Q, keys K, and values V are input matrices. The part inside the softmax function measures key-query similarity. The softmax function normalizes these similarities into attention weights for value weighting.

Through the use of attention, Transformers are better at modeling sequential data than RNNs. Their parallel nature also increases computational efficiency. Furthermore, the attention mechanism is permutation equivariant, which means that if we change the order of inputs in Q, K, and V, the output will change in the same way.

2.3 Portfolio Allocation

In this section, we build on the introduction in Section 1.2 and introduce a mathematical framework for the portfolio allocation task. The portfolio allocation problem, sometimes also called the portfolio management or optimization problem, is a well-studied foundation of financial decision-making. It aims to find the optimal capital distribution among a set of n assets. This formally means finding a vector of weights, $\mathbf{w_t} = (w_t^1, w_t^2, \ldots, w_t^n)$, where w_t^i denotes the proportion of total wealth allocated to asset i at time step t, with the goal of maximizing some defined objective function.

We can specify this objective function through the utility function $U(\cdot)$. The utility function is a function of the portfolio return $R_{p,t}$ at time t and it can be customized to the investor's risk preferences. For example, as higher returns most often mean higher volatility, some investors prefer more conservative risk adjusted utility functions.

In the portfolio allocation problem, the goal is to maximize the expected utility over a planning horizon T, discounted by a factor γ:

$$\max_{\{\mathbf{w_t}\}_{t=0}^{T}} \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t U(R_{p,t})\right], \tag{2.19}$$

where $\{\mathbf{w_t}\}_{t=0}^{T}$ denotes that we are optimizing the set of weights at each time step over the horizon T. We can decompose the portfolio return $R_{p,t}$ in terms of the asset returns $\mathbf{r_t} = (r_t^1, r_t^2, \ldots, r_t^n)$ [2] as

$$R_{p,t} = \mathbf{w_t}^\top \mathbf{r_t}. \tag{2.20}$$

[2] To avoid confusion on the meaning of r, we will be absolutely clear whether we are dealing with financial returns or RL rewards in the rest of this thesis.

Right now, we assume that the assets in the portfolio can be rebalanced at any time without incurring any costs. However, in practice, optimization must take into account transaction costs. If we adjust the portfolio weights from $\mathbf{w_{t-1}}$ to $\mathbf{w_t}$, we pay a fee often proportional to the difference between these weights. Mathematically, let $c(\mathbf{w_{t-1}}, \mathbf{w_t})$ be the transaction cost function. Then, the net portfolio return $\tilde{R}_{p,t}$ after accounting for transaction costs is

$$\tilde{R}_{p,t} = R_{p,t} - c(\mathbf{w_{t-1}}, \mathbf{w_t}). \tag{2.21}$$

After adding the transaction costs to the objective function, we then aim to maximize the expected utility of the net portfolio returns

$$\begin{aligned} \max_{\{\mathbf{w_t}\}_{t=0}^{T}} \quad & \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t U(\tilde{R}_{p,t})\right] \\ \text{s.t.} \quad & \sum_{i=1}^{n} w_t^i = 1 \\ & w_t^i \geq 0 \end{aligned} \tag{2.22}$$

where we constrain the weights to sum to one and be nonnegative, i.e. we do not allow for shortselling assets. There are many ways in which we can define the cost function $c(\mathbf{w_{t-1}}, \mathbf{w_t})$. We use a proportional transaction cost scheme, where transaction costs are a percentage of the amount transacted, as in Davis and Norman, 1990.

Chapter 3

Methodology

In this section, we detail the methodology of this master thesis. We introduce the deep reinforcement learning (DRL) algorithms used in the experiments, the baselines with which we compare our DRL algorithms, the performance metrics on which they will be compared, the neural network architecture of the agents, and the portfolio allocation environment with which they interact. We also introduce the data on which we train our algorithms, stressing the importance of careful handling of financial data to mitigate biases.

3.1 Continuous Deep Reinforcement Learning Algorithms

With our current understanding of reinforcement learning and deep learning, we can turn to scrutinizing the three DRL algorithms that are used for experiments in this thesis. Specifically, we use three of the most popular continuous-action DRL algorithms in finance, namely Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2015), Proximal Policy Optimization (PPO) (Schulman et al., 2017), and Soft Actor-Critic (SAC) (Haarnoja et al., 2018). We dive into the motivations behind each of these algorithms, along with their strengths and weaknesses, to gain a deeper understanding of the potential pitfalls in applying them to the portfolio allocation task. For each algorithm, we use the implementations of stable baselines 3 (Raffin et al., 2021).

We adopt continuous actions represented as direct portfolio weights for our portfolio allocation task, despite the numerous alternatives. We deem discrete actions impractical due to the combinatorial complexity arising from multiple stocks. Multidiscrete actions, where each stock allocation can be chosen independently, are unsupported by most popular algorithms [1], including DDPG and SAC. Although defining actions as weight changes could simplify minimizing asset turnover, which is an important consideration in high-transaction-cost environments, representing actions as direct weights affords the agent greater control and precision over the portfolio allocation, despite the additional complexity. Drawing on DRL's proven success in continuous-control domains like robotics and logistics (Arulkumaran et al., 2017), where agents need to precisely control voltage on robot joints or the allocation of inventory across warehouses, we opt for this approach to enable precise adjustments. From a Markov Decision Process (MDP) perspective, where state and action sequences must be non-random, and to equip the agent with a mechanism to manage turnover, incorporating the previous portfolio weights into the observation is crucial when actions are defined as direct portfolio weights.

[1] At the time of writing, DDPG and SAC only support Box (continuous) action spaces in Stable Baselines 3 (Raffin et al., 2021).

Now, we provide a discussion of DDPG, PPO, and SAC, explaining their architectures and key mechanisms. We also compare them, hypothesizing what features might be beneficial in the portfolio allocation task.

3.1.1 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) extends Deep Q-Learning (Mnih et al., 2015) to continuous action spaces by using an actor-critic framework (Lillicrap et al., 2015). The actor outputs continuous actions and the critic evaluates them. DDPG is model-free and off-policy, as it draws transitions from a replay buffer. DDPG performs well on many continuous-control physics tasks, such as the cartpole swing-up (Lillicrap et al., 2015), dexterous manipulation (Gu, Lillicrap, and Levine, 2017), and car driving (Jin et al., 2021).

The motivation behind DDPG lies in its ability to output continuous actions, as many real-world tasks, such as robotic control or financial portfolio optimization, involve continuous action spaces. Traditional value-based methods struggle in these domains due to the need to discretize actions. Policy-based methods can output continuous actions, but often suffer from high variance in gradient estimates (Williams, 1992).

DDPG solves these issues by using a deterministic policy to reduce variance in the gradient estimates and maintaining two neural networks, namely an actor network $\mu(s|\theta^\mu)$ and a critic network $Q(s, a|\theta^Q)$. Additionally, DDPG uses two target networks, μ' and Q', inspired by DQNs, to stabilize learning. These target networks are updated to reflect the parameters of the online actor and critic networks with a soft-update rule.

In Algorithm 1, we show the DDPG algorithm in pseudocode, as it appears in Lillicrap et al., 2015. Noteworthy is the use of $\mathcal{N}$ for exploration, which essentially means that we add noise to the actions to induce exploration. This noise process can take any form, but most commonly it is either random normally distributed or Ornstein-Uhlenbeck noise, which is temporally correlated (Uhlenbeck and Ornstein, 1930). The addition of stochasticity on the level of the actions may be suboptimal in highly dynamic environments.

Algorithm 1 DDPG Algorithm (Lillicrap et al., 2015)

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$.
Initialize replay buffer R.
for episode = 1 to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration.
  Receive initial state s_1.
  for $t = 1$ to T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise.
    Execute action a_t and observe reward r_t and new state s_{t+1}.
    Store transition (s_t, a_t, r_t, s_{t+1}) in R.
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R.
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$.
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$.
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}.$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'},$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}.$$

  end for
end for

To conclude, DDPG handles high-dimensional, continuous action spaces effectively without discretization. The replay buffer facilitates off-policy learning, making efficient use of past experiences. This can be particularly useful when sampling random portfolios, where feature distributions may change at the start of every episode. Using previous experience, the agent can improve stability and improve generalization across stocks. However, DDPG does not account for the overestimation bias, as later improved upon in the TD3 algorithm (Fujimoto, Hoof, and Meger, 2018). Also, in practice, DDPG's performance can be highly sensitive to hyperparameter tuning, including learning rates, noise scaling, and the size of the replay buffer. Most importantly, DDPG relies on action noise for exploration, which may be suboptimal in highly dynamic environments.

DDPG is used frequently in the literature on portfolio optimization and will serve as a useful baseline in this thesis. However, the Proximal Policy Optimization and Soft Actor-Critic algorithms, discussed in the next chapters, use stochastic policies, which make them likely more effective in financial tasks where the signal-to-noise ratio in the data is low.

3.1.2 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a widely used reinforcement learning algorithm that simplifies the complex trust region optimization in Trust Region Policy

Optimization (TRPO) (Schulman et al., 2015) while retaining its key benefits. Introduced by Schulman et al., 2017, PPO is an on-policy algorithm with a novel objective function that enables multiple epochs of minibatch updates. Due to these improvements, PPO is simple to implement, sample efficient, and shows great performance on simulated robotic tasks and Atari games.

Schulman et al., 2017 recognize that vanilla policy gradient methods are not very sample-efficient and not robust enough, and that TRPO is a relatively complicated algorithm. Traditional policy gradient methods are unstable when the new policy deviates too much from the old one. TRPO addresses this issue by constraining the policy update with a trust region, but requires solving a complex optimization problem. PPO simplifies this by replacing the trust region constraint with a clipped objective function. Using this surrogate objective, PPO allows for larger updates while ensuring that the new policy does not diverge excessively from the old policy.

In Algorithm 2, we present the basic pseudocode for PPO, as described in Schulman et al., 2017[2]. Unlike off-policy algorithms such as DDPG, PPO is an on-policy algorithm and uses fresh trajectories for each update, gathered by the current policy. Additionally, PPO uses minibatches of sampled trajectories for multiple epochs of optimization, further improving sample efficiency.

Algorithm 2 PPO Algorithm (Schulman et al., 2017)

Initialize policy network $\pi_\theta(a|s)$ and value function $V_\phi(s)$ with random weights θ and ϕ.
for iteration = 1 to M **do**
  Collect T timesteps of data by running the policy π_θ in the environment.
  Compute advantage estimates $\hat{A}_t$ using the collected trajectories.
  for epoch = 1 to K **do**
    Sample a minibatch of trajectories.
    Compute the PPO objective:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right].$$

    Update the policy network by maximizing $L^{\text{CLIP}}(\theta)$.
    Compute the value loss $L^{\text{VF}}(\phi)$.
    Update the value function by minimizing $L^{\text{VF}}(\phi)$.
  end for
end for

To provide some intuition on the PPO objective, it essentially prevents excessive policy updates. It does this by combining the unclipped surrogate loss (first term in the `min` operator) with a clipped version that limits the probability ratio $r_t(\theta)$ to the interval $[1-\epsilon, 1+\epsilon]$. By taking the minimum of these two terms, the algorithm ensures a pessimistic lower bound on the original objective. PPO selectively uses the change in probability ratio, disregarding it when it improves the objective and including it only when it worsens the objective (Schulman et al., 2017).

Unfortunately, PPO is quite sensitive to hyperparameter settings, requiring careful implementation (Huang et al., 2022). Key parameters such as learning rate, clipping range, batch size, and number of epochs significantly influence its performance.

[2]For deeper explanations of the terms such as $L^{\text{CLIP}}(\theta)$ and $L^{\text{VF}}(\phi)$ we refer to the original PPO paper.

To summarize, PPO aims to balance stability and simplicity, as it enables policy updates without requiring the complex trust region constraints used in TRPO. Although theoretically promising, it has also been shown to work practically in high-variance environments, such as financial markets (Wen, Yuan, and Yang, 2021; Sood et al., 2023). The use of advantage estimates $\hat{A}_t$ helps reduce variance in gradient updates, while the clipped surrogate objective ensures stable learning, which is important in the inherently uncertain and non-stationary financial markets. Because of this, its ability to handle stochastic policies makes it a strong candidate for financial portfolio optimization.

However, its on-policy reliance on fresh trajectories can limit sample efficiency, potentially amplifying noise if sampled data is unrepresentative or volatile. To mitigate this, PPO may benefit from noise-reduction strategies, such as training on longer intervals (Wen, Yuan, and Yang, 2021). In addition, PPO is quite sensitive to the choice of hyperparameters (Huang et al., 2022).

3.1.3 Soft Actor-Critic (SAC)

The last algorithm we discuss is the Soft Actor-Critic (SAC) algorithm. SAC is an off-policy algorithm that introduces entropy maximization into the policy objective to encourage more exploratory behavior (Haarnoja et al., 2018). Just like PPO, it uses a stochastic policy. It also incorporates a temperature parameter into the policy update to balance the exploration-exploitation trade-off. SAC is particularly effective in environments with high-dimensional, continuous action spaces and is therefore widely applied in robotics and other continuous control tasks.

The key motivation behind SAC lies in improving sample efficiency and robustness. Unlike DDPG, SAC uses a stochastic policy, which tries to avoid overfitting to specific regions of the state-action space. SAC uses an actor-critic architecture with two critics to mitigate the overestimation bias often observed in Q-learning methods, which is inspired by TD3. Additionally, SAC adds an entropy term to the objective, such that we now maximize a trade-off between the expected reward and the entropy of the policy. This is formulated as maximizing $J(\pi) = \mathbb{E}\left[\sum_t \gamma^t (r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t)))\right]$, where α is the temperature parameter, or entropy coefficient, and $\mathcal{H}$ is the entropy. A higher entropy coefficient places more weight on the entropy term in the objective, leading to more randomness in action selection. This coefficient is learned by the algorithm and adapted to the specific dynamics of the environment, as the algorithm tries to match a certain target entropy. The target entropy is normally set at the negative value of the size of the action space, to introduce more exploratory behavior if the action space increases. Furthermore, the algorithm uses a replay buffer for experience sampling, improving sample efficiency.

In Algorithm 3, we provide pseudocode for SAC, as outlined in Haarnoja et al., 2018. Important to note is that SAC uses many networks, namely a value function ψ, a target value function ψ^{target}, two Q-functions θ_1, θ_2, and a policy network ϕ. In more recent implementations of SAC, however, the value function and the target value function are inferred from the Q functions and the policy network. This stabilizes learning and reduces computational complexity.

Algorithm 3 SAC Algorithm (Haarnoja et al., 2018)

Initialize parameter vectors ψ (value function), ψ^{target} (target value function), θ_1, θ_2 (Q-functions), and ϕ (policy).
Initialize replay buffer $\mathcal{D}$.
for each iteration **do**
  for each environment step **do**
    Sample action $a_t \sim \pi_\phi(a_t|s_t)$.
    Observe next state $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$ and reward $r_t = r(s_t, a_t)$.
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$.
  end for
  for each gradient step **do**
    Sample a minibatch of transitions (s_t, a_t, r_t, s_{t+1}) from $\mathcal{D}$.
    Update value function: $\psi \leftarrow \psi - \lambda_V \nabla_\psi J_V(\psi)$
    Update Q-functions: $\theta_i \leftarrow \theta_i - \lambda_Q \nabla_{\theta_i} J_Q(\theta_i), \quad \forall i \in \{1, 2\}$
    Update policy: $\phi \leftarrow \phi - \lambda_\pi \nabla_\phi J_\pi(\phi)$
    Update target value function: $\psi^{\text{target}} \leftarrow \tau\psi + (1-\tau)\psi^{\text{target}}$
  end for
end for

SAC appears to be well suited for financial portfolio optimization due to its sample efficiency, which is critical when working with limited data in financial markets. Its stochastic policy is particularly advantageous in handling the inherent uncertainty and non-stationarity of market environments, allowing the agent to explore a more targeted range of actions compared to deterministic approaches like DDPG. Furthermore, we expect that the entropy-augmented objective helps SAC outperform on tasks where balancing exploration and exploitation is crucial. As we are working with a deterministic financial dataset, we do not necessarily expect exploration to have a huge impact, although it is important for an algorithm to be exposed to a wide variety of allocations to distinguish profitable assets.

The off-policy nature of SAC allows for more efficient data utilization compared to PPO. We hypothesize that this is primarily important when sampling random portfolios. Additionally, its stochastic policy and entropy-based exploration offer a more sophisticated approach to handling uncertainty. However, this increased sophistication comes at the cost of higher complexity, which can make implementation and debugging more challenging.

3.2 Data Acquisition and Preprocessing

In this section, we discuss the acquisition and pre-processing of data that is used to train our DRL algorithms and compare them with traditional portfolio optimization methods. We use a substantial data set of companies from the S&P1500 index, which contains the large-cap stocks in the S&P500, mid-cap stocks from the S&P400 and the small-cap stocks in the S&P600 of the United States of America. As the S&P1500 contains about 90% of the market capitalization of the US, this index is often indicative of the wider US equity market (S&P Dow Jones Indices, 1995).

We have data available for a wide variety of company fundamentals, analyst predictions and sentiment, pricing series, macroeconomic indicators, and data on which companies were in the S&P1500 on a monthly basis, which we call the universe from now on. Our data spans from 1995 until the beginning of 2024, but some features are only available from 2010. Therefore, we train, validate, and test our algorithms

FIGURE 3.1: Adjusted close price series of the S&P 1500, highlighting the training, validation, and test periods, with annotations of significant geopolitical events potentially impacting the U.S. market.

using data from 2010 to the middle of 2024. Specifically, our train set spans from 2010 until the end of 2019, our validation set is from 2020 until the end of 2021 and our test set encompasses the years 2022, 2023 and 2024 until the end of June.

In Figure 3.1, we present the adjusted close price series of the S&P 1500 as a reference, along with the time spans of the training, validation, and test sets. We have also annotated key geopolitical events that may have influenced the U.S. market. The validation set may be affected by these geopolitical events in particular. This is because the COVID-19 crisis happens at the beginning of the validation period (early 2020), which triggered extreme market volatility followed by substantial market growth. The test set captures both periods of decline and upward trends, influenced by uncertainties surrounding geopolitical developments, including the U.S. presidential election, the Russia-Ukraine conflict, and other global tensions.

In Section 3.2.1, we start with a brief description of the most common biases that can arise in financial data and methods we employ to mitigate them. Then, in Section 3.2.2, we discuss the calculation of the return series, which is used to benchmark our algorithms and traditional methods, and which serves as a basis for the reward for the DRL agents. In Section 3.2.3, we briefly discuss the features that give the algorithms a sense of the potential of an asset. In Section 3.2.4, we investigate what constitutes the universe, and in Section 3.2.5, we detail the macroeconomic indicators that will form the contextual state of the financial market.

3.2.1 Mitigating Biases

As handling big data in finance is so complex, it requires a careful approach to mitigate biases and other potential pitfalls that can arise within the data (Kousen et al., 2023). Some of the most prominent biases are survivorship bias, look-ahead bias or forward-looking bias, and data mining.

Survivorship bias arises when an investor backtests or trains their algorithms using data exclusively from companies that remain listed in an index, neglecting to incorporate data from companies that have been delisted. For example, consider

training an algorithm on 10 years of data from the current S&P500 constituents, excluding companies that were delisted during that period despite being present at the start. In that case, we train the algorithms on survivor data, which skews investment strategies and degrades performance in live situations where future survivors remain unknown. To account for this bias, we always train, validate and test our methods using the index universe at the start of each of these datasets. This means that some companies in the resulting portfolios will be delisted. If this occurs, we stop trading in that company and treat positive weights in that asset as if it were invested in the cash position. We discuss this further in Section 3.2.4.

The second eminent bias is look-ahead bias, also referred to as forward-looking bias, and it occurs when we use data for training that wasn't available at that point in time. For instance, this can happen if we use a company's earnings data before it was made publicly available. Using unavailable data leads to algorithms that learn incorrect patterns, resulting in backtests that often perform deceptively better and live trading that fails to meet expectations. In our feature calculations and preprocessing, we take the utmost care in preventing data from spilling over to the future.

The last important bias is the data mining bias, which arises when investors go through large amounts of data in search for profitable patterns. This can lead to overfitting, where models identify spurious relationships that do not generalize to new data. For example, an investor might employ an excessively complex deep neural network, resulting in a model that captures noise rather than genuine market signals. This overfitting produces an overly optimistic view of the model's performance, which often fails to translate to real-world trading scenarios. In our framework, we employ early stopping, halting training our models once performance on a validation set degrades. Furthermore, we use relatively small models with minimal nodes and layers. These solutions are both more computationally efficient and less prone to overfit. We then test the algorithms on a separate test set that has not been used in either training or validation, which provides a reliable measure of the model's performance on unseen data. On top, as we use diverse portfolios in our testing framework, we minimize the possibility of cherry-picking a single, well-performing portfolio.

3.2.2 Return Series

To reward agents on the portfolio allocation task, we use the returns of the individual assets. To ensure realistic and unbiased rewards, we have to carefully curate them. The price of a financial asset can be measured at multiple times of a trading day; often at the open, high, low, and close.

Denoting P_t as the close price on day t, the close-to-close return on day t can be formulated as

$$r_t = \frac{P_t}{P_{t-1}} - 1. \tag{3.1}$$

We have to address the timing of rebalancing decisions before calculating the agents' reward. Specifically, allocations made at the end of day t can only rely on feature data available up to that point. To avoid look-ahead bias, where some features may incorporate information unavailable before market close, we have to consider only feature data up to day $t-1$. Following Benhamou, 2023, this issue is resolved by shifting the close-to-close return series by one day. Consequently, allocations for day t are based on features from day $t-1$, and the corresponding reward is calculated using return r_{t+1}, taking day t as the rebalancing day.

However, we can also eliminate this bias if we use the open-to-open return series, which assumes trades occur at the open price of the following day. As the market prices in new information very quickly, taking one full rebalancing day might reduce the informative value of the features. Although in practice it is not feasible to trade directly at the open, this approach remains very realistic and is more powerful than taking one day to rebalance.

In our final implementation, we also incorporate unadjusted dividend, payed out at the ex-dividend day. We utilize unadjusted open prices, meaning these prices do not account for stock splits, mergers, stock buybacks, or other corporate actions. Then, the reconstructed return for day t is calculated as follows:

$$r_t = \frac{\mathrm{CAI}_{t+1} \cdot \mathrm{OP}_{t+1}}{\mathrm{CAI}_t \cdot \mathrm{OP}_t} + \frac{\mathrm{D}_{t+1}}{\mathrm{OP}_t}, \tag{3.2}$$

where CAI_t is the capital adjustment index, accounting for stock splits and other adjustments, OP_t is the unadjusted open price, and D_t is the unadjusted dividend payout, all for day t.

Data availability for the open price series, dividends, and the capital adjustment index is nearly complete, minimizing the need to impute missing values. However, if they are encountered, we impute them with zero, effectively treating those days as having no returns.

3.2.3 Features

Within our framework, the DRL agents can access 116 features per company, encompassing a broad range of information, including company fundamentals, technical indicators, analyst predictions, and sentiment data. This section provides a concise overview of each feature group, outlining the rationale for their inclusion in the dataset.

Company fundamentals are derived from the financial statements of a company, indicating its financial health. Examples include quarterly operating cashflow, dividend yields, book-to-price ratio, debt-to-equity ratio, earnings yields, asset turnover, and return on investment.

Technical indicators give insights into the movement of the stock price and volume, such as momentum indicators. In our dataset, we include the moving average convergence/divergence (MACD) (Appel, 2005), the relative strength index (RSI) (Wilder, 1978), the average daily volume (ADV), asset volatility, and multi-period return series.

We also incorporate analyst predictions on a wide variety of stock attributes, as advice from financial experts can have a substantial impact on price movements. Predictions range from earnings (growth), sales (growth), dividends (growth), and general recommendations. These features can also carry information about the general sentiment towards these companies.

Finally, we add one more feature that indicates if that specific company is in the universe, which is a simple dummy variable. This feature takes the value 1 if the company is in the universe, and 0 otherwise. It should help the algorithms to divest from this company, as no returns can be made after it has been delisted.

These $K = 117$ features exist for companies N on dates T, such that they form a $(T \times K \times N)$ matrix. To make this more intuitive, this structure is visualized in Figure 3.2. Note that not all x_{ij} might be available, as some feature values are missing.

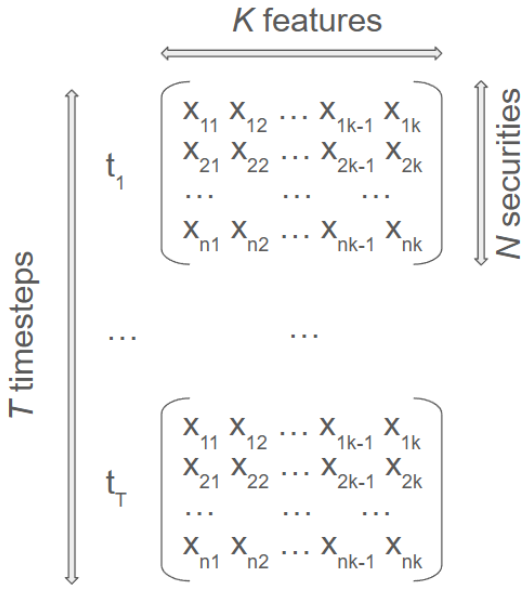


FIGURE 3.2: Schematic overview of the features data, where $K = 117$ is the total number of features, N is the number of companies, and T are the total number of dates in the feature data.

Feature Normalization

The deep neural networks in DRL algorithms benefit a lot from normalized data (Patro, 2015). Asset prices, earnings, ratios, and analyst predictions often fall outside the zero-to-one range. Therefore, we have adopted a unique approach towards normalization, where we rank feature data within predefined groups to ensure comparability across different scales.

First, we create group-specific subsets using the raw data and some normalization attributes. Specifically, we group on the beta of an asset and the sector in which the company operates, where the beta indicates how much a specific asset correlates with the wider equity market. We use this grouping as we expect that the features dynamics of an asset in different sectors and beta values can differ significantly. Then, within these groups, features are ranked proportionally to their values, transforming them into percentile scores. Finally, we ensure the normalized values lie within the interval $(0, 1)$.

Specifically, let the raw data be X and the neutralization attributes (sector and beta) define the grouping variables G. The normalized feature X_{norm} for each group $g \in G$ is computed in two steps.

First, each feature x_i in group g is transformed into its percentile rank, as

$$R(x_i) = \frac{\text{rank}(x_i)}{n_g}, \tag{3.3}$$

where $\text{rank}(x_i)$ is the rank of x_i in group g, and n_g is the total number of elements in g. Second, the rank is adjusted to center the data on $(0, 1)$, using

$$X_{\text{norm}} = R(x_i) - \frac{1}{2n_g}. \tag{3.4}$$

Now, all features are normalized within the same scale, reducing potential biases introduced by heterogeneous feature distributions.

Missing Values and Data Cleaning

Cleaning, preprocessing and handling missing values in the feature data requires a methodical approach to avoid biases. In most neural networks, in contrast to bagging and boosting algorithms, for example, missing values either have to be removed or imputed. In our analysis, missing values for each feature belong to one of the following four cases:

1. The feature has no data for all companies on all dates (no data for the feature).
2. The feature has no data for some companies on some dates.
3. The feature has no data for all companies on some dates (the feature does not exist on some dates).
4. The feature has no data for some companies on all dates (the feature does not exist for some companies).

We take the following steps to remove the missing values, which belong to each of the four classes described above.

1. Features with virtually no data are removed. This solves class 1.
2. We forward fill the features with a limit of 5 days (1 trading week) to fill any gaps in feature availability. This will partially tackle class 2.
3. We fill dividend features with 0 (unless they are revision or growth), as missing values for dividend features are likely companies that give no dividends, or not yet on those dates. Missing dividends mostly belong to class 4, as some companies don't have dividends.
4. For each feature, we calculate the mean value grouped by date and fill the remaining missing values with these means. This is intuitive, as we consider these companies to be the market average. This procedure will mainly tackle class 4 and long gaps in class 2, as some companies have to have data on a specific date in order for a mean to be calculated.
5. We are now left with class 3. We fill these values with 0.5, as we expect all companies to be the average of the market in that case.

After these steps, all missing values of the features of those companies that are in the index are either dropped or filled with good proxies. There should be no remaining missing values.

3.2.4 Universe

The universe is a mapping from each date in the history to the companies that are in the S&P1500. This mapping happens on a monthly basis, but we forward filled the entries to give us a daily mapping. To ensure the stability of our DLR algorithms and maintain the integrity of our analysis, we excluded companies once their stock prices fell below one dollar. Such low stock prices often indicate financial distress and frequently lead to unusually high or low returns, potentially destabilizing the learning process of our algorithms. After removing these companies, we are left with 1469 securities at the start of the training set, 1464 securities at the start of the validation set, and 1465 securities at the start of the test set.

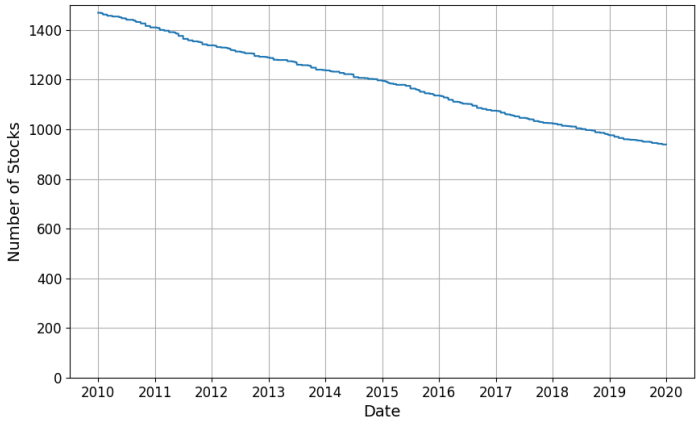


FIGURE 3.3: The size of the universe over the training horizon, i.e. the number of stocks that are listed in the S&P1500, after removing securities without feature data and with prices below one dollar.

Some of the initial companies in the universe are delisted during the training period T_{train}, validation period T_{val}, and testing period T_{test}. Even though new companies take their place, we couldn't have known that at the start of the optimization horizon. Therefore, as introduced in Section 3.2.1, we treat positive weights in these companies as an investment in the cash position, which returns 0 at each step t [3]. Furthermore, features are forward filled over the entire horizon T as soon as a company has been delisted. To provide an intuition on how many of the initial companies remain in the universe over the training period, we include the universe size in Figure 3.3.

3.2.5 Macroeconomic Indicators

In our study, we use macroeconomic indicators to gauge the state of the US equity market. Specifically, we use 7 variables to help the algorithms in capturing market dynamics.

- We include 4 treasury bills, represented by the 3-month, 2-year, 10-year, and 30-year benchmarks, which reflect the term structure of interest rates and provide insights into expectations around economic growth and inflation.
- Additionally, we include the U.S. Dollar Index, often denoted as the DXY, which highlights the relevance of currency strength in global trade and its impact on multinational corporations and commodities.
- Another important indicator is the West Texas Intermediate (WTI), a benchmark for crude oil prices. This serves as a barometer for energy market conditions and broader economic activity.
- Finally, the high-yield corporate bond spreads encapsulate credit market stress and investor sentiment towards riskier asset classes.

[3] An alternative approach could involve replacing delisted companies with their new counterparts. However, this introduces a potential risk of destabilizing the learning process, as the distributional change in returns and features data might disrupt the recursive relationships between states, rewards, and actions defined by the Bellman equations. Therefore, we have chosen for this simpler approach.

In the final states, we use the percentage change of these macroeconomic indicators rather than their raw values, acting as a normalization step. Besides, the changes in macroeconomic activity are assumed to carry more information than their absolute values.

3.3 Environment Design

For the portfolio allocation task, we have designed an environment following the Gymnasium API (Towers et al., 2024), which significantly facilitates the engineering process. This environment provides the agents with states and rewards, following an action.

A crucial aspect of our environment lies in the dynamic selection of portfolio assets. Rather than optimizing for a fixed set of assets, we sample n stocks from a pool of N available stocks in the data. This helps us address the problem of limited data availability and allows us to build a framework that is not dependent on the specific companies in the portfolio. Also, this approach allows us to build a statistically valid framework, as we can compare algorithms and baselines across multiple portfolios.

In this section, we detail the specifics of the states, actions, and rewards. Specifically, in Section 3.3.1, we introduce different types of observation (state) spaces, in Section 3.3.2, we discuss the action space, and in Section 3.3.3, we list the reward function used in the experiments of this thesis.

3.3.1 Observation Space

The observations $o_t \in \mathcal{O}$ in our environment can take three forms. Each type builds on the next, adding more information.

1. The first type of observation only uses company-specific data. More specifically, the state encompasses feature data X for each asset in the n companies in the portfolio. We can optionally incorporate a lookback window of size l_X, which takes feature values from steps $t - l_X, ..., t$ into account. Taking this all together, this forms a $(l_X \times n \times k_X)$ matrix. To properly form this matrix according to RL standards, we need a square matrix without missing values. This means all companies will have data on all dates on the horizon T, also those that are delisted. As noted previously, this is done by forward-filling the last known feature values of those companies. Figure 3.4 illustrates the transformation process from raw feature data to a structured observation within our environment.

2. The second observation type adds the previous weights $\mathbf{w_{t-1}}$ to the state space. This helps the agent to identify the impact of his actions, especially when dealing with transaction costs. As noted previously, this element is vital when using actions as direct portfolio weights. It also serves a theoretical purpose, as agents can now directly influence the observation with their action, an important attribute of a Markov Decision Process.

3. The third and last type of observation incorporates contextual market data C. This is a $(l_C \times k_C)$ matrix comprising k_C macroeconomic indicators, with an optional lookback window of size l_C.

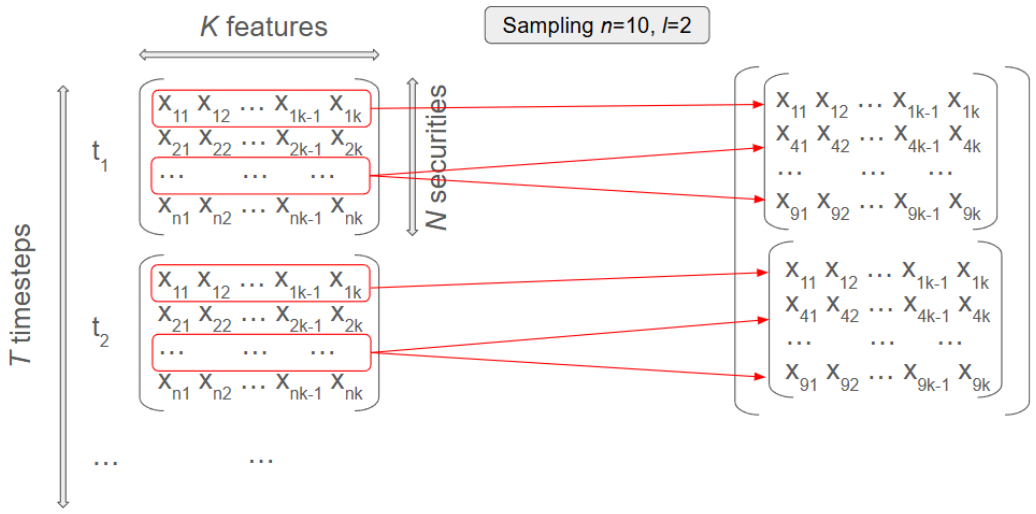


FIGURE 3.4: A schematic overview of the sampling process from the pool of feature data to an actual observation at each step t in the environment, for an observation with $l = 2$ and $n = 10$.

3.3.2 Action Space

The action vector $\mathbf{a_t}$ determines the weights for each of the n assets in the portfolio and a cash position. The cash position has 0 returns and offers the agent a way to divest in times of market turbulence. The actions fall in the range $[-1, 1]$, following the advice from Stable Baselines 3 (Raffin et al., 2021) to normalize the action space.

To construct these actions into a weight vector $\mathbf{w_t}$, we shift the actions to [0, 1] and normalize them as follows:

$$\mathbf{w_t} = \frac{\frac{\mathbf{a_t}+1}{2}}{\sum \frac{\mathbf{a_t}+1}{2}}. \tag{3.5}$$

3.3.3 Reward Function

Our environment accommodates one primary reward function, namely the logarithmic return. In our setup, the reward at time t, denoted r_t, is defined based on the change in portfolio value resulting from action a_t taken in state s_t, leading to next state s_{t+1}. Concretely, the reward is computed as:

$$r(s_t, a_t, s_{t+1}) = \log\left(\frac{v_{t+1}}{v_t}\right), \tag{3.6}$$

where v_t and v_{t+1} denote the portfolio values at time t and $t+1$, respectively. This definition ensures compatibility with the Bellman recursion, which requires the rewards to be additive over time. In financial applications, using arithmetic returns directly violates this constraint due to their multiplicative nature over time. Logarithmic returns, however, are additive and therefore align naturally with RL's return formulation $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$, as introduced in Chapter 2. Furthermore, using the logarithmic return ensures proportional scaling and mitigates the influence of extreme fluctuations. It is widely used in DRL applications in finance.

To promote stable learning, we apply reward scaling to bring the reward distribution closer to a standard normal distribution, which helps avoid vanishing or

exploding gradients. Empirically, we scale the reward by a factor of 70 for a portfolio of $n = 10$ stocks, yielding

$$\tilde{r}_t = 70 \cdot r_t. \tag{3.7}$$

This choice is based on empirical observations of the distribution of rewards between training episodes.

Transaction Costs

To make our portfolio allocation environment more realistic, we use transaction costs. In essence, we subtract a small fee from the total portfolio return, proportional to the reallocated capital. Transaction costs can be calculated in a multitude of ways, but we opt for a simple approach. Specifically, we formulate the transaction cost on day t as

$$TC_t = c \cdot |\mathbf{w_t} - \mathbf{w_{t-1}}| \cdot v_{t-1}, \tag{3.8}$$

where c is the level of the fee, $\mathbf{w_t}$ are the portfolio weights on day t, and v_{t-1} is the value of the portfolio on the previous day. The level of c in a liquid market, such as the US stock market, is often around 5 basis points, or 0.05 %. In chapter 4, we experiment with this value to see how it impacts our DRL agents. The transaction cost is subtracted from the reward, incentivizing agents to minimize asset turnover.

3.3.4 Transition Dynamics

At each time step t, the environment transitions according to the following procedure. First, the agent selects an action $\mathbf{a}_t \in [-1, 1]^{n+1}$, which is mapped to a normalized portfolio weight vector $\mathbf{w}_t \in \Delta^{n+1}$. The portfolio value v_t is updated using asset returns observed between $t-1$ and t, and transaction costs are subtracted based on the change in allocation $|\mathbf{w}_t - \mathbf{w}_{t-1}|$. The immediate reward is computed as the logarithmic return. The next state s_{t+1} is constructed from updated feature data, optionally including a lookback window, contextual variables, and the portfolio weights $\mathbf{w}_t$. In this design, r_t and s_{t+1} depend only on s_t and a_t, satisfying the Markov property. In our setting we do not model market impact, such that only the portfolio weights are directly influenced by a_t and the rest is determined by the market.

We conclude these dynamics in Algorithm 4, where we show the general workflow of our portfolio environment.

Algorithm 4 Portfolio Environment Workflow

Define hyperparameters: number of securities in the portfolio n, stock pool size N, initial portfolio value v_0, transaction cost percentage c, the reward function $\mathcal{R}$, reward scaling coefficient α, and the observation space $\mathcal{O}$.
for episode = 1 to M **do**
  Set portfolio value v_0
  Randomly select n from N
  for environment step = 1 to T **do**
    Select action $\mathbf{a_t}$
    Normalize actions to form portfolio weights $\mathbf{w_t}$:

$$\mathbf{w_t} \leftarrow \frac{\mathbf{a_t} + 1}{2} \quad \text{(Normalize actions to [0,1] range)}$$

$$\mathbf{w_t} \leftarrow \frac{\mathbf{w_t}}{\sum \mathbf{w_t}} \quad \text{(Normalize weights to sum to 1)}$$

    Compute delta weights, exluding the cash position: $\Delta \mathbf{w_t} \leftarrow \mathbf{w_t} - \mathbf{w_{t-1}}$
    Calculate transaction cost: $TC_t \leftarrow c \cdot |\Delta \mathbf{w_t}| \cdot v_{t-1}$
    Update portfolio value: $v_t \leftarrow v_{t-1} - TC_t$
    Update portfolio allocation using the daily returns:

$$\mathbf{x_t} \leftarrow \mathbf{w_t} \odot \mathbf{r_t} \cdot v_t \quad \text{(Hadamard product of portfolio weights and returns)}$$

    Update portfolio value: $v_t \leftarrow \sum \mathbf{x_t}$
    Update weights: $\mathbf{w_t} \leftarrow \frac{\mathbf{x_t}}{v_t}$
    Compute reward: $R_t \leftarrow \log(\frac{v_t}{v_{t-1}})$
    Scale reward: $R_t \leftarrow \alpha R_t$
    Get next observation: $o_{t+1} \in \mathcal{O}$
    if $v_t < 0$ **then**
      Break (terminate episode if portfolio value is negative)
    end if
  end for
end for

3.4 Neural Network Architecture

In this section, we detail the neural network architecture which is used by the algorithms. We employ one shared features extractor that handles the features of each of the assets in the portfolio, similar to the approach of Jiang, Xu, and Liang, 2017. This extractor is trained to identify valuable signals that can be applied throughout the portfolio by uniformly analyzing the characteristics of each company. In Figure 3.5, we show the general model architecture. The architecture changes depending on the inclusion of a lookback window in the feature and contextual data, but we discuss that later in this chapter.

As mentioned before, at each step t, we can use contextual data C_t, the previous weights $\mathbf{w_{t-1}}$, and feature data X_{ti} for each company i in the portfolio. The feature data is processed by a shared features extractor of multiple Linear, LayerNorm, and ReLU layers. Layer normalization is used to tackle the vanishing or exploding gradient problem, as we input many different feature matrices which might have different data distributions. ReLU is used to introduce non-linearity in the modeling process. Each of the feature matrices is processed into an embedding vector f_i, which is a

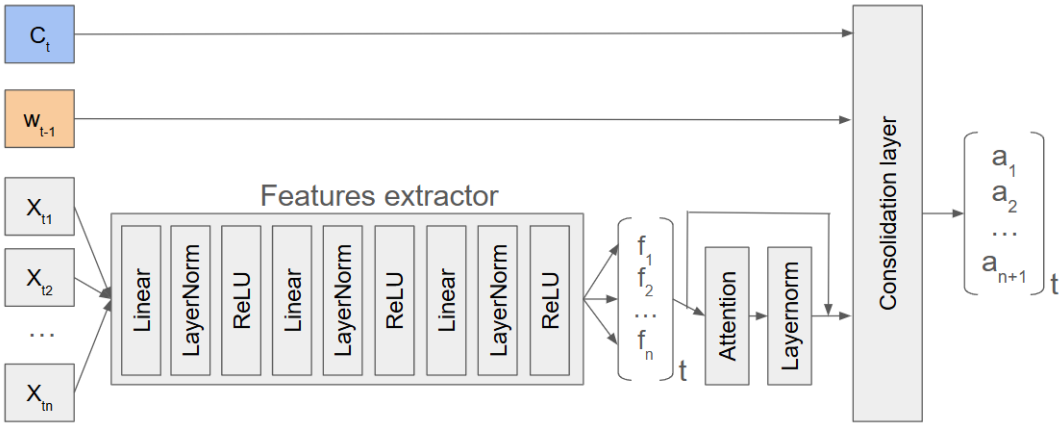


FIGURE 3.5: The model architecture when only using data of the current time step t. At each step t, the model processes contextual data C_t, the previous weights $\mathbf{w_{t-1}}$, and feature data X_{ti} for each company i in the portfolio. The feature data is processed by a shared features extractor of multiple Linear, LayerNorm, and ReLU layers, and optionally processed by an Attention layer for permutation equivariance, producing embedding vectors f for each asset in the portfolio. The consolidation layer consolidates all the information into an action vector, which is used by the environment to produce portfolio weights.

small vector that contains compressed information on company i. After the features extractor, the embedding vectors are jointly passed through an optional one-head attention layer, which should aid in recognizing different permutations of assets. We also use a residual layer that circumvents the attention layer, an approach used to stabilize training in relatively deep networks, such as the Transformer. The residual layer ensures that the model can continue learning even if the attention mechanism struggles to capture relevant patterns. Finally, all information is processed by the consolidation layer, which is a small fully connected (Linear) layer that outputs the actions, which become the weights of the assets in the portfolio.

In case a lookback window is employed, the extra time dimension needs to be handled. This is done by incorporating a 1-dimensional convolutional layer, as shown in Figure 3.6. [4] The convolutions are applied along the time dimension t of the features (of size l_X) or contextual variables (of size l_C), seeking to extract temporal patterns. Convolutions are not applied along the feature dimension $k \in K$ and the security dimension $n \in N$, as there is no underlying spatial structure in these dimensions. The securities are shuffled, rendering spatial pattern extraction irrelevant, and the features are inputted without a predefined order. The feature matrices of the different companies are treated as independent inputs to the feature extractor, with the individual features serving as channels in the convolutional process. This approach avoids imposing any structural assumptions on the feature dimension, as the output of the different channels are ultimately summed, and summation

[4] In our experiments, we opt for short lookback windows to manage memory constraints, given that our algorithms employ replay buffers holding 1 million observations, each encompassing 117 features across 10 companies. For instance, storing a replay buffer of this size with a lookback window of 10 and 32-bit floating-point precision requires approximately 44 GB of RAM for the observations alone. For algorithms that also store next states, this demand doubles to 88 GB, escalating rapidly beyond practical limits. To address this, we restrict our experiments to a four-step look-back window, such that a single convolutional layer with a kernel size of 3 suffices.

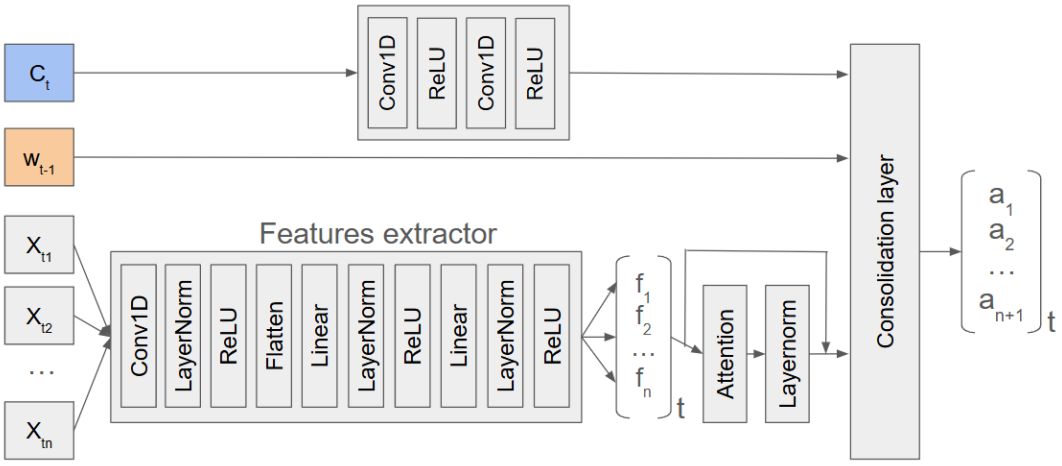


FIGURE 3.6: The model architecture when using a lookback window in the feature data and the contextual data. The feature data is again processed by a shared features extractor. The Conv1D layer processes the time dimension by extracting temporal patterns. The consolidation layer consolidates all the information into an action vector, which is used by the environment to produce portfolio weights.

is inherently permutation-invariant. The rest of the architecture remains the same.

Our implementation uses the DRL agents provided by Stable Baselines 3, adhering to their default hyperparameters whenever possible (Raffin et al., 2021). We have customized the feature extractors to align with the specific requirements of our portfolio optimization task. Other hyperparameters are listed in Table 3.1

3.4.1 Permutation Equivariance

The random sampling of assets can be hard for the consolidation layer to process, as it must handle diverse asset orderings and maintain representational consistency. To handle this, the literature introduces several solutions.

One option is to expose the model to a diverse set of permutations and assume that it will learn the inherent complexity itself. This approach can be useful, but as the size of the portfolio increases, the number of different permutations grows rapidly.

An alternative is to use Graph Neural Networks (GNNs). Here, nodes are often treated as unordered sets, which do not take into account the random ordering of the assets. However, due to their increased complexity and limited usage in DRL tasks, we have opted against this approach.

A widely used architecture is the attention mechanism, which is inherently permutation equivariant (Vaswani, 2017). This means that permuting the input sequence does not alter the overall output, only its ordering. Using the mathematical notation for attention in Equation 2.18 and permuting by a permutation σ, the attention mechanism computes the output as

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK_\sigma^T}{\sqrt{d_k}}\right) V_\sigma. \tag{3.9}$$

Due to the invariance of the softmax function and matrix multiplication under permutation, the output is simply the reordered version of the original; in other words,

the mechanism is permutation equivariant. This property is valuable in the portfolio allocation task, where the model must maintain consistency across different asset orderings. The attention mechanism also inspired the Sensory Neuron (Tang and Ha, 2021), where agents were successfully trained on different tasks using varying permuted observations.

3.5 Baselines

To benchmark the performance of our DRL agents, we consider three widely recognized methods for the portfolio allocation task, namely the Markowitz min-variance portfolio, the equal-weighted portfolio, and the buy-and-hold strategy.

Remember that we try to find weights $\mathbf{w_t}$ for each discrete step t that maximize the expected portfolio return. The Markowitz mean-variance portfolio formulates this as a static optimization problem (Markowitz, 1952). It aims to maximize expected portfolio returns for a given level of risk, or, equivalently, minimize risk for an expected target portfolio return. Risk is typically modeled as the variance of the portfolio returns, while the expected return is assumed to be stationary and estimated from historical data. Hence, mathematically, the Markowitz solution optimizes

$$\begin{aligned} \min_{\{\mathbf{w_t}\}_{t=0}^{T}} \quad & \mathrm{Var}[\mathbf{w_t}^\top \mathbf{r_t}] \\ \text{s.t.} \quad & \mathbb{E}[\mathbf{w_t}^\top \mathbf{r_t}] = k, \end{aligned} \tag{3.10}$$

where $\mathbf{r_t}$ are the individual asset returns on day t and k is the target return. If we do not want to formulate k, we can use the global minimum variance portfolio, outlined in Back, 2010. This is a subtype of the mean-variance portfolio where we remove the target return constraint. Then, using covariance matrix Σ and a vector of ones $\mathbf{1}$, the global minimum variance portfolio weights are given by

$$\mathbf{w}^* = \frac{\Sigma^{-1}\mathbf{1}}{\mathbf{1}\Sigma^{-1}\mathbf{1}}. \tag{3.11}$$

The equal-weighted portfolio is another popular baseline, but it is much simpler than the global minimum variance portfolio. At each time step t in the planning horizon, we divide the total available wealth equally over each of the n assets. Therefore, the weights become

$$\mathbf{w_t} = \frac{\mathbf{1}}{n}, \quad \forall t \in \{1, \ldots, T\}, \tag{3.12}$$

again using the vector of ones $\mathbf{1}$. As prices of assets fluctuate, we need to rebalance on every step t to maintain the equal weights. This portfolio is simple, yet effective, as it effectively spreads out risk over the assets in the portfolio. In many cases it is really hard to outperform such a strategy due to its simplicity (Clarke, De Silva, and Thorley, 2011).

The buy-and-hold strategy offers another simple approach. In this method, the investor initializes the portfolio with a vector of weights $\mathbf{w_0}$ and holds these allocations constant throughout the investment horizon T, regardless of market dynamics or portfolio performance. Formally,

$$\mathbf{w_t} = \mathbf{w_0}, \quad \forall t \in \{1, \ldots, T\}. \tag{3.13}$$

This strategy avoids transaction costs entirely and is inherently passive, making it a common benchmark for evaluating active strategies. However, it lacks adaptability

to changing market conditions, potentially leading to suboptimal performance in volatile or evolving markets. Furthermore, volatility is expected to increase, as some holdings might have disproportionally increased in value and take up a large part of the portfolio approaching the end of the horizon T. In bullish markets, however, this strategy is very hard to beat.

In our experiments, we also employ a random agent, or random policy, as a baseline. This strategy simply outputs random allocations. This is done by drawing a vector of 11 random numbers between 0 and 1 and normalizing these into a proper weight vector.

3.6 Performance Metrics

To evaluate the performance of portfolio strategies, we use several widely recognized metrics from quantitative finance.

First, we use the Sharpe ratio, which measures the risk-adjusted return of a portfolio and is defined as

$$SpR = \frac{\mathbb{E}[R_p - R_f]}{\sigma_p}, \tag{3.14}$$

where R_p is the portfolio return, R_f is the risk-free rate, and σ_p is the portfolio's standard deviation. A higher Sharpe ratio indicates better risk-adjusted performance. In practice, we often set R_f to zero.

Second, we utilize the Sortino ratio, which is a refinement of the Sharpe ratio that focuses on downside risk. It is given by

$$StR = \frac{\mathbb{E}[R_p - R_f]}{\sigma_{\text{down}}}, \tag{3.15}$$

where σ_{down} is the standard deviation of negative returns. This metric penalizes only downside volatility.

The third performance metric is the Calmar ratio. This metric captures the return per unit of maximum drawdown, and is defined as

$$CR = \frac{\mathbb{E}[R_p]}{\text{MDD}}, \tag{3.16}$$

where MDD is the maximum drawdown, which is the largest peak-to-trough decline of the portfolio's value over the evaluation period. Higher values indicate better performance relative to drawdowns.

We also use the annual return and annual volatility to benchmark portfolio strategies, providing standardized measures of performance and risk across different investment approaches. The annual return quantifies the annualized performance of an investment based on a series of daily periodic returns, assuming 252 trading days per year. It is defined as:

$$\text{AR} = \left(\prod_{t=1}^{n} (1 + r_t) \right)^{\frac{252}{n}} - 1, \tag{3.17}$$

where r_t represents the daily asset return for each of n observations. This formula compounds the returns over the sample period and scales them to an annual equivalent. The annual volatility measures the risk of the investment strategy by scaling

the standard deviation of daily portfolio returns, denoted σ_p, to an annual basis:

$$\text{AV} = \sigma_p \cdot \sqrt{252}. \tag{3.18}$$

Here, σ_p is the sample standard deviation of the daily returns, and the factor $\sqrt{252}$ annualizes the volatility under the assumption of uncorrelated returns over time.

3.7 Experimental Setup

Our experiments systematically analyze different facets of applying DRL to the portfolio allocation task. We conduct these experiments by adjusting the parameters outlined in Table 3.1 individually, while maintaining the remaining parameters at their default values.

Hyperparameter	**Default Value(s)**
Algorithm	DDPG, PPO, SAC
Training Seed	12, 42, 1234
Number of Training Securities N	100
Initial Cash	1,000,000
Number of Securities n	10
Lookback Window Length Features l_X	1
Lookback Window Step	1
Lookback Window Length Contextual l_C	1
Reward Function $\mathcal{R}$	log_return
Reward Scaling	70
Transaction Cost	0.05%
Observation Space $\mathcal{O}$	Features and previous weights
Replay Buffer Size (DDPG, SAC)	1,000,000
Warmup Steps Before Learning (DDPG, SAC)	10,000
Consolidation Layer Architecture	Linear(64)
Embedding Vector Dimension	10
Use Attention Layer	False
Hidden Layer Dimension	256
Conv1D Kernel Size	3
Learning rate (DDPG, SAC)	0.0003
Learning rate (PPO)	0.00025

TABLE 3.1: Hyperparameter Settings for Experiments.

We use warmup steps before learning starts for DDPG and SAC to prevent premature convergence to the first sampled portfolio.

In general, our experiments are run for 3 seeds. This is on the small end of the spectrum, but as runs of $5M$ steps often take about 30 to 40 hours to complete, we have to make some concessions. This work was performed using the ALICE compute resources provided by Leiden University. We have trained our algorithms using an NVIDIA A100 GPU and 64 GB of RAM. To keep an organized overview of all the different experiment results, tracked metrics, and trained models, we use Weights and Biases (Biewald, 2020).

When plotting learning curves, the runs are averaged over the seeds and error bars are plotted with them. Due to the noisy nature of financial returns, these error bars constitute 0.1 times the standard deviation of the rewards.

When we perform statistical tests, we use Bonferroni-corrected p-values. This correction is a statistical technique used to address the problem of multiple hypothesis testing, which becomes critical when evaluating numerous performance metrics or comparisons simultaneously, which is something we do in our experiments. The Bonferroni method adjusts the significance level α by dividing it by the number of tests (k), such that the new threshold becomes α/k.

Chapter 4

Experiments

In this chapter, we evaluate the performance of the DRL algorithms. First, we assess their learning capabilities by training them on portfolios of varying sizes, using learning curves to examine training speed and stability in Section 4.1. Once we establish the stability across different portfolio sizes, we explore the impact of exploration when training using an environment built on top of a deterministic financial dataset in Section 4.2. Next, we evaluate the out-of-sample performance of the agents, comparing their results to various baseline strategies, and conduct an in-depth analysis of their behavior in Section 4.3. In Section 4.4, we also investigate the permutation equivariance of the current network, aiming to ensure that different input permutations lead to similar portfolio allocations. In Section 4.5, we investigate the effect of scaling the size of the training pool on model generalization. In addition, in Section 4.6, we examine the performance and behavior of the algorithms under different transaction cost regimes. Finally, in Section 4.7 and Section 4.8, we enhance the observations by incorporating contextual market information and a lookback window in the feature set. Our goal is to offer a robust framework for testing DRL agents and to provide a thorough understanding of the impact of different DRL configurations in the portfolio optimization task.

4.1 Scaling Portfolio Size

This experiment serves as a test of the ability of DDPG, PPO, and SAC to learn the dynamics of portfolio allocation on a pool of $N = 100$ stocks. We gradually scale the size of the portfolio to see how these algorithms behave and cope with larger observation and action spaces. Specifically, we analyze their performance on portfolios of size 1, 4, 7, and 10, all including a cash position. For example, in the case of a portfolio of size $n = 1$, the task is to determine the allocation among one stock and a cash position. For DDPG and SAC, we also track how they perform on even larger portfolios of size 20 and 50. Except for n, all other hyperparameters are fixed, including those of the network architecture.

In Figure 4.1, we present learning curves of DDPG, PPO, and SAC for different portfolio size (n) scenarios over $T = 5M$ timesteps. Note that the scale of the y-axis is logarithmic. The red dotted line approximates the performance of a random policy, where one would equally distribute its capital over the assets in the portfolio. The gray line represents the optimal policy, which is a strategy where we invest all capital in the best performing asset in the portfolio on each day. In essence, this strategy has perfect knowledge of the future, and is very unrealistic. However, it affords us an adequate intuition on how well our agents learn the mapping from asset features to returns. Reaching the optimal policy would indicate perfect overfitting to the training data. Note that the value of the optimal policy goes up as the

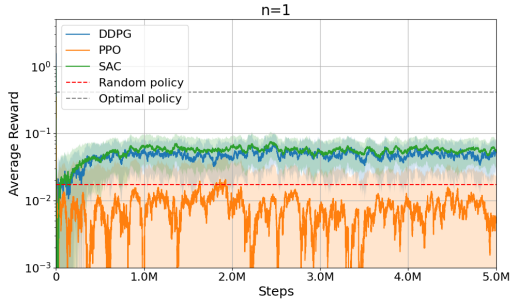
number of stocks in the portfolio increases, as there is a higher chance of sampling a well-performing stock in a larger portfolio.

From the figures, we observe that PPO performs either worse than or comparable to a random policy. While it demonstrates some learning at the start, there is no evidence of overfitting to the training data. This suggests that PPO struggles to effectively capture the underlying dynamics of the training data and identify which stock features contribute to higher returns. This limitation may arise from PPO's on-policy nature, which requires fresh trajectories for each update. Lacking a replay buffer, PPO may face difficulties with the random sampling of portfolios, hindering its ability to generalize effectively across diverse portfolio configurations.

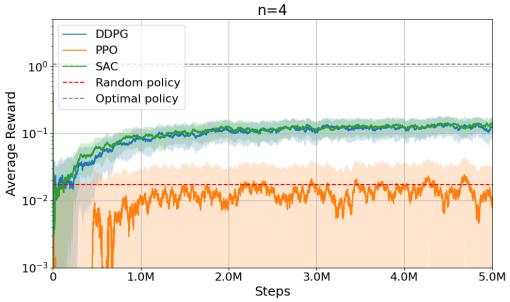
Its off-policy counterparts, DDPG and SAC, achieve higher rewards. Their capacity to fit well to the training data indicates their ability to find patterns between the features of the companies in the portfolio and their profitability. Unlike PPO, their off-policy nature allows them to reuse past experiences, enabling better exploration of the feature space and effectively generalizing across portfolios. Although they do not converge to the optimal policy, the portfolios developed by the end of the training period prove highly profitable, transforming an initial investment of 1 million into a range of 50 million to 1 billion over only 10 years.

For a portfolio size of $n = 10$, both DDPG and SAC are able to learn the underlying dynamics, where DDPG slightly outperforms SAC toward the end of training. PPO, however, shows initial promise but quickly plateaus, achieving rewards comparable to a random policy for most of the training period. This shows an inability to learn the attributes of a 10-stock portfolio.

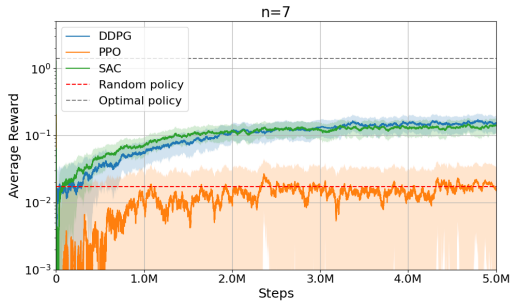
As the portfolio size increases to 20 and 50 stocks, the performance of DDPG and SAC begins to deteriorate. When portfolios grow, the attribution of a single stock return on the total return declines, making it harder for algorithms to pinpoint valuable stocks within a portfolio. By $n = 50$, both DDPG and SAC struggle to learn meaningful dynamics, leading to performance around the random policy. While the dilution of individual stock signals is a significant challenge, designing a more informative reward function, such as a risk-adjusted one, could enhance the algorithms' ability to discern valuable patterns. Another potential explanation is that the consolidation layer, comprising 64 nodes, may lack sufficient capacity to handle the growing complexity of larger portfolios, which produce a total output of 500 values (10 per stock). Enhancing the capacity of this layer could improve the algorithms' learning performance.



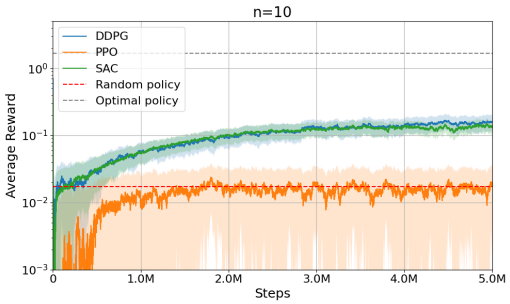
(A) Learning curves for $n = 1$.



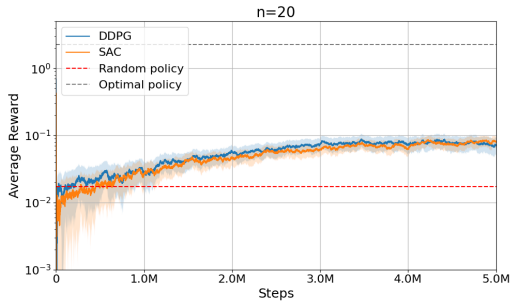
(B) Learning curves for $n = 4$.



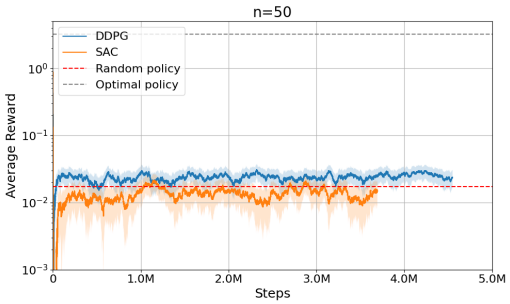
(C) Learning curves for $n = 7$.



(D) Learning curves for $n = 10$.



(E) Learning curves for $n = 20$.



(F) Learning curves for $n = 50$.

FIGURE 4.1: Learning curves for DDPG, PPO, and SAC for $n \in [1, 4, 7, 10, 20, 50]$, trained on $N = 100$ stocks, with three different seeds each ($s \in [12, 42, 1234]$). The scale of the y-axis is logarithmic. We also show the average performance of a random agent that would invest an equal amount in each of the assets in the portfolio, and an optimal policy, which invests in the best performing stock in the portfolio on each day. We can see that PPO performs poorly and that DDPG and SAC perform equally well. None of the algorithms reaches the optimal policy.

To investigate the problematic learning of PPO a little further, we present Figure 4.2. For this experiment, we have trained PPO on a highly simplified portfolio of size $n = 1$, selecting stocks from a pool of size $N \in [1, 10]$, over a training period of $T = 1M$ timesteps. By stripping the task down to its bare essentials, we hope to pinpoint the root causes of PPO's suboptimal performance.

When we sample from $N = 1$ stocks, PPO learns very steadily. The decline after 400K steps is an interesting phenomenon, possibly indicating a search for stochasticity in the policy while PPO is overfitting to this particular stock. When the stock pool is expanded to $N = 10$, PPO's performance deteriorates dramatically from the

outset. We can conclude that it fails to generalize across portfolios, likely due to the lack of a replay buffer, as found in DDPG and SAC. In other words, an on-policy algorithm likely fails to provide the diverse data needed to develop a robust, stock-agnostic strategy.

Since PPO learns on $N = 1$, the underlying neural network architecture does not appear to be the primary issue. The ability to learn a valid policy suggests that the model's structure is sufficient for a simple environment. Of course, this conclusion assumes that the architecture remains equally suitable as the task scales, but we have observed that this is the case for DDPG and SAC. Given PPO's inability to generalize across a modest pool of 10 stocks, coupled with its sensitivity to hyperparameters, we opt to exclude it from further experiments. While tuning hyperparameters might yield improvements, the fundamental algorithmic challenges of PPO suggest that DDPG and SAC may offer greater promise, especially when experiments grow more complex.

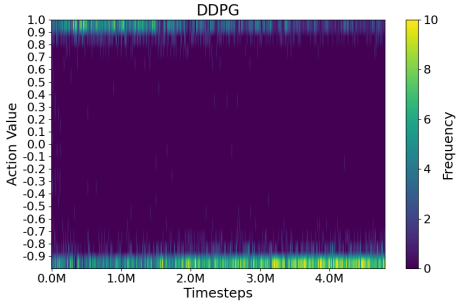
FIGURE 4.2: Learning curves for PPO for a portfolio of $n = 1$, trained on $N \in [1, 10]$ stocks, with only one seed (12). The scale of the y-axis is logarithmic. We can see that PPO fails to generalize across portfolios when the number of stocks we sample from increases.

In Figure 4.3, we show the distribution of the 11 actions that DDPG, PPO, and SAC perform at intervals of 10000 for the portfolio of size $n = 10$ over the training period of $T = 5M$. It serves as a guide for the behavior and exploration strategies of these algorithms during training.

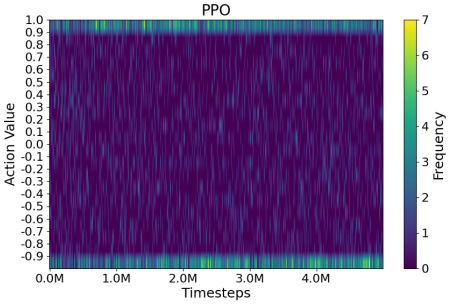
DDPG's reliance on action noise for exploration is evident in the relatively unchanging distribution of actions over time. It also tends to take the extreme actions (-1, and 1). The noise introduces a normal distribution-like spread near the action boundaries, indicating that deviations from the policy's preferences are noise-driven rather than learned. PPO remains stochastic over the full training period, as it is taking actions over the full range, with a slight preference for extreme actions. This sustained diversity reflects PPO's inherent exploration mechanism, where actions are sampled from a probability distribution. It also highlights the failure of PPO to converge to a more exploitative policy. SAC similarly explores the full range of actions, using the bucket from -0.9 to -1.0 more towards the end of the training period. This trend might indicate that SAC is learning to divest from assets yielding negative returns, clarifying its strength towards the end of the training period.

To conclude, DDPG relies on external noise for exploration, resulting in a rigid yet consistent action distribution. PPO and SAC leverage stochastic policies, where

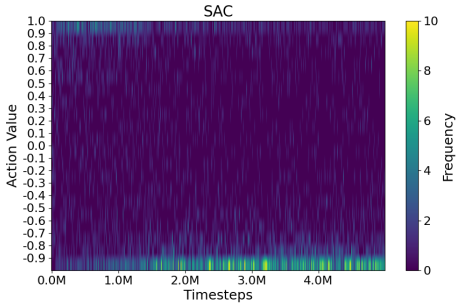
we observe that PPO seems incapable of converging to a more rewarding policy and SAC divests from suboptimal assets. The shared preference for extreme actions across all three may reflect the reward function, which prioritizes significant adjustments, as more weight in the best performing asset directly and linearly translates to a higher reward. To put this into perspective: If we were to employ a risk-adjusted reward function, allocating more weight to a single asset would not inherently yield a higher reward. This is because concentrating weight in one asset increases the portfolio's volatility and risk.



(A) Distribution of actions of DDPG.



(B) Distribution of actions of PPO.



(C) Distribution of actions of SAC.

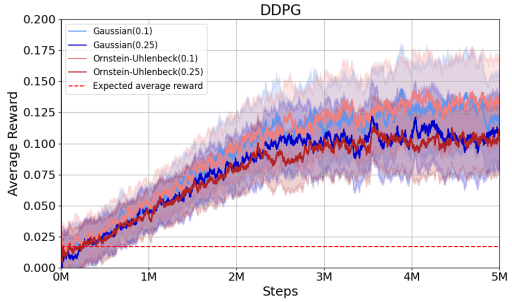
FIGURE 4.3: Distribution of actions of DDPG, PPO, and SAC for $n = 10$, trained on $N = 100$ stocks, and $s = 42$. The frequency is shown as a scale on the right side of the plot.

Now that we have seen that off-policy algorithms are able to learn from the inherently complex feature distributions in finance across varying portfolio sizes, we turn to investigate the impact of exploration in an environment built on a deterministic financial dataset.

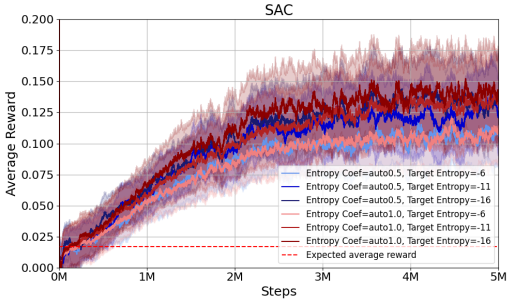
4.2 Varying Exploration Parameters

An important aspect of any RL algorithm is its strategy to explore the environment. In a fixed dataset, such as the financial dataset we are using, exploration is not as intuitive as in a game, for example, where the agent fails to see a part of the state space without proper exploration. When using a fixed dataset, we will serve the agent with the complete set of states, regardless of his actions, except for some action-dependent state components, like the agent's previous weights or its available capital.

This experiment aims to investigate the role of exploration in RL when applied to a deterministic dataset. We hypothesize that the agent fundamentally requires a minimal degree of exploration to construct a reliable mapping from states to actions



(A) Learning curves for DDPG for different exploration settings. We have trained DDPG using Gaussian noise (blue) and Orstein-Uhlenbeck noise (red), with varying $\sigma \in [0.1, 0.25]$. Darker colors are higher values for σ.



(B) Learning curves for SAC for different exploration settings. We vary the initial entropy coefficient (0.5 in blue, 1.0 in red) and the target entropy. Darker colors indicate a higher target entropy.

FIGURE 4.4: Learning curves for DDPG and SAC for different exploration parameters, trained on $N = 100$ stocks, with three different seeds each ($s \in [12, 42, 1234]$). We have again included a random policy. We observe that exploration influences algorithm performance minimally.

and that the specific exploration strategy employed will have a limited impact on the agent's performance. A comprehensive exposure to a diverse set of portfolio allocations is sufficient to develop this mapping.

For each algorithm, we tune distinct exploration parameters to evaluate their impact. In the case of DDPG, exploration is done by introducing noise to its deterministic actions. We experiment with two widely used noise processes, namely Gaussian noise and Ornstein-Uhlenbeck noise, the latter being temporally correlated. For both noise types, we adjust the σ parameter, which controls the magnitude of variability in the noise. We anticipate that Ornstein-Uhlenbeck noise will be more effective for the portfolio optimization task, as it helps prevent excessive fluctuations in weights between timesteps, which reduces the risk of incurring high transaction costs and chasing illusive, short-lived patterns.

SAC uses stochastic policies to balance exploration and exploitation. We tune both the initial entropy coefficient and the target entropy, the latter defining the entropy level the algorithm seeks to maintain during training. Over time, SAC learns the optimal value for the entropy coefficient, so we do not expect this parameter to substantially influence learning.

All parameter settings are informed by established practices in the RL field and tailored to the specific action space used in our experiments. In particular, we select σ values that align with the [-1, 1] range of the action space. For SAC, we have found that entropy coefficients are often chosen between 0 and 1, and the target entropy is set around the size of the action space, which is 11 in our case.

We observe that learning is influenced marginally by exploration, which is in line with our hypothesis. For SAC, learning is slightly improved in higher exploratory settings. For DDPG, higher exploration leads to lower rewards, possibly because random noise leads to higher transaction costs. There is no clear difference between Gaussian noise and Ornstein-Uhlenbeck noise. This shows that the type of noise does not contribute significantly in the process of forming a rewarding policy. In both settings, exploration doesn't seem to accelerate learning significantly.

For SAC, the entropy coefficient seems to have little influence on learning performance, with slightly higher rewards when using the entropy coefficient of 1.0,

where 'auto' simply means that SAC optimizes the coefficient to achieve the target entropy. The target entropy does influence the learning process, where higher (more negative) values tend to improve rewards. In this case, the agent has likely seen more diverse state-action pairs and has learned which actions contribute to higher rewards.

Further investigation into the relationship between the entropy coefficient and the target entropy of SAC shows us that the entropy coefficient rapidly diminishes to values between 0.0001 and 0.001, which are much lower than its initial settings of 0.5 and 1.0. This is probably due to noisy financial rewards and a high inherent entropy in the rewards. This partly explains why the initial setting has little influence on the learning process.

In the rest of the experiments, we use the settings corresponding to the most stable learning process and the least complexity. For DDPG, we continue to use Gaussian noise with $\sigma = 0.1$. Although initial entropy does not influence learning a lot for SAC, we opt for a value of 1.0. We set the target entropy as the default for an action space of dimension 11, namely at -11.

Even though we have excluded PPO from the analysis, we have run some experiments where we tune the entropy coefficient, analyzing whether this parameter could help PPO generalize better across sampled portfolios. The entropy coefficient is a regularizer that encourages entropy in the policy and prevents premature convergence to a deterministic policy. It should help with choosing more stochastic policies (Ahmed et al., 2019), possibly policies that generalize better. We have included this analysis in Appendix B.

4.3 Out Of Sample Performance

So far, we have analyzed the in sample learning performance of DDPG, PPO, and SAC in our portfolio optimization task. Our findings indicate that DDPG and SAC effectively leverage company features to develop well-performing investment strategies, achieving high logarithmic returns. PPO, being an on-policy algorithm, struggles with the random portfolio sampling and encounters difficulties when the total training pool modestly increases to $N = 10$. DDPG and SAC demonstrate the ability to generalize across diverse portfolios, establishing a stock-agnostic mapping from company features to portfolio returns. This in-sample validation using learning curves, often omitted in the existing literature, is essential before we assess the out-of-sample performance of these algorithms.

In most RL tasks, agents are trained, validated and tested on the same task. There is no need to partition the data into separate sets, as game engines, for example, maintain consistent dynamics. In finance, however, it is crucial to validate model performance on a separate piece of data. As markets evolve constantly, the training, validation, and test sets can exhibit substantial differences, placing significant stress on the algorithms. Achieving robust out of sample performance requires algorithms that generalize well, and avoid overfitting to unique dynamics of the training set. Therefore, this experiment serves both as a gauge of the performance of our algorithms when they would be deployed in the live market, and as a metric of the generalizability to new market dynamics and stocks.

In this experiment, we continue to train on a pool of $N = 100$ stocks, as this choice balances computational complexity by reducing the number of stocks compared to larger sets, while preliminary findings suggest it supports robust performance. In the following sections, we experiment with training on larger pools. In

Figure 4.5, we show the performance of DDPG and SAC on a separate evaluation set, for three runs with seeds $s \in [12, 42, 1234]$. To improve generalizability of the algorithms, we sample portfolios from the full universe of 1464 stocks available at the start of the validation period, such that the algorithms could be assessed on more than 1,300 previously unseen stocks, in theory. Due to computational constraints, we evaluate performance on 100 portfolios, every 2520 training steps [1]. The sample of portfolios is always the same across steps to maintain comparability. To avoid overfitting, we have implemented an early stopping mechanism with a patience of 20 evaluation steps.

From the figure, we can make two primary observations. First, the randomly initialized model, which is evaluated at step 0, performs really well. Further investigation shows that this model behaves like an equally-weighted portfolio, allocating capital equally over assets. As the general stock market has increased significantly over the evaluation period, as we observed in Figure 3.1, this strategy is really good. Second, we see that SAC achieves higher episodic rewards than DDPG, and also surpasses the initial setting. This shows that SAC not only learns from the company features in the training data, but also succeeds in generalizing to new and previously unseen stocks. DDPG exhibits more erratic behavior, especially around $800K$ steps. On average, it doesn't seem to surpass the initial evaluation, which shows that it faces difficulties in generalizing to the evaluation data.

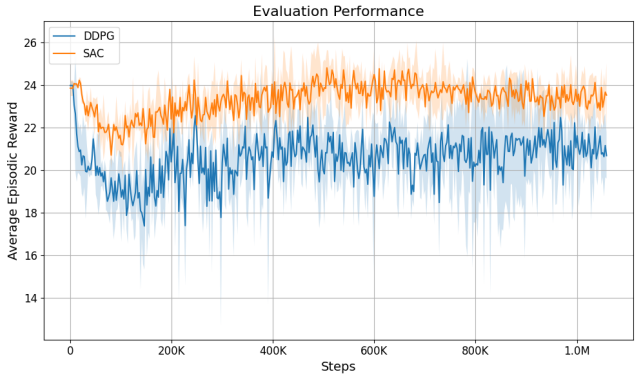


FIGURE 4.5: The performance of DDPG and SAC on 100 evaluation portfolios per evaluation step, sampled from a pool of size $N_{\text{val}} = 1464$ stocks, trained on $N_{\text{train}} = 100$ stocks, and run for 3 seeds ($s \in [12, 42, 1234]$). Note that the y-axis shows the episodic reward, which is the sum of all the step rewards within an episode and the x-axis are the training steps. We can see that a randomly initialized model already performs really well and that SAC achieves higher evaluation rewards than DDPG.

For all three seeds, we take the model that achieves the highest evaluation reward and test it on a separate test set, averaging per algorithm to improve statistical validity. As we are interested in the generalizability, we sample 200 portfolios from the full universe of size $N_{\text{test}} = 1465$ [2]. In this experiment, we use the first six months of data for the estimation of the covariance matrix for the global minimum variance

[1] As we only sample 100 portfolios, we do not see every stock in the evaluation. Specifically, the expected percentage of sampled stocks is $\left(1 - \left(1 - \frac{10}{1464}\right)^{100}\right) \cdot 1464/1464 \cdot 100\% \approx 50\%$.

[2] We expect to sample $\left(1 - \left(1 - \frac{10}{1465}\right)^{200}\right) \cdot 1465/1465 \cdot 100\% \approx 75\%$ of the total pool of stocks.

portfolio (MinVar). Later, we also use offsets for lookback windows. Hence, we start testing from July 2022 to facilitate a fair comparison between algorithms.

The results of the testing phase are presented in Table 4.1 and Figure 4.6. Specifically, we present the average Sharpe, Sortino, and Calmar ratios, and the annual return and volatility. We also include the total rewards, which constitute the sum of the step rewards over the testing period. In the figure, we show the cumulative portfolio returns averaged over the 200 portfolios. In essence, the cumulative returns indicate the portfolio value at each timestep as a fraction of the initial value. In Table A.1 in Appendix A, we present Bonferroni-corrected p-values to provide a statistical assessment of the results in Table 4.1.

TABLE 4.1: Mean and standard error of the mean (SEM) of metrics for different algorithms and baselines, trained on a pool of $N = 100$ stocks. The best configuration is shown in bold.

	DDPG	SAC	BuyAndHold	EqualWeighted	MinVar	Random
Sharpe	0.470 ± 0.020	0.501 ± 0.019	**0.627 ± 0.028**	0.561 ± 0.028	0.552 ± 0.027	0.136 ± 0.030
Sortino	0.714 ± 0.031	0.755 ± 0.029	**0.937 ± 0.044**	0.843 ± 0.042	0.827 ± 0.042	0.210 ± 0.044
Calmar	0.387 ± 0.020	0.421 ± 0.020	**0.545 ± 0.033**	0.477 ± 0.031	0.468 ± 0.031	0.093 ± 0.025
Annual Return	8.40% ± 0.49%	8.88% ± 0.44%	**11.68% ± 0.70%**	9.81% ± 0.62%	9.32% ± 0.59%	1.05% ± 0.59%
Annual Volatility	21.87% ± 0.14%	21.05% ± 0.12%	20.09% ± 0.16%	20.01% ± 0.17%	19.32% ± 0.16%	**18.87% ± 0.16%**
Total Rewards	10.347 ± 0.623	11.130 ± 0.550	**14.797 ± 0.834**	12.543 ± 0.766	11.963 ± 0.734	0.996 ± 0.790

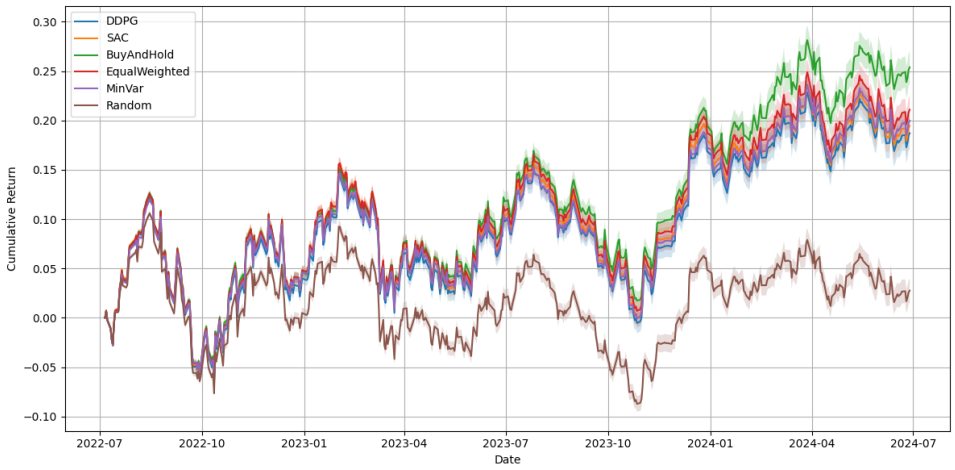


FIGURE 4.6: The cumulative returns of DDPG, SAC, and the baselines averaged over 200 test portfolios sampled from a pool of size $N_{\text{test}} = 1465$ stocks. The error bounds are based on the standard error of the mean (SEM). We can see that the Buy and Hold strategy outperforms all other strategies and that the Random agent performs worst.

From the figure, we notice that the BuyAndHold strategy stands out with a steady climb to approximately 0.25 by mid-2024, outpacing all others, especially from the start of 2024. The EqualWeighted and MinVar strategies, and DDPG and SAC all seem to perform equally well, with marginal deviations. The Random strategy languishes near or below zero, underscoring its ineffectiveness, probably due to high transaction costs with frequent reallocation of capital.

The table underscores BuyAndHold's superior performance across all metrics, with the exception of annual volatility, where the Random strategy exhibits the lowest value due to its consistently poor performance. As BuyAndHold does not rebalance, its notable surge in performance could be attributed to the significant gains of a few high-performing stocks, which other strategies fail to fully capture due to

their periodic rebalancing. This is partially evidenced by BuyAndHold's marginally higher volatility compared to EqualWeighted, MinVar, and Random, suggesting a potential overexposure to these outperformers. Statistical significance tests reveal that BuyAndHold significantly outperforms DDPG and SAC on most metrics. BuyAndHold does not significantly outperform MinVar and EqualWeighted. Among the actively managed strategies (DDPG, SAC, and MinVar), no significant differences emerge, except for annual volatility, where the Minimum Variance (MinVar) portfolio justifies its name by achieving the lowest risk. Market volatility during the test period (2022–2024), which is characterized by geopolitical conflicts, is likely to favor the passive approach of BuyAndHold, while RL algorithms struggle to adapt to these dynamic conditions. In a bullish market, a straightforward passive strategy could leverage broad trends more effectively than RL models trained using DDPG and SAC. These algorithms might overcomplicate decision-making or falter in noisy, non-stationary environments.

Figure 4.7 illustrates the sector allocations for SAC, DDPG, and baseline strategies, defined as the ratio of weight allocated to stocks in various sectors. It provides insights into the investment behavior of the strategies and contextualizes previously presented performance metrics. For instance, the Random strategy's low volatility can be attributed to its consistently high allocation to cash, an inherently stable asset. Using the sector allocations, we can also analyze the outperformance of the BuyAndHold strategy. As detailed in Table 4.2, BuyAndHold's allocation to Industrials increases by 19% over the test period, while other strategies maintain or slightly reduce their exposure. This surge in performance, as BuyAndHold is a passive strategy, reflects the relative appreciation of the Industrials sector compared to other sectors. The other algorithms do not capitalize on this trend by increasing their exposure to this sector, potentially explaining their inferior performance. Another notable trend not mirrored by the other algorithms is the increased exposure to the Consumer Cyclicals sector, which represents a substantial proportion of assets within the U.S. market.

A potential reason why DDPG and SAC didn't fully capitalize on these trends could be a change in feature distributions over time. Markets evolve and features that were important previously may diminish in relevance. This stresses the importance of more frequent retraining, possibly on a monthly basis. We defer this investigation to future research.

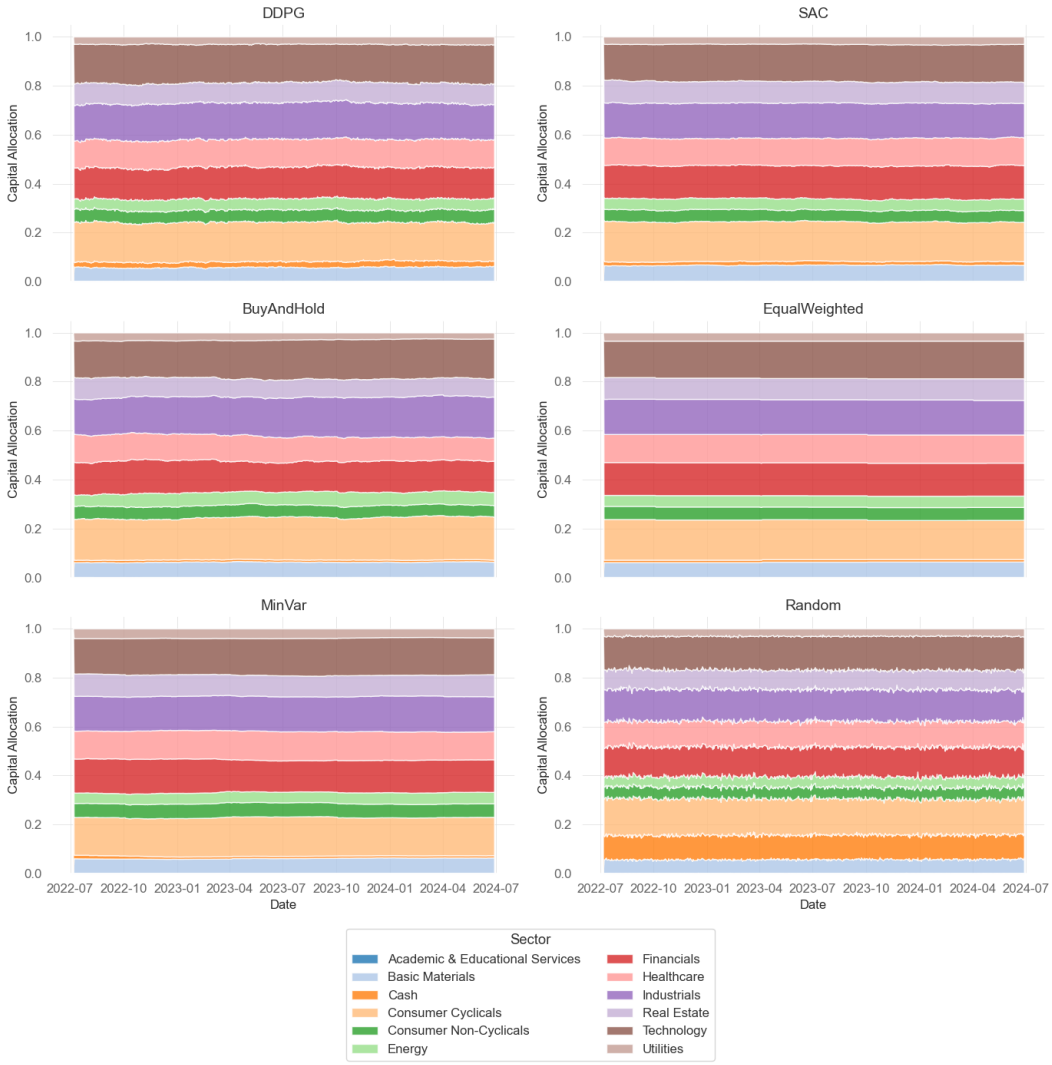


FIGURE 4.7: Time-series allocation of capital over sectors and a cash position for DDPG, SAC, and baseline strategies, evaluated and averaged over 200 portfolios over the test period (July 2022 to July 2024). Each strategy's capital allocation is represented as a stacked area plot. The plot illustrates the behavior of active strategies (DDPG, SAC, MinVar, Random) compared to the static allocations of BuyAndHold and EqualWeighted, and their potential preference for different sectors. We observe minimal changes, possibly due to the averaging out of idiosyncrasies of the different strategies.

TABLE 4.2: Percentage changes in sector allocations over the test period.

	DDPG	SAC	BuyAndHold	EqualWeighted	MinVar	Random
Academic & Educational Services	-8.62%	-18.86%	8.96%	0.00%	-7.07%	28.18%
Basic Materials	3.11%	0.99%	2.58%	4.13%	6.34%	11.56%
Cash	-6.32%	-29.08%	-15.61%	0.00%	-36.47%	1.25%
Consumer Cyclicals	-3.67%	-3.02%	6.05%	-3.61%	1.43%	-2.59%
Consumer Non-Cyclicals	0.98%	2.64%	-12.68%	-1.87%	-3.69%	0.53%
Energy	12.62%	1.56%	18.97%	3.33%	10.21%	-3.47%
Financials	0.38%	3.26%	-5.35%	0.37%	-4.19%	4.45%
Healthcare	2.55%	1.48%	-17.42%	0.44%	0.68%	-0.78%
Industrials	-1.83%	-1.84%	16.89%	-1.74%	0.19%	-6.13%
Real Estate	-4.13%	-6.66%	-16.31%	0.00%	-0.79%	-3.08%
Technology	1.27%	7.33%	9.40%	3.02%	4.14%	0.54%
Utilities	4.10%	4.56%	-25.86%	0.00%	-8.34%	8.45%

To further illustrate the variability in performance and highlight the pitfalls of limited benchmarking, we examine the distribution of annual returns across multiple portfolios. Figure 4.8 showcases the risks associated with presenting results from a single model applied to a single portfolio. In the figure, we see a box plot of annual returns for each algorithm across 200 sampled portfolios. This analysis reveals that DDPG achieves both the highest and lowest annual returns among all strategies, indicating that reporting only one portfolio could misleadingly position DDPG as the superior performer across all strategies. For example, benchmarking only the best-performing DDPG portfolio could lead to the mistaken conclusion that it outperforms SAC and other baselines, potentially even on risk-adjusted metrics.

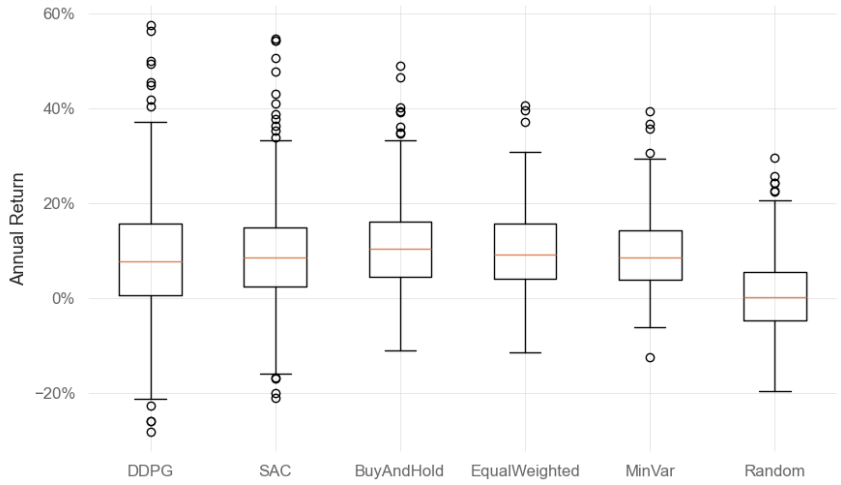


FIGURE 4.8: A boxplot displaying the annual return distributions for DDPG, SAC, and baseline strategies, evaluated across 200 portfolios sampled from a pool of $N_{\text{test}} = 1465$ stocks. The results reveal that DDPG and SAC exhibit both lower and higher annual returns compared to the baselines, reflecting the inherent variability when evaluating financial strategies.

In conclusion, this experiment reveals that, despite the superior generalization and stability of SAC over DDPG when tested on 1,465 unseen stocks after training on just 100, both fall short of the passive BuyAndHold strategy in the volatile market of 2022-2024. BuyAndHold delivers high returns, manages downside risk, and exceeds other strategies in mitigating drawdowns. Other strategies do not effectively capitalize on the growing prominence of the Industrials sector, which contributes to BuyAndHold's superior performance.

In the following experiments, we aim to identify the precise reasons for this observation. Specifically, since BuyAndHold does not incur transaction costs, we will investigate whether transaction costs contribute to the suboptimal performance of the RL algorithms. Another option is that training on a pool of $N = 100$ stocks does not suffice and that the algorithms would benefit from learning on a larger pool. In addition, integrating contextual data, such as macroeconomic variables, could enhance performance in volatile markets, adding an additional layer of knowledge about the context of the wider market. Finally, incorporating a lookback window may enable the algorithms to develop a more stable policy by providing additional

historical information on companies, which helps with more deliberate and longer-term portfolio reallocations, addressing the long-term dependency problem in finance.

Before proceeding there, we first investigate the network architecture. Specifically, we aim to determine whether the current network is permutation equivariant, meaning that different permutations of stock inputs should result in the same allocations.

4.4 Permutation Equivariance

In this experiment, we investigate the properties of the algorithm architecture concerning input order. When sampling random portfolios, each stock is assigned to an input node of the network. Given the use of shared weights across stocks, the feature extraction process remains stock-agnostic, and we ensure its proper functioning by incorporating layer normalization, which effectively handles varying distributions of company features across the network.

However, following feature extraction, we concatenate the embedding vectors for each stock. This concatenation operation introduces order dependency. In portfolio management, stock order should ideally be inconsequential; the weights should remain consistent regardless of input order. Given the narrow profit margins typical in finance, such aberrations can substantially impair overall performance. Currently, the consolidation layer lacks input agnosticism, or more precisely, permutation equivariance. When stocks A and B are swapped, the corresponding actions or weights are not similarly rearranged, indicating that the consolidation layer inadvertently learns from the input order, despite its irrelevance. This limitation arises as a drawback of employing a linear layer.

This experiment investigates architectural modifications aimed at enhancing the permutation equivariance of the network. Since a linear layer cannot achieve full permutation equivariance, our focus is on facilitating this property within the consolidation layer. Specifically, we explore two architectural adjustments: varying the dimensionality of embedding vectors and incorporating an attention layer following the feature extraction stage. With smaller embedding vectors, it may become more challenging to encode order information, potentially reducing the consolidation layer's sensitivity to input sequence post-extraction. Furthermore, we explore the use of an attention mechanism as a potential remedy, leveraging its theoretical permutation equivariance to potentially manage input order complexity and facilitate learning in the consolidation layer.

In Figure 4.9, we present a box plot illustrating the weight differences between the original weights on a trading day and those obtained after permuting the input stocks, based on 100 distinct permutations and evaluated across 504 trading days for the different network architectures aimed at improving permutation equivariance. These differences are computed as the absolute difference in weight between the original and the permuted weights. The labels indicate the embedding dimension and a boolean value specifying whether the attention layer is used. A limitation is that we have trained only one model per configuration due to computational constraints. The results clearly indicate substantial variations in weights when the input

order is altered, a behavior that is undesirable. Notably, weight differences occasionally exceed those of a random agent, suggesting that the algorithms may be overfitting to the specific order of inputs. Regarding architectural design, smaller embedding dimensions are associated with greater weight differences, likely due to an increased propensity for the consolidation layer to overfit when using smaller embedding vectors. Furthermore, the inclusion of an attention layer markedly enhances permutation equivariance, as evidenced by reduced weight differences across all embedding dimensions.

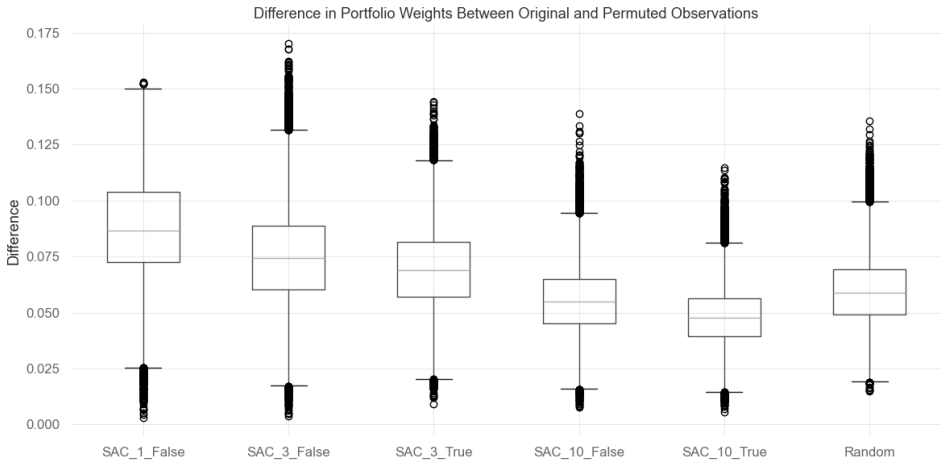


FIGURE 4.9: A box plot illustrating the distributions of weight differences between 100 permutations and the original configuration for various SAC architectures, evaluated for 504 trading days. The architecture names indicate the embedding dimension and a boolean value specifying whether the attention layer is used. We observe that the incorporation of an attention layer and larger embedding dimensions effectively reduce these weight differences.

We now examine the influence of greater weight differences on out-of-sample performance, presenting performance metrics for various SAC architectures evaluated across 200 sampled portfolios from a pool of size $N_{\text{test}} = 1465$. Learning curves assessing training stability across these configurations are provided in Appendix B. The test results for different SAC network architectures are detailed in Table 4.3, accompanied by statistical significance tests in Appendix A.2. In particular, the configuration with an embedding dimension of 10 and an attention mechanism outperforms the other configurations on most metrics; however, the statistical significance tests in the appendix indicate that these findings are not statistically significant. Nonetheless, we can see a correlation between high performance and reduced weight differences, underscoring our hypothesis. Interestingly, the model with an embedding dimension of 1 achieves the highest annual returns. Using multiple models in this experiment may contribute to more robust results.

TABLE 4.3: Mean and standard error of the mean (SEM) of metrics for different network architectures for SAC. The best configuration is shown in bold.

	SAC_1_False	SAC_3_False	SAC_3_True	SAC_10_False	SAC_10_True	Random
Sharpe	0.533 ± 0.035	0.460 ± 0.032	0.469 ± 0.033	0.522 ± 0.032	**0.557 ± 0.032**	0.136 ± 0.030
Sortino	0.807 ± 0.054	0.692 ± 0.049	0.708 ± 0.050	0.785 ± 0.049	**0.840 ± 0.049**	0.210 ± 0.044
Calmar	0.458 ± 0.037	0.374 ± 0.033	0.398 ± 0.034	0.438 ± 0.033	**0.483 ± 0.035**	0.093 ± 0.025
Annual Return	**9.93% ± 0.87%**	7.98% ± 0.75%	8.17% ± 0.76%	9.14% ± 0.73%	9.90% ± 0.72%	1.05% ± 0.59%
Annual Volatility	21.77% ± 0.23%	20.78% ± 0.19%	20.85% ± 0.20%	20.58% ± 0.20%	20.48% ± 0.20%	**18.87% ± 0.16%**
Total Rewards	12.275 ± 1.080	9.998 ± 0.941	10.209 ± 0.963	11.513 ± 0.921	**12.502 ± 0.899**	0.996 ± 0.790

In conclusion, we observe that weight changes decrease with the incorporation of an attention mechanism and by maintaining a higher embedding dimension. Out-of-sample performance decreases with additional variability, but not on a statistically significant level. To fully address the issue of weight changes resulting from input permutation, we can consider several architectural and training modifications. One approach to achieve complete weight consistency involves extending weight sharing throughout the network architecture, by that means eliminating the need for a consolidation layer. With this approach, we would design actor and critic networks that process each stock individually via shared parameters, incorporating contextual information and previous weights, and directly mapping to actions, possibly through a softmax function to ensure action space constraints. This would mean that we do not model interactions between assets anymore, which is something that we hypothesize is important in portfolio optimization. Further research is needed to evaluate this architecture.

Alternatively, a Set Transformer could be used (Lee et al., 2019). This architecture offers an inherently permutation-equivariant solution. By employing attention mechanisms to model all pairwise interactions within the input set of stocks, Set Transformers naturally produce order-agnostic per-element features that can directly be translated into weights. This mechanism would replace the consolidation layer. Another option is the Graph Neural Network (GNN) (Scarselli et al., 2008), such as the Graph Convolutional Network (Kipf and Welling, 2017). While stock data is often represented as a set of independent time series, a GNN can exploit relationships between stocks to improve learning. However, using complex architectures such as the Set Transformer and GNNs requires careful implementation and validation, which we leave for further investigation.

Beyond architectural changes, we can also change the training methodology to improve permutation robustness. For example, we could permute actions and observations within the replay buffer, which would expose the model to a wider range of input orders, potentially forcing it to learn order-invariant representations. However, whether this would fit within the RL methodology is questionable. Upon first impression, this would not break the Bellman equations or other RL fundamentals, as long as the permutations are consistent across the experience tuples. Another, potentially simpler option would be to decrease the training frequency in SAC, which could promote the sampling of more diverse portfolios across training iterations, as we simply sample more portfolios over time. This would mitigate overfitting to specific input sequences observed early in training and implicitly enhance robustness to input permutations. This does bring more computational complexity.

In the next few experiments, we hope to gain a deeper understanding on what impacts the performance of our DRL algorithms. Particularly, we vary the training pool size and transaction costs, and we change the observation space by including contextual market information and a lookback window.

4.5 Scaling Training Pool Size

This experiment serves as a measure for how the total pool of stocks influences the performance of DRL algorithms. Our hypothesis is that with a larger N_{train}, algorithms become better at generalization and achieve a higher performance as a result, simply because they are exposed to a wider variety of feature distributions. Therefore, we gradually increase the number of stocks in the training pool from 100 to 500 to 1469.

In Figure 4.13, we present the learning curves for SAC across various pool sizes N. The analysis reveals that training on a larger pool of stocks introduces increased complexity, necessitating effective learning of a greater number of features and enhanced generalization capabilities. Specifically, SAC fails to establish a stable policy when trained on $N = 1469$ stocks, potentially due to technical challenges such as an insufficiently large network to accommodate the wider variation in feature distributions, excessive conflicting signals within the state, or inadequate training duration. In Appendix B, we explore the network size as a potential factor, but as this was not the cause, we continue to hypothesize about the underlying reason for this learning limitation. Although we do not investigate the issue of conflicting states, a look-back window might address this effectively. For the present study, we proceed with $N = 500$, as this size effectively demonstrates the point of this experiment while remaining computationally feasible.

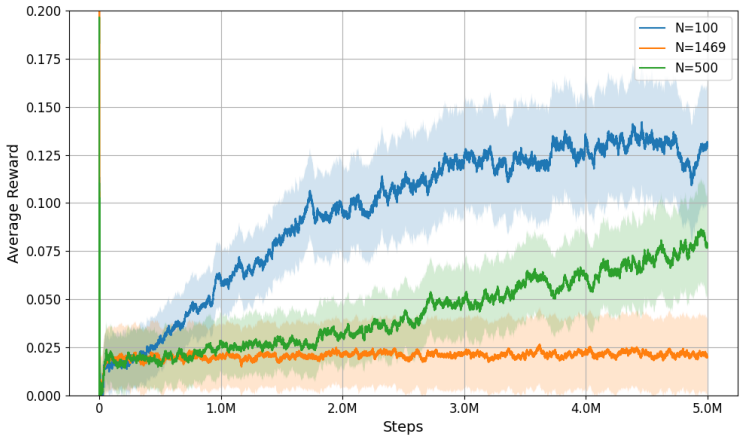


FIGURE 4.10: Learning curves for SAC, trained on different pool sizes $N \in [100, 500, 1469]$ stocks, with three different seeds each ($s \in [12, 42, 1234]$). We observe that an increased pool size brings more complexity in learning a feasible policy, leading to prolonged or even failed learning. The green spike at the beginning is caused by the moving average having insufficient data early on, leading to delayed smoothing.

In Figure 4.11, we present the performance of DDPG and SAC on a distinct evaluation set. The evaluation interval has been extended to 12,600 steps to accommodate the prolonged training times associated with a larger stock pool. The results indicate that DDPG shows a notable improvement from training on a larger number of stocks compared to the $N = 100$ case, although it remains somewhat more erratic compared to SAC. SAC, on the other hand, achieves slightly higher evaluation rewards and consistently outperforms the randomly initialized model, attaining its peak performance around 2.5 million steps.

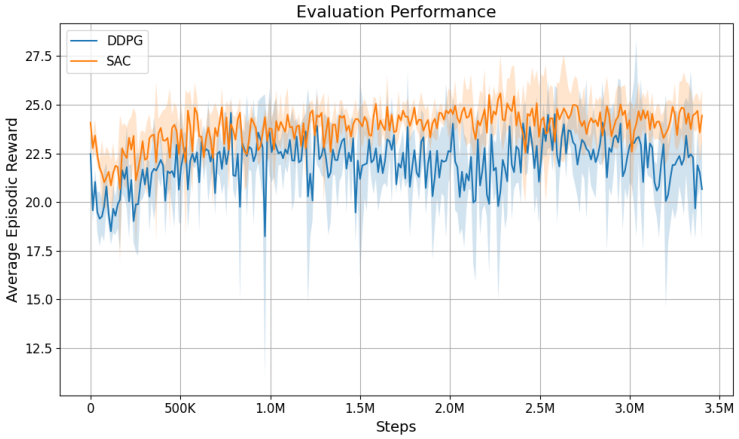


FIGURE 4.11: The performance of DDPG and SAC on 100 evaluation portfolios per evaluation step, sampled from a pool of size $N_{\text{val}} = 1464$ stocks, trained on $N_{\text{train}} = 500$ stocks, and run for 3 seeds ($s \in [12, 42, 1234]$). We can see that SAC achieves higher evaluation rewards than DDPG and that these rewards are slightly higher than the $N = 100$ case. DDPG particularly seems to improve from training on more stocks.

In Table 4.4, we present the usual performance metrics, but now for models trained on a pool of $N = 500$ stocks, accompanied by statistical significance test results in Table A.3 in Appendix A. The values for the baseline strategies remain unchanged. An interesting observation emerges from the table: both DDPG and SAC show degraded performance compared to the $N = 100$ scenario, which is unexpected given that training on a larger stock pool should theoretically enhance generalization and performance. To understand this phenomenon, we must first acknowledge the increased complexity associated with a larger pool. With a broader range of stocks, more diverse portfolios can be sampled, reinforcing the issue of asset permutation sensitivity compared to training on $N = 100$. Limited exposure to permutations may lead the network to incorrectly learn that order matters, despite its irrelevance. Another possibility is that the network struggles to accommodate a wider variety of feature distributions, particularly if the consolidation layer is of insufficient size. The size of the feature extractor probably is not the underlying issue, as scaling the hidden layer dimension for $N = 1469$ proved ineffective.

The other issues that we discussed for the $N = 1469$ case might also surface here, albeit in a milder form. For instance, the state space may become more conflicted, causing the network to learn noise rather than meaningful signals. In addition, with a larger dataset, the model might overfit more easily to the evaluation set's dynamics, which may not align with those of the test set. To put it more broadly, the heightened heterogeneity of the training data could disrupt learning of generalizable patterns, as the statistical properties of stock features, their correlations, and their market dynamics might become more diverse and less consistent for a 500-stock pool compared to 100. This increased diversity, potentially worsened by outlier stocks exhibiting unusual feature distributions or behaviors, may skew the learning process, particularly if the model is overly sensitive to such anomalies. In the final analysis, the increased search space introduced by a larger pool likely complicates the model's ability to identify stable, generalizable policies, favoring noise over meaningful patterns. Implementing a lookback window could provide additional context on company features, mitigating issues of data heterogeneity and conflicting states

TABLE 4.4: Mean and standard error of the mean (SEM) of metrics for different algorithms and baselines, trained on a pool of $N = 500$ stocks. The best configuration is shown in bold.

	DDPG	SAC	BuyAndHold	EqualWeighted	MinVar	Random
Sharpe	0.447 ± 0.020	0.466 ± 0.019	**0.627 ± 0.028**	0.561 ± 0.028	0.552 ± 0.027	0.136 ± 0.030
Sortino	0.679 ± 0.031	0.705 ± 0.029	**0.937 ± 0.044**	0.843 ± 0.042	0.827 ± 0.042	0.210 ± 0.044
Calmar	0.370 ± 0.020	0.380 ± 0.019	**0.545 ± 0.033**	0.477 ± 0.031	0.468 ± 0.031	0.093 ± 0.025
Annual Return	7.76% ± 0.50%	8.34% ± 0.46%	**11.68% ± 0.70%**	9.81% ± 0.62%	9.32% ± 0.59%	1.05% ± 0.59%
Annual Volatility	22.72% ± 0.18%	21.78% ± 0.12%	20.09% ± 0.16%	20.01% ± 0.17%	19.32% ± 0.16%	**18.87% ± 0.16%**
Total Rewards	9.435 ± 0.661	10.352 ± 0.586	**14.797 ± 0.834**	12.543 ± 0.766	11.963 ± 0.734	0.996 ± 0.790

to make a more robust, general policy. However, enhancing the network architecture or training methodology to address permutation equivariance appears to be a more pressing priority.

In the following experiments, we focus exclusively on SAC, as we have observed that DDPG and SAC perform similarly, with SAC showing a slight advantage. In the next experiment, we train SAC under various transaction cost regimes to examine its adaptability to different settings.

4.6 Varying Transaction Costs

In this experiment, we analyze the impact of different transaction cost schemes on the performance and behavior of SAC and the baselines. Transaction costs increase with the turnover of an investment strategy, thereby constraining an investor's ability to frequently reallocate their portfolio. Hence, frequently reallocated strategies, such as the Random strategy, are expected to degrade rapidly under higher transaction costs. Agents must adopt a longer-horizon strategy, as short-term gains become less attainable. In this study, we vary the level of transaction costs on the reallocated capital between 0, 5, 15, 25, and 35 basis points (0.01 %). In a liquid market, a level around 5 basis points is considered standard, and levels around 25 to 35 are considered very high. We train and evaluate 3 models from SAC per transaction cost scheme, trained on $N = 100$ stocks, with the expectation that it will learn to rebalance more conservatively under higher transaction costs, given that these reduce overall rewards. In Table 4.5, we show the usual metrics for the different algorithms at several transaction cost (TC) basis points (bps).

The table reveals that BuyAndHold consistently achieves the highest performance across most transaction cost regimes, which is to be expected given its passive nature as it only incurs transaction costs on the first trading day of an episode. In the absence of transaction costs, SAC demonstrates higher annual returns than MinVar and Random, highlighting its capability to identify profitable assets based on company features. However, its risk-adjusted metrics lag, potentially because the reward function lacks a risk component. The lack of incorporating risk partially accounts for its lower annual return compared to MinVar at 5 basis points, where excessive volatility and associated transaction costs reduce annual returns. As transaction costs increase to 15–35 basis points, SAC adapts by reducing its rebalancing frequency, resulting in the lowest volatility among all algorithms. Further analysis of the weight allocations indicates that SAC's behavior mirrors that of the EqualWeighted strategy, except for its utilization of a cash position, unlike EqualWeighted. This explains the near-identical performance metrics between SAC and EqualWeighted at 15, 25, and 35 basis points.

TABLE 4.5: Mean and standard error of the mean (SEM) of metrics for different algorithms at various transaction cost (TC) basis points (bps). The best configuration is shown in bold.

TC (bps)	Metric	SAC	BuyAndHold	EqualWeighted	MinVar	Random
0	Sharpe	0.539 ± 0.019	**0.628 ± 0.028**	0.570 ± 0.028	0.562 ± 0.027	0.543 ± 0.029
	Sortino	0.816 ± 0.030	**0.939 ± 0.044**	0.857 ± 0.042	0.842 ± 0.042	0.818 ± 0.045
	Calmar	0.457 ± 0.020	**0.546 ± 0.033**	0.486 ± 0.031	0.479 ± 0.031	0.465 ± 0.032
	Annual Return	9.90% ± 0.47%	**11.71% ± 0.70%**	10.01% ± 0.62%	9.54% ± 0.59%	9.02% ± 0.61%
	Annual Volatility	21.83% ± 0.15%	20.09% ± 0.16%	20.01% ± 0.17%	19.32% ± 0.16%	**18.93% ± 0.16%**
	Total Rewards	12.321 ± 0.598	**14.832 ± 0.834**	12.794 ± 0.766	12.234 ± 0.734	11.530 ± 0.774
5	Sharpe	0.501 ± 0.019	**0.627 ± 0.028**	0.561 ± 0.028	0.552 ± 0.027	0.136 ± 0.030
	Sortino	0.755 ± 0.029	**0.937 ± 0.044**	0.843 ± 0.042	0.827 ± 0.042	0.210 ± 0.044
	Calmar	0.421 ± 0.020	**0.545 ± 0.033**	0.477 ± 0.031	0.468 ± 0.031	0.093 ± 0.025
	Annual Return	8.88% ± 0.44%	**11.68% ± 0.70%**	9.81% ± 0.62%	9.32% ± 0.59%	1.05% ± 0.59%
	Annual Volatility	21.05% ± 0.12%	20.09% ± 0.16%	20.01% ± 0.17%	19.32% ± 0.16%	**18.87% ± 0.16%**
	Total Rewards	11.130 ± 0.550	**14.797 ± 0.834**	12.543 ± 0.766	11.963 ± 0.734	0.996 ± 0.790
15	Sharpe	0.543 ± 0.016	**0.624 ± 0.028**	0.543 ± 0.028	0.531 ± 0.027	-0.656 ± 0.029
	Sortino	0.815 ± 0.024	**0.933 ± 0.044**	0.815 ± 0.042	0.796 ± 0.042	-0.902 ± 0.038
	Calmar	0.468 ± 0.018	**0.543 ± 0.033**	0.459 ± 0.031	0.448 ± 0.030	-0.330 ± 0.010
	Annual Return	8.70% ± 0.32%	**11.62% ± 0.70%**	9.41% ± 0.61%	8.90% ± 0.59%	-12.85% ± 0.47%
	Annual Volatility	**18.24% ± 0.09%**	20.08% ± 0.16%	20.01% ± 0.17%	19.32% ± 0.16%	18.85% ± 0.15%
	Total Rewards	11.204 ± 0.403	**14.727 ± 0.834**	12.041 ± 0.766	11.422 ± 0.734	-19.475 ± 0.753
25	Sharpe	0.524 ± 0.016	**0.622 ± 0.028**	0.525 ± 0.028	0.511 ± 0.027	-1.474 ± 0.033
	Sortino	0.787 ± 0.024	**0.930 ± 0.044**	0.788 ± 0.042	0.765 ± 0.042	-1.960 ± 0.041
	Calmar	0.451 ± 0.018	**0.541 ± 0.033**	0.441 ± 0.031	0.429 ± 0.030	-0.685 ± 0.025
	Annual Return	8.31% ± 0.32%	**11.57% ± 0.70%**	9.02% ± 0.61%	8.47% ± 0.58%	-25.10% ± 0.44%
	Annual Volatility	**18.20% ± 0.09%**	20.08% ± 0.16%	20.01% ± 0.17%	19.31% ± 0.16%	18.88% ± 0.16%
	Total Rewards	10.715 ± 0.402	**14.657 ± 0.834**	11.538 ± 0.766	10.880 ± 0.734	-40.531 ± 0.803
35	Sharpe	0.506 ± 0.016	**0.619 ± 0.028**	0.507 ± 0.028	0.491 ± 0.027	-2.306 ± 0.036
	Sortino	0.759 ± 0.024	**0.926 ± 0.044**	0.760 ± 0.042	0.734 ± 0.041	-2.946 ± 0.041
	Calmar	0.433 ± 0.018	**0.539 ± 0.033**	0.423 ± 0.030	0.408 ± 0.030	-0.554 ± 0.003
	Annual Return	7.96% ± 0.32%	**11.51% ± 0.69%**	8.62% ± 0.61%	8.05% ± 0.58%	-35.93% ± 0.37%
	Annual Volatility	**18.21% ± 0.09%**	20.08% ± 0.16%	20.01% ± 0.17%	19.31% ± 0.16%	18.91% ± 0.16%
	Total Rewards	10.265 ± 0.403	**14.587 ± 0.834**	11.036 ± 0.767	10.339 ± 0.735	-62.168 ± 0.798

While SAC's strategy under higher transaction costs is not inherently ineffective, it is not very dynamic. The higher transaction costs appear to outweigh potential gains from rebalancing, making an equal-weight allocation the optimal policy, or the easiest to converge to. It seems that long-term dependencies are hard to find with the current feature set, and the reward function plays a big role in this. Again, incorporating a more informative reward function, possibly one that includes a risk term, could enable SAC to devise more profitable, diversified and dynamic allocations.

In Figure 4.12, we show a bar plot of the average transaction cost per step incurred by SAC and the baselines, under the same transaction cost regimes. We do not include the Random strategy, as it incurred extremely high transaction costs at each step, distorting the plot. Most notably, we see that the transaction costs per step scale linearly with the imposed transaction cost for the baselines, as these algorithms are unaware of of the imposed transaction costs. SAC, however, directly incorporates transaction costs into its reward function, allowing it to adapt dynamically. As we have seen previously, under higher transaction costs, SAC mirrors the EqualWeighted strategy with a cash position. This is also evident from the average transaction costs. Interestingly, at 5 basis points, SAC follows a more volatile policy that incurs transaction costs similar to those at 35 basis points. This behavior could suggest model misalignment or overfitting, but it may also indicate that a more volatile policy is advantageous under lower transaction costs.

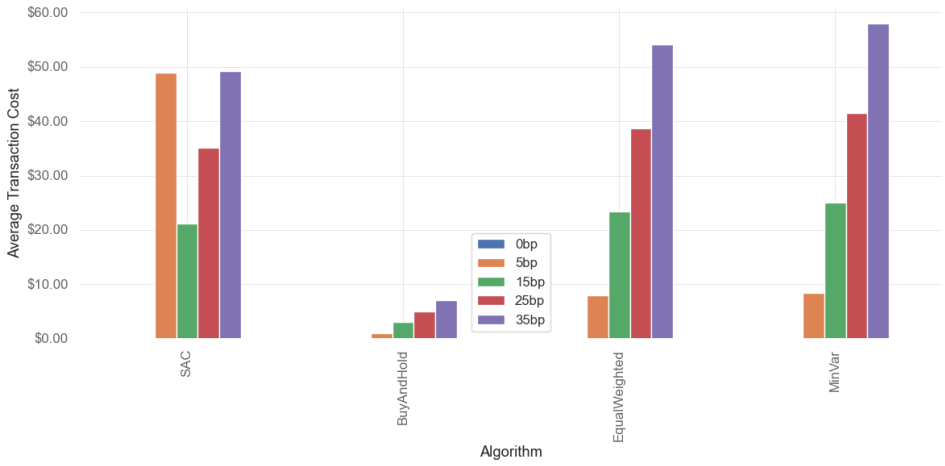


FIGURE 4.12: The average transaction cost per step for different algorithms under various transaction cost schemes (bp $\in [0, 5, 15, 25, 35]$). We exclude the Random strategy, as its transaction costs are excessively high (around \$1500 per step), which would distort the plot. For the baseline models, transaction costs per step scale linearly with the imposed transaction cost. SAC exhibits distinct behavior at 5 basis points, demonstrating its dynamic adaptability to the environment.

In the final set of experiments, we will vary the observation space to assess its impact on SAC's learning behavior and performance in the portfolio optimization task.

4.7 Contextual Market Information

This experiment modifies the observation space by incorporating contextual market information into the environment's state. These contextual variables include macroeconomic indicators such as the WTI Crude price, various Treasury bill rates, the Dollar Index (DXY), and the credit spread. We hypothesize that these variables will provide agents with a broader market context, helping them identify which company features are most relevant under different market conditions. Specifically, we expect SAC to exhibit lower volatility, as it may learn to shift allocations toward cash during periods of market turmoil.

Figure 4.13 presents the learning curves for two observation settings: the original setup (including company features and previous weights, denoted as `prevweights`) and the extended setup incorporating contextual information (denoted as `context`). We observe that SAC demonstrates stable learning behavior in both configurations.

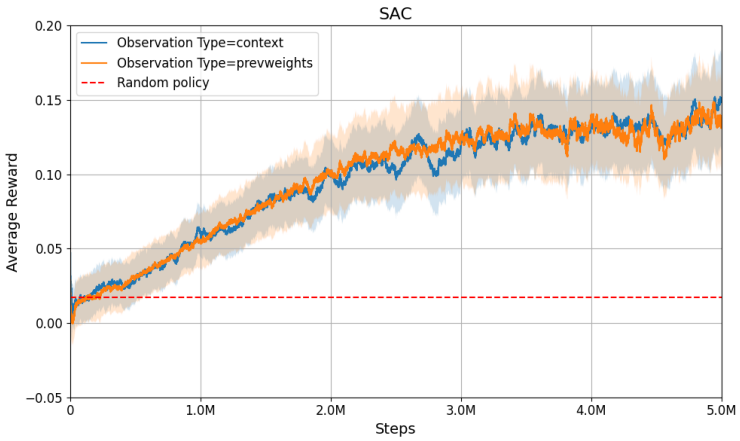


FIGURE 4.13: Learning curves for SAC, trained using two different observation spaces (previous weights and context), with three different seeds each ($s \in [12, 42, 1234]$). We observe that SAC learns stably with both observation types.

Figure 4.14 presents the evaluation performance of the two observation settings. SAC with contextual information appears to underperform slightly compared to SAC without context. Additionally, it exhibits greater instability, as indicated by the larger downward spikes in the error bounds.

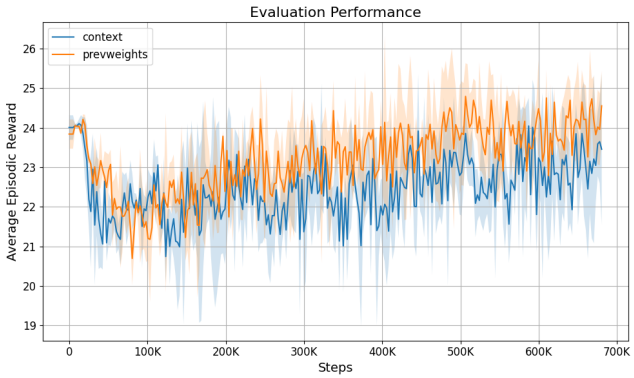


FIGURE 4.14: The performance of SAC across two observation settings, evaluated on 100 evaluation portfolios per evaluation step, sampled from a pool of size $N_{\text{val}} = 1464$ stocks, trained on $N_{\text{train}} = 100$ stocks, and run for 3 seeds ($s \in [12, 42, 1234]$). Note that the y-axis shows the episodic reward, which is the sum of all the step rewards within an episode and the x-axis are the training steps. We can see that SAC with context appears to achieve lower rewards than the original setting on average.

Table 4.6 highlights this instability, showing that SAC with context performs slightly worse than SAC without context across most metrics. Notably, SAC trained with contextual market information exhibits marginally lower annual volatility, though the difference is statistically insignificant. Given the minimal variations across other metrics, it remains unclear whether the observed differences stem from the change in the observation space or natural model variability.

Adding contextual market information does not appear to meaningfully improve SAC's decision-making in this setup. It introduces slightly more instability without a clear benefit. This suggests that SAC may not effectively utilize macroeconomic

TABLE 4.6: Mean and standard error of the mean (SEM) of metrics for different algorithms. The best configuration is shown in bold.

	SAC	SAC_context	BuyAndHold	EqualWeighted	MinVar	Random
Sharpe	0.501 ± 0.019	0.491 ± 0.018	**0.627 ± 0.028**	0.561 ± 0.028	0.552 ± 0.027	0.136 ± 0.030
Sortino	0.755 ± 0.029	0.740 ± 0.028	**0.937 ± 0.044**	0.843 ± 0.042	0.827 ± 0.042	0.210 ± 0.044
Calmar	0.421 ± 0.020	0.407 ± 0.019	**0.545 ± 0.033**	0.477 ± 0.031	0.468 ± 0.031	0.093 ± 0.025
Annual Return	8.88% ± 0.44%	8.61% ± 0.42%	**11.68% ± 0.70%**	9.81% ± 0.62%	9.32% ± 0.59%	1.05% ± 0.59%
Annual Volatility	21.05% ± 0.12%	20.93% ± 0.12%	20.09% ± 0.16%	20.01% ± 0.17%	19.32% ± 0.16%	**18.87% ± 0.16%**
Total Rewards	11.130 ± 0.550	10.838 ± 0.529	**14.797 ± 0.834**	12.543 ± 0.766	11.963 ± 0.734	0.996 ± 0.790

indicators in its learning process, possibly because these variables do not directly translate into actionable signals within the portfolio optimization task. This suggests the need for feature engineering in the indicators as well, to facilitate information extraction by the algorithms. For instance, we can use growth rates or momentum indicators in the macroeconomic variables. Furthermore, contextual market signals may be highly noisy, as shifting market regimes can obscure meaningful patterns, making it challenging for SAC to extract useful information. Incorporating a lookback window in the macroeconomic variables could provide additional context about the current market conditions and help clarify conflicting signals. The current environment is already designed to accommodate this, making further research straightforward. On top, a feature extractor for the contextual market variables could facilitate learning in the consolidation layer. Finally, the current reward function may not effectively encourage the utilization of macroeconomic indicators, as it is not designed to account for volatility. In its current form, these indicators are unlikely to enhance logarithmic returns, they might even introduce unnecessary complexity. However, their value may become more apparent in reducing risk when paired with a risk-adjusted reward function. We will leave this for further investigation.

In the last experiment, we investigate whether incorporating a lookback window in the company features helps improve the performance of SAC.

4.8 Lookback Window

In this section, we experiment with a lookback window in the features in the observation space. Including a lookback window can be useful because it gives the agent more context about what happened in previous steps. In essence, the agent has a sequence of past observations of the company features to its disposal, aiding in detecting patterns or trends that might not be obvious from only the most recent state. Including a lookback window can lead to improved learning and more robust performance, especially when delayed effects are at play, which is the case in financial markets.

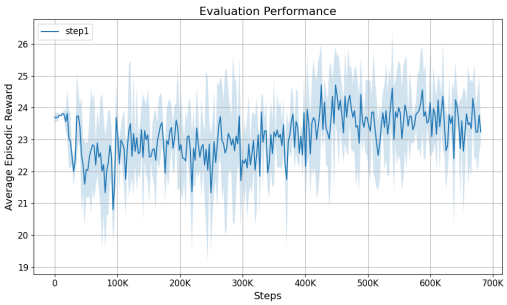
We also hypothesize that a lookback window can help disambiguate observations by providing the agent with more context about a particular portfolio configuration. This becomes particularly relevant as the number of stocks in the training pool increases. While the enhanced robustness of a lookback window may be beneficial in this scenario, we do not explicitly test for this in our experiments and leave it for future research.

However, lookback windows become less useful when the state already includes well-designed features, such as financial returns or momentum, which naturally capture time-dependent patterns. In these cases, the added computational cost of a lookback window may not be justified, as the agent's neural network can already rely on concise and informative inputs to make decisions without needing to reconstruct past trends from raw data. Since our dataset includes high-quality features,

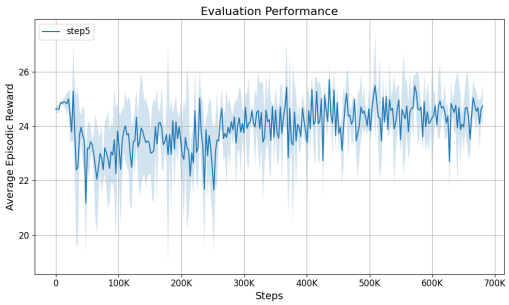
the primary benefit of a lookback window may be improved stability and reduced ambiguity for the agent.

Due to computational constraints, as discussed in Chapter 3, we use relatively short lookback windows of length 4. While including more steps is probably beneficial, it requires more computational resources than we have at our disposal. To further investigate the effect of a lookback window, we vary the step size between consecutive observations, using step sizes of 1, 5, and 21, corresponding to a trading day, week, and month, respectively. A larger step size exposes the agent to a longer time frame in the market, potentially enhancing its robustness to short-term fluctuations. In addition, longer-term dependencies should become easier to identify, helping to address one of the fundamental challenges of applying DRL in finance through the use of a lookback window.

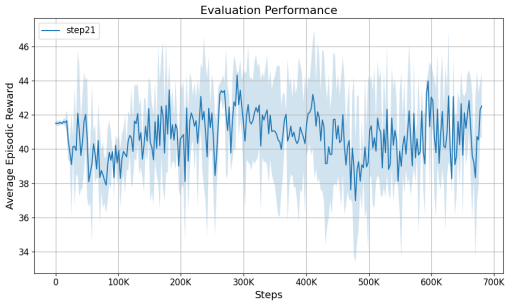
Figure 4.15 presents the evaluation performance across the three lookback window step sizes. In all cases, we observe an improvement over the initial model, though the effect is more pronounced for step sizes 1 and 21. The performance for step size 21 is erratic and declines after $300K$ training steps, suggesting high training variability for this setting.



(A) SAC with a lookback window with a step size of 1 (one trading day).



(B) SAC with a lookback window with a step size of 5 (one trading week).



(C) SAC with a lookback window with a step size of 21 (one trading month).

FIGURE 4.15: The performance of SAC across three lookback window step sizes, evaluated on 100 evaluation portfolios per evaluation step, sampled from a pool of size $N_{\text{val}} = 1464$ stocks, trained on $N_{\text{train}} = 100$ stocks, and run for 3 seeds ($s \in [12, 42, 1234]$). Note that the y-axis shows the episodic reward, which is the sum of all the step rewards within an episode and the x-axis are the training steps, and is not equal between the plots. We can not directly compare the level of the curves, as the lookback window influences the start of the evaluation period (as we have to incorporate a window in the observations) and hence the average performance. We can see that each setting shows an improvement over the training period. A step size of 21 leads to more fluctuations and a downward trend around $300K$ steps, indicating training variability.

In Table 4.7, we show the standard performance metrics for four different observation spaces in SAC: the baseline setting without a lookback window and three settings with a lookback window using different step sizes. The results indicate that incorporating a lookback window leads to slightly lower annual volatility, and according to Table A.8, this difference is statistically significant between SAC and SAC_1. Among the evaluated models, SAC_5 achieves the highest Sharpe, Sortino, and Calmar ratios, suggesting that incorporating information from previous weeks helps in learning a more profitable and risk-aware policy. However, most observed differences are not statistically significant. This could be due to several factors, including the relatively short length of the lookback window. With longer lookback windows, and additional convolutional layers to process them, the agent might have extracted more useful information from past states, potentially improving its trading strategy. While the marginal improvements observed for SAC_5 are promising, the challenge of permutation sensitivity may be limiting further gains and we probably need a more suitable architecture before exploring more advanced observation

spaces. An alternative explanation is that lookback windows only start to make sense when training on a larger pool of stocks or with a different reward mechanism.

TABLE 4.7: Mean and standard error of the mean (SEM) of metrics for different lookback window step sizes. The best configuration is shown in bold.

	SAC	SAC_1	SAC_5	SAC_21
Sharpe	0.501 ± 0.019	0.496 ± 0.018	0.516 ± 0.019	0.501 ± 0.019
Sortino	0.755 ± 0.029	0.747 ± 0.028	0.779 ± 0.028	0.757 ± 0.029
Calmar	0.421 ± 0.020	0.416 ± 0.019	0.442 ± 0.020	0.411 ± 0.019
Annual Return	8.88% ± 0.44%	8.63% ± 0.42%	9.11% ± 0.43%	8.97% ± 0.45%
Annual Volatility	21.05% ± 0.12%	20.57% ± 0.11%	20.66% ± 0.12%	20.90% ± 0.12%
Total Rewards	11.130 ± 0.550	10.875 ± 0.523	11.460 ± 0.536	11.222 ± 0.558

Chapter 5

Conclusion

This thesis explores the application of Deep Reinforcement Learning (DRL) algorithms to the complex task of portfolio allocation. Our primary goal is to evaluate the performance of several state-of-the-art DRL agents, namely DDPG, PPO, and SAC, under various conditions and to identify key factors influencing their ability to learn effective investment strategies. Rather than seeking a single optimal configuration or portfolio, we offer a comprehensive analysis of the elements that can enhance DRL agents in this context, including the incorporation of lookback windows, contextual information, feature engineering, and different exploration strategies.

We have designed our experiments to cover key challenges in DRL for finance, including high-dimensional state spaces, long-term dependencies, limited data availability, sample efficiency, training instability, and generalization, with the goal of understanding the effectiveness and current standing of DRL in portfolio allocation. We focus on validating the stability and generalizability of these models in a statistically robust framework, prioritizing these aspects over mere raw performance. Through this investigation, we have identified core challenges inherent to DRL, particularly applied to finance, such as the importance of out-of-sample generalization, the need to mitigate permutation sensitivity in network architectures when handling data that can be structured as sets, the maintenance of model stability, and the influence of transaction costs.

To address the issue of limited data, we have engineered 116 features from raw financial pricing, fundamentals and sentiment data (Chakraborty, 2019). On top, we sample portfolios from a pool of 1,500 stocks in the S&P 1500, enhancing generalization across portfolios and assets while improving data efficiency (Wang, 2019). To further refine the models, we designed a shared feature extractor for each company, incorporating an optional attention layer to improve permutation equivariance in a dense network (Tang and Ha, 2021). Finally, a consolidation layer integrates these company-specific features with contextual market data and previous portfolio weights, offering a comprehensive representation of portfolio potential.

In this chapter, we first provide a brief conclusion of the main findings. Then, we put these findings into perspective, discussing their implications for the field and outlining potential directions for future research.

5.1 Main Findings

The experiments carried out in this thesis yield several important insights.

1. When scaling the portfolio size, off-policy algorithms like DDPG and SAC demonstrate a greater capacity to learn profitable strategies compared to the on-policy PPO. We conclude that a replay buffer is essential when sampling

diverse portfolios and when generalization to diverse assets is desired. However, the performance of DDPG and SAC deteriorates as the portfolio size increases significantly, suggesting limitations in handling very high-dimensional action spaces, primarily due to the dilution of signals to the reward function.

2. The impact of varying exploration parameters in a deterministic financial dataset is marginal, indicating that sufficient exposure to the state space might be more critical than extensive exploration of novel actions. However, this can also originate from the reward function, as using logarithmic portfolio returns promotes extreme actions and more complex reward functions possibly require more exploration.

3. The out-of-sample evaluation reveal that while SAC shows better generalization capabilities than DDPG, both algorithms underperform the simple BuyAndHold strategy during the testing period. Analysis of sector allocations suggests that the passive BuyAndHold strategy benefits from the appreciation of specific sectors, a trend not fully captured by the actively managed DRL agents.

4. The network architecture exhibits sensitivity to the order of input stocks, highlighting a lack of permutation equivariance which negatively impacts performance. Although incorporating an attention layer helps, it does not fully mitigate the problem. The issue of asset orderings is even more evident when increasing the size of the training pool. Training on larger stock pools, DDPG and SAC show less generalization and more instability, highlighting the need for a robust network architecture and training methodology.

5. Examining the influence of transaction costs demonstrates that SAC can adapt its trading strategy to the environment, mirroring a low-turnover strategy similar to EqualWeighted under high transaction costs. However, BuyAndHold consistently outperforms across most transaction cost regimes due to its inherently low turnover.

6. Incorporating contextual market information into the observation space does not lead to significant performance improvements, suggesting that these macroeconomic indicators, in their current form, may not be directly beneficial without further processing or a more tailored reward function.

7. A lookback window in the features does not directly lead to statistically significant performance gains, except for volatility, which is lower than without a lookback window.

5.2 Future Work

In this thesis, we have explored various aspects of developing a robust investment strategy using DRL. Key design choices include the state space, action space, reward function, network architecture, and learning algorithms. While we have addressed several of these areas, many promising directions remain open for future research. Through the development of our robust statistical framework for evaluating DRL agents, we empower researchers to systematically explore and refine these design choices within the portfolio allocation task.

Using our framework, we have found that while DRL algorithms, particularly off-policy methods, show promise in learning complex financial dynamics, they face

significant challenges in outperforming simple, well-established baseline strategies like BuyAndHold, especially in volatile market conditions. We hypothesize that the primary issue is the sensitivity to input order (lack of permutation equivariance) in standard network architectures, an issue that needs to be addressed for better and more robust performance. We have identified several solutions, both architectural and methodological. The most promising avenue would be the adoption of a fully permutation equivariant network, such as a Graph Neural Network (Scarselli et al., 2008), which also models asset interactions. Simpler approaches would be to reduce the training frequency in SAC and DDPG or to permute asset and action ordering directly in the replay buffer such that the algorithms are exposed to more diverse portfolio configurations.

Another important research direction constitutes the design of the reward function, as it plays a crucial role in shaping the agent's investment behavior. The popular logarithmic return used in this thesis might not properly incentivize desirable characteristics such as more dynamic allocations in high transaction cost regimes or adaption to contextual market information. Alternative reward functions, such as the differential Sharpe ratio, the value at risk, or the Kelly criterion, could promote more dynamic behavior and can be seamlessly integrated into our framework. These reward functions can also help the agent focus on longer-term patterns rather than short-lived and elusive gains, leading to more stable investment behavior.

With respect to the action space, we have found that using actions as direct portfolio weights allows DRL agents to make more targeted allocations, but it also introduces extra complexity. An interesting direction for future research is to explore whether defining actions as weight changes could be effective, particularly in market environments with high transaction costs, where this structure might lead to more efficient trading.

Although we have experimented with different observation spaces, including contextual information and a lookback window, an important next step is to analyze the impact of individual features on the agent's decisions. A feature sensitivity analysis can help identify the most influential inputs, improving the interpretability of DRL models. One approach is to compute the partial derivatives of the action vector with respect to each input feature, revealing how changes in features affect the agent's decisions (Benhamou, 2023). Furthermore, Shapley values provide a model-agnostic method to quantify the contribution of each feature to the agent's output (Shapley, 1953).

Apart from diverse portfolios, it would also be valuable to train on a more diverse set of date ranges than those used in this thesis. Benhamou, 2023 and Sood et al., 2023 employ a sliding training window, gradually sliding the evaluation and test sets forward in time. This approach enables researchers to assess DRL algorithms across a broader range of market conditions, improving the robustness of experiments.

Finally, while we have explored various aspects individually, integrating these elements could lead to even better performance and training stability. For example, a lookback window coupled with contextual information, and training on a larger pool of assets could create synergies between these design choices. The extent of these synergies remains an open question and requires further empirical validation, a process which is greatly facilitated by our robust framework.

Appendix A

Significance Tests

A.1 Out of Sample Performance

TABLE A.1: Bonferroni-corrected p-values from Student's t-tests comparing algorithms across various metrics, based on training with a pool of $N = 100$ stocks. Statistically significant p-values at the 5% level are highlighted in bold.

Metric	Algorithm	DDPG	SAC	BuyAndHold	EqualWeighted	MinVar	Random
Sharpe	DDPG	-	0.263	**0.000***	0.008	0.017	**0.000***
	SAC	0.263	-	**0.000***	0.072	0.127	**0.000***
	BuyAndHold	**0.000***	**0.000***	-	0.098	0.056	**0.000***
	EqualWeighted	0.008	0.072	0.098	-	0.802	**0.000***
	MinVar	0.017	0.127	0.056	0.802	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Sortino	DDPG	-	0.323	**0.000***	0.014	0.029	**0.000***
	SAC	0.323	-	**0.001***	0.088	0.157	**0.000***
	BuyAndHold	**0.000***	**0.001***	-	0.122	0.069	**0.000***
	EqualWeighted	0.014	0.088	0.122	-	0.790	**0.000***
	MinVar	0.029	0.157	0.069	0.790	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Calmar	DDPG	-	0.216	**0.000***	0.015	0.026	**0.000***
	SAC	0.216	-	**0.001***	0.130	0.194	**0.000***
	BuyAndHold	**0.000***	**0.001***	-	0.134	0.089	**0.000***
	EqualWeighted	0.015	0.130	0.134	-	0.843	**0.000***
	MinVar	0.026	0.194	0.089	0.843	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Annual Return	DDPG	-	0.469	**0.000***	0.075	0.229	**0.000***
	SAC	0.469	-	**0.001***	0.219	0.544	**0.000***
	BuyAndHold	**0.000***	**0.001***	-	0.045	0.010	**0.000***
	EqualWeighted	0.075	0.219	0.045	-	0.569	**0.000***
	MinVar	0.229	0.544	0.010	0.569	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Annual Volatility	DDPG	-	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***
	SAC	**0.000***	-	**0.000***	**0.000***	**0.000***	**0.000***
	BuyAndHold	**0.000***	**0.000***	-	0.752	**0.001***	**0.000***
	EqualWeighted	**0.000***	**0.000***	0.752	-	**0.003***	**0.000***
	MinVar	**0.000***	**0.000***	**0.001***	**0.003***	-	0.046
	Random	**0.000***	**0.000***	**0.000***	**0.000***	0.046	-
Total Rewards	DDPG	-	0.346	**0.000***	0.027	0.094	**0.000***
	SAC	0.346	-	**0.000***	0.136	0.365	**0.000***
	BuyAndHold	**0.000***	**0.000***	-	0.048	0.011	**0.000***
	EqualWeighted	0.027	0.136	0.048	-	0.586	**0.000***
	MinVar	0.094	0.365	0.011	0.586	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-

A.2 Permutation Equivariance

TABLE A.2: Bonferroni-corrected p-values from Student's t-tests comparing different SAC architectures across various metrics, based on training with a pool of $N = 100$ stocks. Statistically significant p-values at the 5% level are highlighted in bold.

Metric	Algorithm	SAC_1_False	SAC_3_False	SAC_3_True	SAC_10_False	SAC_10_True	Random
Sharpe	SAC_1_False	-	0.128	0.189	0.820	0.611	**0.000***
	SAC_3_False	0.128	-	0.841	0.173	0.033	**0.000***
	SAC_3_True	0.189	0.841	-	0.252	0.057	**0.000***
	SAC_10_False	0.820	0.173	0.252	-	0.438	**0.000***
	SAC_10_True	0.611	0.033	0.057	0.438	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Sortino	SAC_1_False	-	0.115	0.176	0.760	0.653	**0.000***
	SAC_3_False	0.115	-	0.826	0.180	0.034	**0.000***
	SAC_3_True	0.176	0.826	-	0.268	0.060	**0.000***
	SAC_10_False	0.760	0.180	0.268	-	0.427	**0.000***
	SAC_10_True	0.653	0.034	0.060	0.427	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Calmar	SAC_1_False	-	0.092	0.234	0.690	0.620	**0.000***
	SAC_3_False	0.092	-	0.617	0.174	0.026	**0.000***
	SAC_3_True	0.234	0.617	-	0.401	0.085	**0.000***
	SAC_10_False	0.690	0.174	0.401	-	0.354	**0.000***
	SAC_10_True	0.620	0.026	0.085	0.354	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Annual Return	SAC_1_False	-	0.091	0.128	0.489	0.980	**0.000***
	SAC_3_False	0.091	-	0.862	0.267	0.065	**0.000***
	SAC_3_True	0.128	0.862	-	0.355	0.098	**0.000***
	SAC_10_False	0.489	0.267	0.355	-	0.460	**0.000***
	SAC_10_True	0.980	0.065	0.098	0.460	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Annual Volatility	SAC_1_False	-	**0.001***	**0.003***	**0.000***	**0.000***	**0.000***
	SAC_3_False	**0.001***	-	0.805	0.471	0.274	**0.000***
	SAC_3_True	**0.003***	0.805	-	0.347	0.191	**0.000***
	SAC_10_False	**0.000***	0.471	0.347	-	0.728	**0.000***
	SAC_10_True	**0.000***	0.274	0.191	0.728	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Total Rewards	SAC_1_False	-	0.114	0.155	0.593	0.872	**0.000***
	SAC_3_False	0.114	-	0.876	0.252	0.056	**0.000***
	SAC_3_True	0.155	0.876	-	0.330	0.083	**0.000***
	SAC_10_False	0.593	0.252	0.330	-	0.444	**0.000***
	SAC_10_True	0.872	0.056	0.083	0.444	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-

A.3 Scaling Training Pool Size

TABLE A.3: Bonferroni-corrected p-values from Student's t-tests comparing algorithms across various metrics, based on training with a pool of $N = 500$ stocks. Statistically significant p-values at the 5% level are highlighted in bold.

Metric	Algorithm	DDPG	SAC	BuyAndHold	EqualWeighted	MinVar	Random
Sharpe	DDPG	-	0.484	**0.000***	**0.001***	**0.002***	**0.000***
	SAC	0.484	-	**0.000***	0.005	0.011	**0.000***
	BuyAndHold	**0.000***	**0.000***	-	0.098	0.056	**0.000***
	EqualWeighted	**0.001***	0.005	0.098	-	0.802	**0.000***
	MinVar	**0.002***	0.011	0.056	0.802	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Sortino	DDPG	-	0.532	**0.000***	**0.002***	0.004	**0.000***
	SAC	0.532	-	**0.000***	0.008	0.017	**0.000***
	BuyAndHold	**0.000***	**0.000***	-	0.122	0.069	**0.000***
	EqualWeighted	**0.002***	0.008	0.122	-	0.790	**0.000***
	MinVar	0.004	0.017	0.069	0.790	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Calmar	DDPG	-	0.697	**0.000***	0.004	0.007	**0.000***
	SAC	0.697	-	**0.000***	0.009	0.015	**0.000***
	BuyAndHold	**0.000***	**0.000***	-	0.134	0.089	**0.000***
	EqualWeighted	0.004	0.009	0.134	-	0.843	**0.000***
	MinVar	0.007	0.015	0.089	0.843	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Annual Return	DDPG	-	0.401	**0.000***	0.010	0.044	**0.000***
	SAC	0.401	-	**0.000***	0.057	0.189	**0.000***
	BuyAndHold	**0.000***	**0.000***	-	0.045	0.010	**0.000***
	EqualWeighted	0.010	0.057	0.045	-	0.569	**0.000***
	MinVar	0.044	0.189	0.010	0.569	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Annual Volatility	DDPG	-	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***
	SAC	**0.000***	-	**0.000***	**0.000***	**0.000***	**0.000***
	BuyAndHold	**0.000***	**0.000***	-	0.752	**0.001***	**0.000***
	EqualWeighted	**0.000***	**0.000***	0.752	-	**0.003***	**0.000***
	MinVar	**0.000***	**0.000***	**0.001***	**0.003***	-	0.046
	Random	**0.000***	**0.000***	**0.000***	**0.000***	0.046	-
Total Rewards	DDPG	-	0.300	**0.000***	**0.002***	0.011	**0.000***
	SAC	0.300	-	**0.000***	0.024	0.088	**0.000***
	BuyAndHold	**0.000***	**0.000***	-	0.048	0.011	**0.000***
	EqualWeighted	**0.002***	0.024	0.048	-	0.586	**0.000***
	MinVar	0.011	0.088	0.011	0.586	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-

A.4 Varying Transaction Costs

TABLE A.4: Bonferroni-corrected p-values from Student's t-tests comparing algorithms across various metrics, based on training with a pool of $N = 100$ stocks and evaluated in an environment with a transaction cost level of 0 basis points. Statistically significant p-values at the 5% level are highlighted in bold.

Metric	Algorithm	SAC	BuyAndHold	EqualWeighted	MinVar	Random
Sharpe	SAC	-	0.009	0.346	0.488	0.897
	BuyAndHold	0.009	-	0.145	0.092	0.038
	EqualWeighted	0.346	0.145	-	0.823	0.499
	MinVar	0.488	0.092	0.823	-	0.643
	Random	0.897	0.038	0.499	0.643	-
Sortino	SAC	-	0.020	0.430	0.603	0.971
	BuyAndHold	0.020	-	0.177	0.111	0.054
	EqualWeighted	0.430	0.177	-	0.811	0.530
	MinVar	0.603	0.111	0.811	-	0.688
	Random	0.971	0.054	0.530	0.688	-
Calmar	SAC	-	0.021	0.428	0.551	0.827
	BuyAndHold	0.021	-	0.188	0.135	0.078
	EqualWeighted	0.428	0.188	-	0.862	0.635
	MinVar	0.551	0.135	0.862	-	0.758
	Random	0.827	0.078	0.635	0.758	-
Annual Return	SAC	-	0.033	0.894	0.626	0.253
	BuyAndHold	0.033	-	0.069	0.018	**0.004***
	EqualWeighted	0.894	0.069	-	0.581	0.256
	MinVar	0.626	0.018	0.581	-	0.542
	Random	0.253	**0.004***	0.256	0.542	-
Annual Volatility	SAC	-	**0.000***	**0.000***	**0.000***	**0.000***
	BuyAndHold	**0.000***	-	0.753	**0.001***	**0.000***
	EqualWeighted	**0.000***	0.753	-	**0.003***	**0.000***
	MinVar	**0.000***	**0.001***	**0.003***	-	0.087
	Random	**0.000***	**0.000***	**0.000***	0.087	-
Total Rewards	SAC	-	0.015	0.627	0.927	0.420
	BuyAndHold	0.015	-	0.073	0.020	**0.004***
	EqualWeighted	0.627	0.073	-	0.598	0.247
	MinVar	0.927	0.020	0.598	-	0.511
	Random	0.420	**0.004***	0.247	0.511	-

TABLE A.5: Bonferroni-corrected p-values from Student's t-tests comparing algorithms across various metrics, based on training with a pool of $N = 100$ stocks and evaluated in an environment with a transaction cost level of 15 basis points. Statistically significant p-values at the 5% level are highlighted in bold.

Metric	Algorithm	SAC	BuyAndHold	EqualWeighted	MinVar	Random
Sharpe	SAC	-	0.013	0.996	0.702	**0.000***
	BuyAndHold	0.013	-	0.041	0.018	**0.000***
	EqualWeighted	0.996	0.041	-	0.759	**0.000***
	MinVar	0.702	0.018	0.759	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	-
Sortino	SAC	-	0.019	0.997	0.689	**0.000***
	BuyAndHold	0.019	-	0.053	0.023	**0.000***
	EqualWeighted	0.997	0.053	-	0.748	**0.000***
	MinVar	0.689	0.023	0.748	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	-
Calmar	SAC	-	0.047	0.791	0.568	**0.000***
	BuyAndHold	0.047	-	0.063	0.035	**0.000***
	EqualWeighted	0.791	0.063	-	0.806	**0.000***
	MinVar	0.568	0.035	0.806	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	-
Annual Return	SAC	-	**0.000***	0.304	0.765	**0.000***
	BuyAndHold	**0.000***	-	0.018	**0.003***	**0.000***
	EqualWeighted	0.304	0.018	-	0.545	**0.000***
	MinVar	0.765	**0.003***	0.545	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	-
Annual Volatility	SAC	-	**0.000***	**0.000***	**0.000***	**0.001***
	BuyAndHold	**0.000***	-	0.750	**0.001***	**0.000***
	EqualWeighted	**0.000***	0.750	-	**0.003***	**0.000***
	MinVar	**0.000***	**0.001***	**0.003***	-	0.038
	Random	**0.001***	**0.000***	**0.000***	0.038	-
Total Rewards	SAC	-	**0.000***	0.336	0.796	**0.000***
	BuyAndHold	**0.000***	-	0.018	**0.003***	**0.000***
	EqualWeighted	0.336	0.018	-	0.561	**0.000***
	MinVar	0.796	**0.003***	0.561	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	-

TABLE A.6: Bonferroni-corrected p-values from Student's t-tests comparing algorithms across various metrics, based on training with a pool of $N = 100$ stocks and evaluated in an environment with a transaction cost level of 25 basis points. Statistically significant p-values at the 5% level are highlighted in bold.

Metric	Algorithm	SAC	BuyAndHold	EqualWeighted	MinVar	Random
Sharpe	SAC	-	**0.003***	0.988	0.667	**0.000***
	BuyAndHold	**0.003***	-	0.015	**0.005***	**0.000***
	EqualWeighted	0.988	0.015	-	0.717	**0.000***
	MinVar	0.667	**0.005***	0.717	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	-
Sortino	SAC	-	**0.005***	0.988	0.654	**0.000***
	BuyAndHold	**0.005***	-	0.020	0.007	**0.000***
	EqualWeighted	0.988	0.020	-	0.706	**0.000***
	MinVar	0.654	0.007	0.706	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	-
Calmar	SAC	-	0.016	0.784	0.523	**0.000***
	BuyAndHold	0.016	-	0.026	0.012	**0.000***
	EqualWeighted	0.784	0.026	-	0.769	**0.000***
	MinVar	0.523	0.012	0.769	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	-
Annual Return	SAC	-	**0.000***	0.310	0.810	**0.000***
	BuyAndHold	**0.000***	-	0.006	**0.001***	**0.000***
	EqualWeighted	0.310	0.006	-	0.521	**0.000***
	MinVar	0.810	**0.001***	0.521	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	-
Annual Volatility	SAC	-	**0.000***	**0.000***	**0.000***	**0.000***
	BuyAndHold	**0.000***	-	0.748	**0.001***	**0.000***
	EqualWeighted	**0.000***	0.748	-	**0.003***	**0.000***
	MinVar	**0.000***	**0.001***	**0.003***	-	0.056
	Random	**0.000***	**0.000***	**0.000***	0.056	-
Total Rewards	SAC	-	**0.000***	0.343	0.844	**0.000***
	BuyAndHold	**0.000***	-	0.006	**0.001***	**0.000***
	EqualWeighted	0.343	0.006	-	0.537	**0.000***
	MinVar	0.844	**0.001***	0.537	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	-

A.5 Contextual Market Information

TABLE A.7: Bonferroni-corrected p-values from Student's t-tests comparing different SAC architectures across various metrics, based on training with a pool of $N = 100$ stocks. Statistically significant p-values at the 5% level are highlighted in bold.

Metric	Algorithm	SAC	SAC_context	BuyAndHold	EqualWeighted	MinVar	Random
Sharpe	SAC	-	0.699	**0.000***	0.072	0.127	**0.000***
	SAC_context	0.699	-	**0.000***	0.034	0.065	**0.000***
	BuyAndHold	**0.000***	**0.000***	-	0.098	0.056	**0.000***
	EqualWeighted	0.072	0.034	0.098	-	0.802	**0.000***
	MinVar	0.127	0.065	0.056	0.802	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Sortino	SAC	-	0.706	**0.001***	0.088	0.157	**0.000***
	SAC_context	0.706	-	**0.000***	0.044	0.085	**0.000***
	BuyAndHold	**0.001***	**0.000***	-	0.122	0.069	**0.000***
	EqualWeighted	0.088	0.044	0.122	-	0.790	**0.000***
	MinVar	0.157	0.085	0.069	0.790	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Calmar	SAC	-	0.606	**0.001***	0.130	0.194	**0.000***
	SAC_context	0.606	-	**0.000***	0.057	0.090	**0.000***
	BuyAndHold	**0.001***	**0.000***	-	0.134	0.089	**0.000***
	EqualWeighted	0.130	0.057	0.134	-	0.843	**0.000***
	MinVar	0.194	0.090	0.089	0.843	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Annual Return	SAC	-	0.656	**0.001***	0.219	0.544	**0.000***
	SAC_context	0.656	-	**0.000***	0.108	0.322	**0.000***
	BuyAndHold	**0.001***	**0.000***	-	0.045	0.010	**0.000***
	EqualWeighted	0.219	0.108	0.045	-	0.569	**0.000***
	MinVar	0.544	0.322	0.010	0.569	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-
Annual Volatility	SAC	-	0.454	**0.000***	**0.000***	**0.000***	**0.000***
	SAC_context	0.454	-	**0.000***	**0.000***	**0.000***	**0.000***
	BuyAndHold	**0.000***	**0.000***	-	0.752	**0.001***	**0.000***
	EqualWeighted	**0.000***	**0.000***	0.752	-	**0.003***	**0.000***
	MinVar	**0.000***	**0.000***	**0.001***	**0.003***	-	0.046
	Random	**0.000***	**0.000***	**0.000***	**0.000***	0.046	-
Total Rewards	SAC	-	0.702	**0.000***	0.136	0.365	**0.000***
	SAC_context	0.702	-	**0.000***	0.068	0.215	**0.000***
	BuyAndHold	**0.000***	**0.000***	-	0.048	0.011	**0.000***
	EqualWeighted	0.136	0.068	0.048	-	0.586	**0.000***
	MinVar	0.365	0.215	0.011	0.586	-	**0.000***
	Random	**0.000***	**0.000***	**0.000***	**0.000***	**0.000***	-

A.6 Lookback Window

TABLE A.8: Bonferroni-corrected p-values from Student's t-tests comparing different lookback window step sizes of SAC across various metrics, based on training with a pool of $N = 100$ stocks. Statistically significant p-values at the 5% level are highlighted in bold.

Metric	Algorithm	SAC	SAC_1	SAC_5	SAC_21
Sharpe	SAC	-	0.842	0.570	0.991
	SAC_1	0.842	-	0.438	0.833
	SAC_5	0.570	0.438	-	0.578
	SAC_21	0.991	0.833	0.578	-
Sortino	SAC	-	0.836	0.560	0.966
	SAC_1	0.836	-	0.426	0.803
	SAC_5	0.560	0.426	-	0.590
	SAC_21	0.966	0.803	0.590	-
Calmar	SAC	-	0.840	0.463	0.715
	SAC_1	0.840	-	0.346	0.869
	SAC_5	0.463	0.346	-	0.268
	SAC_21	0.715	0.869	0.268	-
Annual Return	SAC	-	0.679	0.701	0.886
	SAC_1	0.679	-	0.417	0.578
	SAC_5	0.701	0.417	-	0.815
	SAC_21	0.886	0.578	0.815	-
Annual Volatility	SAC	-	**0.003***	0.016	0.360
	SAC_1	**0.003***	-	0.568	0.041
	SAC_5	0.016	0.568	-	0.145
	SAC_21	0.360	0.041	0.145	-
Total Rewards	SAC	-	0.737	0.668	0.907
	SAC_1	0.737	-	0.435	0.650
	SAC_5	0.668	0.435	-	0.758
	SAC_21	0.907	0.650	0.758	-

Appendix B

Supplementary Plots

B.1 PPO Investigation

In Figure B.1, we include learning curves for PPO with varying levels of the entropy coefficient. The levels of the entropy coefficient for PPO are chosen based on commonly accepted values, typically ranging between 0 and 0.01. A higher entropy coefficient might help PPO generalize better across different portfolios, aiding in converging to a more general policy. On the contrary, we see lower rewards with higher values for the entropy coefficient. Apparently, PPO is not able to find a stable policy under higher entropy, and the decrease in rewards could be due to an increase in transaction costs. Entropy seems to hinder PPO's learning more than it generalizes it.

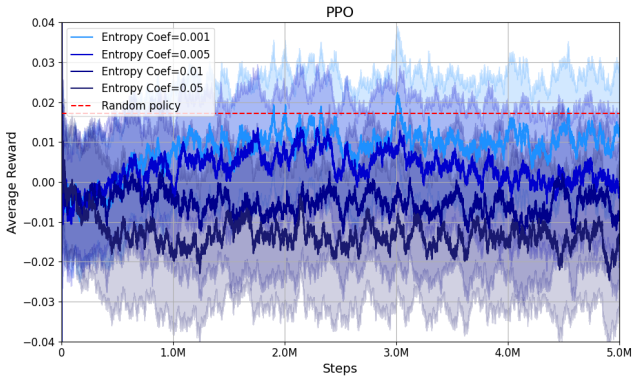


FIGURE B.1: Learning curves for PPO for different exploration settings, trained on $N = 100$ stocks, with three different seeds each ($s \in [12, 42, 1234]$). Specifically, we have varied the entropy coefficient between 0.001, 0.005, 0.01 and 0.05. Darker colors are higher values for the entropy coefficient. We observe that higher values for the entropy lead to lower rewards, possibly because of higher transaction costs resulting from more stochastic behavior.

B.2 Permutation Equivariance

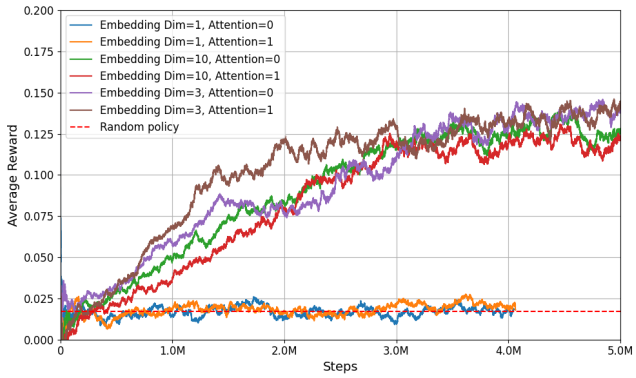


FIGURE B.2: Learning curves for SAC for different network architectures aimed at improving permutation equivariance, trained on $N = 100$ stocks, on one seed $s = 12$. Specifically, we vary the embedding dimension and the use of an attention layer. We observe that attention improves learning speed and that an embedding dimension of one appears too small for stable learning.

B.3 Scaling Hidden Layer Dimension

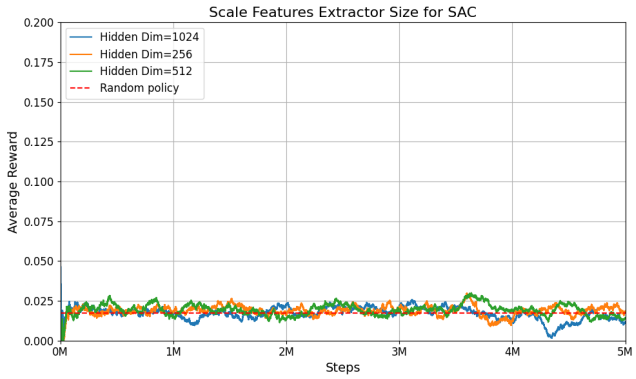


FIGURE B.3: Learning curves for SAC for different hidden layer dimensions in the features extractor, trained on $N = 1469$ stocks, on one seed $s = 12$. We observe that the dimensionality of the hidden layers does not help in learning a stable policy.

Bibliography

Afsar, M Mehdi, Trafford Crump, and Behrouz Far (2022). "Reinforcement learning based recommender systems: A survey". In: *ACM Computing Surveys* 55.7, pp. 1–38.

Ahmed, Zafarali et al. (2019). "Understanding the impact of entropy on policy optimization". In: *International conference on machine learning*. PMLR, pp. 151–160.

Appel, Gerald (2005). *Technical Analysis: Power Tools for Active Investors*. The book discusses various technical analysis tools including the Moving Average Convergence Divergence (MACD). Financial Times/Prentice Hall.

Arulkumaran, Kai et al. (2017). "Deep reinforcement learning: A brief survey". In: *IEEE Signal Processing Magazine* 34.6, pp. 26–38.

Back, Kerry (2010). *Asset pricing and portfolio choice theory*. Oxford University Press.

Bellman, Richard E. (1957). "A Markovian decision process". In: *Journal of Mathematics and Mechanics* 6.5, pp. 679–684.

Benhamou, Eric (2023). "Can Deep Reinforcement Learning solve the portfolio allocation problem?" PhD thesis. Université Paris sciences et lettres.

Benhamou, Eric et al. (2021). "Deep reinforcement learning (drl) for portfolio allocation". In: *Machine Learning and Knowledge Discovery in Databases. Applied Data Science and Demo Track: European Conference, ECML PKDD 2020, Ghent, Belgium, September 14–18, 2020, Proceedings, Part V*. Springer, pp. 527–531.

Berner, Christopher et al. (2019). "Dota 2 with large scale deep reinforcement learning". In: *arXiv preprint arXiv:1912.06680*.

Biewald, Lukas (2020). *Experiment Tracking with Weights and Biases*. Software available from wandb.com. URL: https://www.wandb.com/.

Bollinger, John (1992). "Using Bollinger Bands". In: *Stocks & Commodities* 10.2. This article by John Bollinger discusses the use and calculation of Bollinger Bands., pp. 47–51.

Carta, Salvatore et al. (2021). "A multi-layer and multi-ensemble stock trader using deep learning and deep reinforcement learning". In: *Applied Intelligence* 51, pp. 889–905.

Chakraborty, Souradeep (2019). "Capturing financial markets to apply deep reinforcement learning". In: *arXiv preprint arXiv:1907.04373*.

Cho, Kyunghyun (2014). "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078*.

Clarke, Roger, Harindra De Silva, and Steven Thorley (2011). "Minimum-variance portfolio composition". In: *Journal of Portfolio Management* 37.2, p. 31.

Cobbe, Karl et al. (2019). "Quantifying generalization in reinforcement learning". In: *International conference on machine learning*. PMLR, pp. 1282–1289.

Corchado, J, Colin Fyfe, and Brian Lees (1998). "Unsupervised learning for financial forecasting". In: *Proceedings of the IEEE/IAFE/INFORMS 1998 Conference on Computational Intelligence for Financial Engineering (CIFEr)(Cat. No. 98TH8367)*. IEEE, pp. 259–263.

Davis, Mark HA and Andrew R Norman (1990). "Portfolio selection with transaction costs". In: *Mathematics of operations research* 15.4, pp. 676–713.

Dixon, Matthew F, Igor Halperin, and Paul Bilokon (2020). *Machine learning in finance*. Vol. 1170. Springer.

Du, Xin, Jinjian Zhai, and Koupin Lv (2016). "Algorithm trading using q-learning and recurrent reinforcement learning". In: *positions* 1.1, pp. 1–7.

Duchi, John, Elad Hazan, and Yoram Singer (2011). "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research* 12, pp. 2121–2159.

Fahad Mon, Bisni et al. (2023). "Reinforcement learning in education: A literature review". In: *Informatics*. Vol. 10. 3. MDPI, p. 74.

Fujimoto, Scott, Herke Hoof, and David Meger (2018). "Addressing function approximation error in actor-critic methods". In: *International conference on machine learning*. PMLR, pp. 1587–1596.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. MIT Press.

Gu, Shixiang, Timothy P Lillicrap, and Sergey Levine (2017). "Deep reinforcement learning for robotic manipulation with asynchronous policy updates". In: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 3389–3396. URL: https://ieeexplore.ieee.org/document/7989230.

Guan, Mao and Xiao-Yang Liu (2022). "Explainable deep reinforcement learning for portfolio management: an empirical approach". In: *Proceedings of the Second ACM International Conference on AI in Finance*. ICAIF '21. Virtual Event: Association for Computing Machinery.

Haarnoja, Tuomas et al. (2018). "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". In: *Proceedings of the 35th International Conference on Machine Learning (ICML)*. Vol. 80. PMLR, pp. 1861–1870.

Hambly, Ben, Renyuan Xu, and Huining Yang (2023). "Recent advances in reinforcement learning in finance". In: *Mathematical Finance* 33.3, pp. 437–503.

Henderson, Peter et al. (2018). "Deep reinforcement learning that matters". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 1.

Hochreiter, S (1997). "Long Short-term Memory". In: *Neural Computation MIT-Press*.

Hornik, Kurt (1991). "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4.2, pp. 251–257.

Hospedales, Timothy et al. (2021). "Meta-learning in neural networks: A survey". In: *IEEE transactions on pattern analysis and machine intelligence* 44.9, pp. 5149–5169.

Huang, Chien Yi (2018). "Financial trading as a game: A deep reinforcement learning approach". In: *arXiv preprint arXiv:1807.02787*.

Huang, Shengyi et al. (2022). "The 37 Implementation Details of Proximal Policy Optimization". In: *ICLR Blog Track*. https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/. URL: https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.

Huisman, Mike, Jan N Van Rijn, and Aske Plaat (2021). "A survey of deep meta-learning". In: *Artificial Intelligence Review* 54.6, pp. 4483–4541.

Jama, Fuaad (2023). "Deep reinforcement learning approach to portfolio management".

Jang, Junkyu and NohYoon Seong (2023). "Deep reinforcement learning for stock portfolio optimization by connecting with modern portfolio theory". In: *Expert Systems with Applications* 218, p. 119556.

Jiang, Zhengyao, Dixing Xu, and Jinjun Liang (2017). "A deep reinforcement learning framework for the financial portfolio management problem". In: *arXiv preprint arXiv:1706.10059*.

Jin, Yanliang et al. (2021). "Deep deterministic policy gradient algorithm based on convolutional block attention for autonomous driving". In: *Symmetry* 13.6, p. 1061.

Kingma, Diederik P and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980*.

Kipf, Thomas N. and Max Welling (2017). "Semi-Supervised Classification with Graph Convolutional Networks". In: *International Conference on Learning Representations (ICLR)*. URL: https://arxiv.org/abs/1609.02907.

Kiran, B Ravi et al. (2021). "Deep reinforcement learning for autonomous driving: A survey". In: *IEEE Transactions on Intelligent Transportation Systems* 23.6, pp. 4909–4926.

Kober, Jens, J. Andrew Bagnell, and Jan Peters (2013). "Reinforcement learning in robotics: A survey". In: *The International Journal of Robotics Research* 32.11, pp. 1238–1274.

Konda, Vijay and John Tsitsiklis (1999). "Actor-critic algorithms". In: *Advances in neural information processing systems* 12.

Kousen, Jasper et al. (2023). *Navigating the World of Quantitative Investing: A Concise Guide*. Tech. rep. Company whitepaper, unpublished.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *Nature* 521.7553, pp. 436–444.

LeCun, Yann et al. (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

Lee, Juho et al. (2019). *Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks*. arXiv: 1810.00825 [cs.LG]. URL: https://arxiv.org/abs/1810.00825.

Levine, Sergey et al. (2020). "Offline reinforcement learning: Tutorial, review, and perspectives on open problems". In: *arXiv preprint arXiv:2005.01643*.

Li, Haifeng and Mo Hai (2024). "Deep Reinforcement Learning Model for Stock Portfolio Management Based on Data Fusion". In: *Neural Processing Letters* 56.2, p. 108.

Li, Yuanzheng et al. (2023). "Deep reinforcement learning for smart grid operations: Algorithms, applications, and prospects". In: *Proceedings of the IEEE* 111.9, pp. 1055–1096.

Li, Yuxi (2017). "Deep reinforcement learning: An overview". In: *arXiv preprint arXiv:1701.07274*.

Lillicrap, Timothy P et al. (2015). "Continuous Control with Deep Reinforcement Learning". In: *arXiv preprint arXiv:1509.02971*.

Liu, Xiao-Yang et al. (2018). "Practical deep reinforcement learning approach for stock trading". In: *arXiv preprint arXiv:1811.07522*.

Liu, Xiao-Yang et al. (2020). "FinRL: A deep reinforcement learning library for automated stock trading in quantitative finance". In: *arXiv preprint arXiv:2011.09607*.

Lotfinejad, Mehdi (2023). *6 Most Common Deep Learning Applications*. https://www.dataquest.io/blog/6-most-common-deep-learning-applications/. Accessed: 2024-11-21.

Lu, Chung I (2023). "Evaluation of Deep Reinforcement Learning Algorithms for Portfolio Optimisation". In: *arXiv preprint arXiv:2307.07694*.

Markowitz, Harry (1952). "Portfolio Selection". In: *The Journal of Finance* 7.1, pp. 77–91. ISSN: 00221082, 15406261. URL: http://www.jstor.org/stable/2975974 (visited on 09/09/2024).

Merton, Robert C. (1969). "Lifetime portfolio selection under uncertainty: The continuous-time case". In: *The Review of Economics and Statistics* 51.3, pp. 247–257.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *nature* 518.7540, pp. 529–533.

Mnih, Volodymyr et al. (2016). "Asynchronous Methods for Deep Reinforcement Learning". In: *Proceedings of the 33rd International Conference on Machine Learning (ICML)*. Vol. 48. PMLR, pp. 1928–1937.

Moerland, Thomas M et al. (2023). "Model-based reinforcement learning: A survey". In: *Foundations and Trends® in Machine Learning* 16.1, pp. 1–118.

Moravčík, Matej et al. (2017). "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker". In: *Science* 356.6337, pp. 508–513.

Nair, Vinod and Geoffrey E. Hinton (2010). "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pp. 807–814.

Ng, Andrew Y, Daishi Harada, and Stuart Russell (1999). "Policy invariance under reward transformations: Theory and application to reward shaping". In: *Icml*. Vol. 99, pp. 278–287.

Osterrieder, Joerg and Chat GPT (2023). "A primer on deep reinforcement learning for finance". In: *Available at SSRN 4316650*.

Packer, Charles et al. (2018). "Assessing generalization in deep reinforcement learning". In: *arXiv preprint arXiv:1810.12282*.

Park, Hyungjun, Min Kyu Sim, and Dong Gu Choi (2020). "An intelligent financial portfolio trading strategy using deep Q-learning". In: *Expert Systems with Applications* 158, p. 113573.

Patro, S (2015). "Normalization: A preprocessing stage". In: *arXiv preprint arXiv:1503.06462*.

Pendharkar, Parag C and Patrick Cusatis (2018). "Trading financial indices with reinforcement learning agents". In: *Expert Systems with Applications* 103, pp. 1–13.

Plaat, Aske (2022). *Deep reinforcement learning*. Vol. 10. Springer.

Polyak, Boris T (1964). "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* 4.5, pp. 1–17.

Raffin, Antonin et al. (2021). "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268, pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

Robbins, Herbert and Sutton Monro (1951). "A Stochastic Approximation Method". In: *The Annals of Mathematical Statistics* 22.3, pp. 400–407. DOI: 10.1214/aoms/1177729586. URL: https://projecteuclid.org/euclid.aoms/1177729586.

Ruder, Sebastian (2016). "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747*.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1986). "Learning representations by back-propagating errors". In: *Nature* 323.6088, pp. 533–536.

Rummery, Gavin A and Mahesan Niranjan (1994). *On-line Q-learning using connectionist systems*. Vol. 37. University of Cambridge, Department of Engineering Cambridge, UK.

Rundo, Francesco et al. (2019). "Machine learning for quantitative finance applications: A survey". In: *Applied Sciences* 9.24, p. 5574.

Sato, Yoshiharu (2019). "Model-free reinforcement learning for financial portfolios: a brief survey". In: *arXiv preprint arXiv:1904.04973*.

Scarselli, Franco et al. (2008). "The graph neural network model". In: *IEEE transactions on neural networks* 20.1, pp. 61–80.

Schulman, John et al. (2015). "Trust Region Policy Optimization". In: *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. PMLR, pp. 1889–1897.

Schulman, John et al. (2017). "Proximal Policy Optimization Algorithms". In: *arXiv preprint arXiv:1707.06347*.

Shapley, Lloyd S (1953). "Stochastic games". In: *Proceedings of the national academy of sciences* 39.10, pp. 1095–1100.

Silver, David et al. (2018). "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419, pp. 1140–1144.

Sood, Srijan et al. (2023). "Deep Reinforcement Learning for Optimal Portfolio Allocation: A Comparative Study with Mean-Variance Optimization". In: *FinPlan 2023*, p. 21.

S&P Dow Jones Indices (1995). *S&P Composite 1500®*. S&P Global. URL: https://www.spglobal.com/spdji/en/indices/equity/sp-composite-1500/ (visited on 04/06/2025).

Sutton, Richard S (2018). "Reinforcement learning: An introduction". In: *A Bradford Book*.

Sutton, Richard S et al. (1999). "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Advances in Neural Information Processing Systems*. Ed. by S. Solla, T. Leen, and K. Müller. Vol. 12. MIT Press.

Tang, Chen et al. (2024). "Deep reinforcement learning for robotics: A survey of real-world successes". In: *Annual Review of Control, Robotics, and Autonomous Systems* 8.

Tang, Yujin and David Ha (2021). "The sensory neuron as a transformer: Permutation-invariant neural networks for reinforcement learning". In: *Advances in Neural Information Processing Systems* 34, pp. 22574–22587.

Tieleman, Tijmen and Geoffrey Hinton (2012). "Lecture 6.5—RMSProp: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural Networks for Machine Learning* 4.2.

Towers, Mark et al. (2024). "Gymnasium: A standard interface for reinforcement learning environments". In: *arXiv preprint arXiv:2407.17032*.

Uhlenbeck, George E and Leonard S Ornstein (1930). "On the theory of the Brownian motion". In: *Physical review* 36.5, p. 823.

Van Hasselt, Hado, Arthur Guez, and David Silver (2016). "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1.

Vaswani, A (2017). "Attention is all you need". In: *Advances in Neural Information Processing Systems*.

Vinyals, Oriol et al. (2019). "Grandmaster level in StarCraft II using multi-agent reinforcement learning". In: *nature* 575.7782, pp. 350–354.

Wang, Haoran (2019). "Large scale continuous-time mean-variance portfolio allocation via reinforcement learning". In: *arXiv preprint arXiv:1907.11718*.

Watkins, Christopher JCH and Peter Dayan (1992). "Q-learning". In: *Machine learning* 8, pp. 279–292.

Wen, Wen, Yuyu Yuan, and Jincui Yang (2021). "Reinforcement learning for options trading". In: *Applied Sciences* 11.23, p. 11208.

Wilder, J. Welles (1978). *New Concepts in Technical Trading Systems*. This book introduces several key concepts in technical analysis, including the Relative Strength Index (RSI). Trend Research.

Williams, Ronald J (1992). "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8, pp. 229–256.

Wu, Yuhuai et al. (2017). "Scalable Trust-Region Method for Deep Reinforcement Learning using Kronecker-Factored Approximation". In: *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 5285–5294.

Yan, Yimo et al. (2022). "Reinforcement learning for logistics and supply chain management: Methodologies, state of the art, and future opportunities". In: *Transportation Research Part E: Logistics and Transportation Review* 162, p. 102712.

Yang, Hongyang et al. (2020). "Deep reinforcement learning for automated stock trading: An ensemble strategy". In: *Proceedings of the first ACM international conference on AI in finance*, pp. 1–8.

Yu, Chao et al. (2021). "Reinforcement learning in healthcare: A survey". In: *ACM Computing Surveys (CSUR)* 55.1, pp. 1–36.

Zejnullahu, Frensi, Maurice Moser, and Joerg Osterrieder (June 2022). *Applications of Reinforcement Learning in Finance – Trading with a Double Deep Q-Network*. English. WorkingPaper. DOI: 10.48550/arXiv.2206.14267.

Zhang, Zidong, Dongxia Zhang, and Robert C Qiu (2019). "Deep reinforcement learning for power system applications: An overview". In: *CSEE Journal of Power and Energy Systems* 6.1, pp. 213–225.

Zhang, Zihao, Stefan Zohren, and Stephen Roberts (2019). "Deep reinforcement learning for trading". In: *arXiv preprint arXiv:1911.10107*.

Zhao, Wenshuai, Jorge Peña Queralta, and Tomi Westerlund (2020). "Sim-to-real transfer in deep reinforcement learning for robotics: a survey". In: *2020 IEEE symposium series on computational intelligence (SSCI)*. IEEE, pp. 737–744.

Zhu, Zhuangdi et al. (2023). "Transfer learning in deep reinforcement learning: A survey". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.11, pp. 13344–13362.