UNIVERSITÀ CATTOLICA DEL SACRO CUORE

Campus of Milan

Faculty of Economics

Master program in Data Analytics for Business

UNIVERSITÀ
CATTOLICA
del Sacro Cuore

Deep Reinforcement Learning for Autonomous Portfolio Management: A Policy Gradient Approach

Supervisor:

Professor Andrea Pozzi

Dissertation by:

Jacopo Signò

Id Number: 5111043

Academic Year 2023/2024

Contents

1 Introduction

The goal of this thesis work is to delve into exploring the theoretical foundations of Reinforcement Learning (RL); a prominent paradigm in the realm of artificial intelligence and machine learning. By digging into the fundamental principles governing RL, this research aims to provide a comprehensive understanding of its mechanisms, limitations, and potential advancements, with a special focus on its applications in the field of finance, more specifically to *portfolio optimization* problem.

Reinforcement learning is a branch of machine learning concerned with learning how to make optimal sequential decisions through interaction with an environment to achieve a specific goal. RL algorithms learn by trial and error, receiving feedback in the form of rewards or penalties based on their actions. This feedback enables the RL agent to learn which actions lead to favorable outcomes and which do not, ultimately optimizing its decision-making process over time.

RL has found applications across a wide range of domains, including robotics, gaming, healthcare, autonomous vehicles and finance.

1.1 Problem Definition

The primary objective of this thesis is to investigate the application of Deep Reinforcement Learning algorithms, to the problem of asset portfolio management. Specifically, we aim to develop and evaluate a DDPG-based agent capable of autonomously managing an investment portfolio in a financial market environment.

The management of investment portfolios is a complex and challenging task, often requiring sophisticated decision-making under conditions of uncertainty, dynamic market conditions, and diverse investment objectives. Traditional portfolio management approaches rely heavily on human expertise, economic models, and historical data analysis to construct and rebalance portfolios. However, these methods are often limited in their ability to adapt to rapidly changing market dynamics and may fail to exploit subtle patterns or emerging trends.

In recent years, the intersection of artificial intelligence (AI) and finance has garnered significant attention from researchers and practitioners. The application of AI techniques, particularly reinforcement learning (RL), to asset portfolio management has emerged as a promising area of study, offering the potential to enhance investment decision-making processes and achieve superior returns in financial markets.

1.2 Structure of the Thesis

The thesis is organized as follows:

- **Chapter 2: Portfolio optimization theory:** provide an overview of portfolio optimization theory, introducing traditional techniques and their shortcomings.
- **Chapter 3: Theoretical Background**: Provides an in-depth exploration of reinforcement learning and the DDPG algorithm, laying the theoretical groundwork for our research.
- **Chapter 4: Methodology and Experimental Evaluation:** Details the design and implementation of the DDPG-based portfolio management agent, including data preparation, model architecture, training process, testings process and results evaluation.
- **Chapter 5: Conclusion:** Summarizes the key findings of the thesis, reflects on the contributions and limitations of the research, and outlines potential directions for future work.

Through this structured approach, we aim to provide a comprehensive analysis of the application of DDPG to asset portfolio management, contributing to the growing body of knowledge at the intersection of AI and finance.

1.3 Literature Review

Due to the vast amount of literature on both traditional portfolio optimization methods and Reinforcement Learning, it would be impractical to cite all relevant works. However, a brief overview of key contributions in each area is provided.

Traditional asset management methods, pioneered by Markowitz with his mean-variance optimization (MVO) framework [25], have evolved to encompass various extensions, such as minimum variance (Chopra and Ziemba 1993)[6], maximum diversification (Choueifaty and Coignard 2008; Choueifaty, Froidure, and Reynier 2012) [7] [8], and risk parity (Maillard, Roncalli, and Teiletche 2010; Roncalli and Weisang 2016)[24][31].

On the Reinforcement Learning side fundamental is the book "Reinforcement Learning: An Introduction" (Sutton and Barto 2018) [37]. The field of deep reinforcement learning (DRL) is growing every day at an unprecedented pace. In recent years, in terms of breakthroughs of deep reinforcement learning, one can cite the work around Atari games from raw pixel inputs (Mnih et al. 2013; 2015) [27], Go (Silver et al. 2016; 2017) [33][34], StarCraft II (Vinyals et al. 2019) [39] , learning advanced locomotion and manipulation skills from raw sensory inputs (Levine et al. 2015; 2016) (Schulman et al. 2015a; 2015b; 2017; Lillicrap et al. 2015)[19] [32], [21], autonomous driving (Wang, Jia, and Weng 2018) [40] and robot learning (Gu et al. 2017) [14].

Within the domain of portfolio allocation, DRL has gained traction, initially with applications to cryptocurrency markets and Chinese financial markets (Jiang and Liang 2016; Zhengyao et al. 2017; Liang et al. 2018; Yu et al. 2019; Wang and Zhou 2019; Saltiel et al. 2020; Benhamou et al. 2020b; 2020a; 2020c) [16][45][42]. Its applicability is now expanding to encompass a wider range of assets (Kolm and Ritter 2019; Liu et al. 2020; Ye et al. 2020; Li et al. 2019; Xiong et al. 2019) [18][22][41]. Additionally, DRL has been applied to other financial problems, including direct trading strategies (Deng et al. 2016; Zhang, Zohren, and Roberts 2019; Huang 2018; Theate and Ernst 2020; Chakraborty 2019; Nan, Perumal, and Zaiane 2020; Wu et al. 2020) [10][43][5], multi-agent trading strategies (Bao and yang Liu 2019) [2], and optimal execution (Ning, Lin, and Jaimungal 2018) [28].
Chapter 2.3 mentions another set of research articles that are more closely related to the work done in this dissertation.
While these papers focus on the intersection of traditional asset management and DRL, we acknowledge that DRL has been applied to various other real-world problem domains beyond portfolio allocation.

2 Portfolio Optimization

Portfolio optimization is a formal mathematical approach to making investment decisions across a collection of financial instruments or assets. [26]

Before we start talking about what methods are used in this area, and what has been the evolution of these methods over the years, let's start by defining the fundamental concepts behind this subject, namely, what is meant by **portfolio** and **portfolio management**.

A portfolio is a collection of multiple financial assets (such as stocks, bonds, cryptocurrency, ecc...) that an investor owns. The portfolio is characterized by its:

- **Constituents**: M assets of which it consists;
- **Portfolio vector** w_t: where its i-th component represents the ratio of the total budget invested to the i-th asset at time t, such that:

$$\boldsymbol{w}_t = \left[\; w_{1,t}, \quad w_{2,t}, \quad \ldots, \quad w_{M,t} \; \right]^T \in \mathbb{R}^M \text{ and } \sum_{i=1}^{M} w_{i,t} = 1$$

Portfolio construction is the process of selecting the M assets and deciding how much of the total available amount to invest in each asset (i.e. decide w_t). This process typically involves considering the following factors:

- **Risk tolerance**: An investor's risk tolerance is their ability and willingness to accept risk. Investors with a low risk tolerance will typically invest in assets with lower volatility and lower returns, while investors with a high risk tolerance will typically invest in assets with higher returns but higher volatility.
- **Investment goals:** : The investor's goals are the reasons why they are investing in the first place. These goals could include saving for retirement, buying a house, or sending their children to prestigious universities.
- **Assets selection:** : This is important because different asset classes have different risk and return characteristics. For example, stocks are typically more risky than bonds, but they also have the potential to generate higher returns.

Portfolio management is the ongoing process of monitoring and managing a portfolio. This includes tasks such as:

- **Tracking performance:** Investors should regularly track the performance of their portfolios to make sure they are meeting their investment goals.

- **Rebalancing:** Rebalancing is the process of adjusting an investor's portfolio to maintain the desired asset allocation (by periodically changing the weights of the portfolio vector). This is important because asset prices can fluctuate over time, which can cause the portfolio's asset allocation to drift away from the desired target.
- **Reviewing asset allocation:** Investors should regularly track the performance of their portfolios to make sure they are meeting their investment goals.

Throughout this thesis, the terms , Portfolio Management and Asset allocation are used interchangeably to refer to the process of selecting the weights to be assigned to each asset in the portfolio to achieve the best possible expected return.

Portfolio Optimization aims to address the allocation problem in a systematic way, where an objective function reflecting the investor's preferences is constructed and optimized with respect to the portfolio vector.

2.1 Traditional methods

Portofolio optimization was mathematically formalized for the first time by Harry Markowitz in 1952 giving rise to the concept of Modern Portfolio Theory (MPT). MPT and Markowitz's original model utilize quadratic programming techniques and make assumptions about the distribution of expected returns and asset volatility, typically employing linear estimators. Subsequently, again Markowitz in 1961 and William Sharpe in 1964 independently introduced the Capital Asset Pricing Model (CAPM), extending the scope of Markowitz Model. Over time various nonlinear regression models, inspired by Signal Processing and Machine Learning have been used since then in order to solve the asset allocation problem.

2.1.1 Modern Portfolio Theory

A revolutionizing result in portfolio optimization was introduced in 1952 by H. Markowitz in the Journal of Finance [25]. Markowitz's innovative approach was rooted in the fundamental principle of diversification. He argued that by spreading investments across multiple assets instead of focusing on a single security, the inherent risk of the portfolio could be significantly mitigated. This concept, known as the "diversification benefit," formed the basis of his methodology, which also incorporated the use of expected return (mean) and standard deviation as metrics for performance and risk, respectively. Consequently, this approach came to be known as mean-variance optimization. Under certain specified assumptions, this method allows for the construction of a set of non-dominated portfolios, indicating portfolios that represent optimal combinations of risk and return, implying that no other portfolio within the set offers a superior risk-return trade-off. The results proposed by Markowitz led to the creation of a new era in portfolio theory, named Modern Portfolio Theory (MPT). Again,

At the heart of MPT lies the concept of diversification, which serves as a cornerstone for constructing efficient portfolios. Diversification is employed to maximize expected returns while maintaining a specific level of risk, or alternatively, to minimize volatility for a given level of return.

Before looking at the mathematical framework of the Markowitz portfolio, it is necessary to make clear the assumption underlying the model, to acknowledge its limitations. Indeed, in practice Markowitz's optimization finds some limitations, mainly due to the very poor effectiveness of the model over extended periods of time as potential change in the markets dynamics often occour, and moreover because it only relies on standard deviation as a measure of risk, neglecting other potential relevant information that might impact portfolio perfomance. On the other hand, as said, it improved the portfolio selection theory and it opened the road for many researchers to refine his theory.

Recalling the key assumptions of the model:

- Rational and risk-averse investors: Investors are assumed to be perfectly informed and objective, seeking to maximize their expected utility. This utility function is concave and increasing. Moreover, investors are price taker, meaning that they believe that cannot influence assets' prices;
- Portfolio risk as return variability: The risk of a portfolio is solely assessed based on its variability of returns, typically measured by standard deviation;
- Single-period investment horizon: All investors share the same investment horizon, focusing on maximizing their expected returns or minimizing risk within a single period;
- Return-risk trade-off: Investors aim to achieve the optimal balance between expected return and risk. They either seek to maximize returns for a given risk level (efficient frontier) or minimize risk for a desired return level;
- Asset universe and divisibility: In the market assets are assumed to be finite and infinitely divisible;
- Short selling, arbitrage and transaction costs: the market is characterized by the absence of arbitrage and transaction costs. Instead, short selling is allowed.

Generally, considering the returns of an assets and the probability associated to each of them to occur, the portfolio's expected returns is nothing but the weighted sum of the returns of the single assets:

$$E(p) = E(\sum_{i=1}^{m} R_i p_i) \tag{1}$$

Whilst the variance of the portfolio is:

$$\sigma_p^2 = \sum_{i=1}^{n} (R_i - E(R_i))^2 p_i \tag{2}$$

Hence, given n risky assets and assuming that all the money available are invested and that short selling is available[1], the Markowitz mean-variance model assuming a minimization cost function can be written as:

$$\begin{aligned} min_X \sum_{j=1}^{n} \sigma_j^2 X_j^2 + 2 \sum_{i=1}^{n} \sum_{j>1} \sigma_{ij} X_i X_j \\ \sum_{j=1}^{n} X_j = 1 \\ \sum_{j=1}^{n} E(R_j) X_j = \mu \end{aligned} \tag{3}$$

Whilst in case of maximization problem, the model becomes:

$$\begin{aligned} max_X \sum_{j=1}^{n} E(R_j) X_j \\ \sum_{j=1}^{n} X_j = 1 \\ \sum_{i=1}^{n} \sum_{j=1} \sigma_{ij} X_i X_j = \sigma^2 \end{aligned} \tag{4}$$

By changing the values of μ in Equation 3 or the values of σ in Equation 4, it is then possible to get an empirical representation of the efficient frontier. Moreover, $\sum_{i=1}^{n} \sum_{j=1} \sigma_{ij} X_i X_j$ represents the covariance between two given assets, useful in determining which assets should be included in the portfolio.

2.2 From MPT to Machine Learning

With his groundbreaking idea, Markowitz has paved the way for many expansions, where the overall idea is to use a different optimization objective.

For example, we could be interested in just minimizing risk (as we are not so much interested in expected returns), which leads to the *minimum variance portfolio* [6]. Several other optimization objectives have been studied during the the years, leading to different portfolio allocation strategies, some examples are *Maximum diversification portfolio* [7][8], *Maximum decorrelation portfolio* [9], *Risk parity portfolio* [24][31].

[1] In case of no short selling assumption, another parameter should be added to the model: $X_j > 0$

These optimization methods serve as the typical way to tackle the planning challenge of determining the most favourable portfolio allocation. All these financial methods to solve the asset allocation problem, treat the problem as a one-step optimization problem, with convex objective functions, and they all present multiple limitations:

- they do not relate market conditions to portfolio allocation dynamically.
- they do not take into account that the result of the portfolio allocation may potentially be evaluated much later.
- they make strong assumptions about risk.

[3]

As computational power and data availability increased, increasingly advanced machine learning techniques have begun to be used in more and more areas of daily life. This happens also with the asset allocation problem, it is since the mid-2010s that machine learning techniques, particularly deep learning algorithms, started to be extensively explored for asset allocation purposes. Researchers and practitioners began to investigate how neural networks, reinforcement learning, and other advanced machine learning methods could enhance traditional portfolio optimization approaches.

2.3 Reinforcement Learning and Asset Allocation

Due to the distinct nature of the portfolio allocation community compared to that of reinforcement learning, RL methods have been ignored for a significant period, despite the increasing interest in utilizing reinforcement learning and deep reinforcement learning in recent years.

Nevertheless, due to its ability to handle complex, dynamic environments and learn optimal decision-making strategies through trial and error, Reinforcement learning has garnered increasing interest in the field of portfolio management. Some of the key points that contribute to the rise of reinforcement learning in this domain are:

Dynamic environment handling: Financial markets are highly dynamic and influenced by various factors such as economic indicators, geopolitical events, and market sentiment. Traditional optimization methods struggle to adapt to these changing conditions. RL, instead can continuously learn and update its strategies based on new information.

Flexibility in decision making: RL allows for flexible decision-making processes, enabling portfolios to adapt to different investment objectives, risk preferences, and constraints. This flexibility is crucial for managing diverse portfolios with varying characteristics and objectives.

Model-Free Approach: Unlike traditional optimization methods that often rely on specific assumptions or models of asset returns, RL can be a model-free approach. It learns directly from data and experiences without requiring explicit models of the underlying dynamics. This can be advantageous in situations where the true data-generating process is complex or unknown.

Handling Non-Stationarity: Financial markets are characterized by non-stationary behaviour, where statistical properties change over time. RL algorithms, particularly those based on deep learning, have shown promise in adapting to non-stationary environments and capturing underlying patterns in the data.

Thanks to these potentials there has been a significant increase in research and development efforts focusing on applying RL to asset allocation and portfolio optimization problems. Researchers are exploring various RL algorithms, such as Q-learning, policy gradients, and actor-critic methods, to develop effective investment strategies.

In recent years, numerous research studies on the use of reinforcement learning have captured significant attention, not only because they no longer need to predict price, but also owing to their exceptional performance in areas such as Board Games, Go, and Video Games [35], which bear some resemblance to the dynamics of financial markets.

After the pioneering work, "Cryptocurrency Portfolio Management with Deep Reinforcement Learning" conducted by Z.Jiang J.Liang in 2016 [15] several research papers set out to develop a reinforcement learning agent that was capable of autonomously managing a portfolio of stocks in the best possible way.

Gao et al. [13] discretized the action space which is defined as the weight of portfolio wealth in different stocks, and combines Convolutional Neural Network (CNN) with dueling Q-net (DQN) to obtain the optimal actions. However, finite action space models might result in local optimum or no optimum at all, as they are limited by the number of action combinations. Although market actions can be discretized, discretization is considered a major drawback, because discrete actions come with unknown risks.

A general-purpose deep RL framework to deal with continuous action spaces, was published in 2015 in the paper *"Continuous control with deep reinforcement learning"* [21] introducing the actor-critic Deterministic Policy Gradient Algorithms. The continuous output in these actor-critic algorithms is achieved by a neural-network approximated action policy function, and a second network is trained as the reward function estimator. Zhang et al. [44], used this framework to implement a Deep Deterministic Policy Gradient (DDPG) agent specifically designed for multi-asset trading strategy in continuous action space. This dissertation is most closely related to this latter paper.

Other works that should definitely be mentioned in terms of the novelties contributed in this area are; the study by Jiang et al. (2017)[16] introduced a deep RL framework for portfolio management, utilizing recurrent neural net-

works to learn allocation policies directly from historical data.

Li et al. (2018)[20] conducted a comprehensive survey on RL techniques in portfolio management, reviewing various algorithms such as Q-learning, policy gradients, and actor-critic methods. They discussed the applications of these methods and their performance in optimizing portfolios.

Ong et al. (2019)[29] explored the use of deep RL techniques, including deep Q-learning and deep deterministic policy gradients, in portfolio management. They investigated the effectiveness of these methods using real-world stock market data.

Jin et al. (2019) [17] proposed a novel deep RL framework called Deep Distributional Reinforcement Learning (DDRL) for risk-aware portfolio optimization. Their approach leveraged distributional prediction to achieve better portfolio performance.

Cartea et al. (2020) [4] provided a comprehensive review of RL techniques applied to automated trading in financial markets, including portfolio optimization and market making. They discussed the challenges and opportunities of using RL for trading strategies.

Lyu et al. (2021) [23] developed a model-free RL algorithm for optimal trading with alpha predictors. Their approach dynamically adjusted trading strategies based on alpha signal predictions to maximize cumulative returns.

Thomas et al. (2021) [38] introduced a RL approach to portfolio optimization combining policy gradient methods with deep neural networks. They demonstrated the effectiveness of their method in adapting to changing market conditions through empirical results.

Durall (2022) [11], Compares various RL algorithms for asset allocation, finding PPO to be most effective in achieving long-term goals.

Overall, these studies underscore the growing interest and potential of reinforcement learning techniques in addressing the complexities of asset allocation and portfolio optimization in financial markets.

3 Reinforcement Learning: Fundamental theoretical concepts

Together with supervised and unsupervised learning, Reinforcement Learning(RF) is one of the three main paradigms of machine learning.

"Reinforcement learning is learning what to do - how to map situations to action - so to maximise a numerical reward signal."[37]

In other words, reinforcement learning is a framework where an **agent**, which is a decision maker, tries to learn the best sequence of actions to take, in order to maximise the **rewards** obtained from the interaction with the surrounding **environment**.

Reinforcement Learning is very close to the kind of learning that humans and other animals experience, indeed many of the core algorithms of RF were originally inspired by biological learning systems.

Beyond the agent and the environment, other main concepts in the RF framework are: *policy*, *reward signal*, *value function*, *model*(optional) and *state*.

A **policy** defines the learning agent's way of behaving at a given time. Roughly speaking, a policy is a mapping from the current state to a probability distribution of the actions to be taken.

The **reward signal** is a critical part of reinforcement learning, as it serves as the primary feedback mechanism that guides the agent's behaviour. At each time step, the *environment* sends to the agent a numerical value called *reward*. The agent's sole objective is to maximise the cumulative reward it receives over the long run. Consequently, the reward signal delineates favourable and unfavourable outcomes for the agent. We can think of the *reward* as the experience of pleasure and pain in biological contexts. Essentially, the reward signal serves as the primary foundation for adjusting the policy; if an action selected by the policy is followed by a low reward, then the policy may be changed to select some other action in that situation in future.

Whereas the reward signal indicates what is good in an immediate case, a **value function** indicates what is good in the long run. The value of a state tells us how good a particular state is in terms of the expected cumulative reward that can be obtained starting from that state and following a particular policy. Our objective is to identify actions that lead to states with the highest value, rather than just the highest immediate reward. Unfortunately, it is much harder to determine values than it is to determine rewards (we will see how one of the main challenges in RF is to find a method for efficiently estimating values).

A **model** of the environment is something that mimics the behaviour of the

environment, or more generally, that allows inferences to be made about how the environment will behave. Methods for reinforcement learning that use models are called *model-based* methods, as opposed to *model-free* methods that are explicitly trial-and-error learners.

Furthermore, we have that Reinforcement learning relies heavily on the concept of **state**, as input to the policy and value function, and as both input to and output from the model. A state in reinforcement learning represents the current environment in which the agent is located. This state can be observed by the agent, and it includes all relevant information about the environment that the agent needs to know in order to make a decision. We assume that the state signal is produced by some preprocessing system that is nominally part of the agent's environment. We do not address the issues of constructing, changing, or learning the state signal in this dissertation.

3.1 Finite Markov Decision Processes (RF framework)

A Reinforcement Learning framework is formulated as a **Markov Decision Process**(MDP). MDPs are a mathematical framework used to model decision-making and learning situations where an agent interacts with an environment to achieve a certain goal. These interact continually, with the agent selecting actions and the environment responding to these actions presenting new situations to the agent and giving rise to rewards that the agent tries to maximise over time.

3.1.1 The Agent-Envirionment Interface

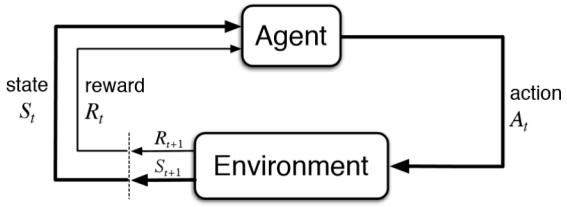
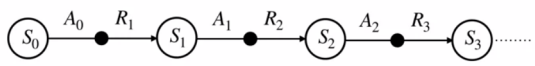


Figure 1: RL basic framework

The *agent* and *environment* interact during successive discrete time steps, $t = 0, 1, 2, 3, \ldots$. At each step t, the agent receives some representation of the environment's state, $S_t \in S$, and on that basis selects an action, $A_t \in A(S_t)$. One time step later, as a consequence of its action, the agent receives a numerical reward, $R_{t+1} \in R \subset \mathbb{R}$, and finds itself in a new state, S_{t+1}.



We can now try to formalize the dynamics of this interaction. Since the outcomes are stochastic we need to use the language of probabilities. When the agent takes an action in a certain state, there are many possible next states and rewards. The *transition dynamics function* P allows us to describe this notion mathematically.

$$p(s', r|s, a) \tag{5}$$

Given a state S and action a, p tells us the joint probability of the next state (s') and the reward (r) (for now we are assuming that the set of states, actions, and rewards are finite).

In a MDP, at a given instant t the next state S_{t+1} only depends on the current state S_t and the decision maker's action A_t, in other words, the probability of each possible value for S_t and R_t depends only on the immediately preceding state and action. The state is said to have the ***Markov property***.

Although the state might not strictly adhere to the Markov property, it remains beneficial in reinforcement learning to conceptualize it as an approximation of a Markov state, specifically, we want the state to be a good basis for predicting future rewards and for selecting actions. The Markov property is important in reinforcement learning because decisions and values are assumed to be a function only of the current state. (Understanding the theory of Markov case is crucial for extending it to the more complex and realistic non-Markov case).

3.1.2 Goals and Returns

In reinforcement learning the purpose of the agent is to maximise future rewards, this means to maximise cumulative reward in the long run (remember that at each step a reward is a number from a certain probability distribution). This idea is stated as the *reward hypothesis*:

"all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward)."

To formalize this idea in a mathematical way we need to introduce G, the

sum of the rewards obtained after time step t:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{6}$$

Note that G is a random variable because the dynamics of MDP can be stochastic, so what we want to maximise is the *expected return*, $E[G_t]$. This approach makes sense only if there is a natural notion of final time step, that is, when the agent-environment interaction breaks naturally into subsequences, which we call *episodes*, for example, we can consider the game of chess as an episodic task, where each game is an episode.

On the other hand, we often have to deal with cases where the agent-environment interaction continues without an end. We call these *continuing tasks*, in this case the return formulation (6) is problematic because we have $T = \infty$, and so the reward that we are trying to maximise could be also infinite.

We need to slightly modify the sum of the rewards so that it will always be finite, and we can do that by discounting future rewards by a discount rate. So now the agent will try to maximise the *expected discounted return*:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{7}$$

where γ is a parameter, $0 \leq \gamma \leq 1$.

In this way, immediate rewards contribute more to the sum, while rewards far into the future contribute less. If $\gamma = 0$, the agent is said to be "myopic", because it is concerned only with maximising immediate rewards. On the other side, as γ approaches 1, the agent takes future rewards more strongly into consideration, becoming more farsighted.

A very important property of the return is the fact that returns at successive time steps are related to each other, i.e. it can be written recursively:

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{8}$$

.

As we will see this is a very important property for the theory and algorithms of RL.

3.1.3 Value functions

A fundamental step in almost all RL algorithms consists of estimating the ***value function***. The value function is a critical component of RL that represents the expected long-term reward for a given state (or state action-pair). In simple words, we can say that it estimates "how good" is for an agent to be in a particular state (or to take a given action in that particular state) in terms of the expected cumulative reward that can be obtained by following a particular policy, allowing the agent to query the quality of its current situation.

Recall that the policy is what defines the way the agent makes decisions. A policy can be *deterministic*, in case it maps each state to a single action ($\pi(s) = a$), or *stochastic*, in this case, a policy assigns probabilities to each action in each state ($\pi(a|s)$), in this case, multiple actions can be selected with non-zero probability.

There are two types of value functions: *state value functions* and *action value function*.

The *state value functions*, denoted $v(s)$, represent the expected long-term reward being in a particular state s and following a particular policy π.

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right] \tag{9}$$

The *action value function*, denoted $Q(s, a)$, represents the expected long-term reward for taking a particular action a in state s and following a particular policy π.

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right] \tag{10}$$

A very important property of value functions is that they satisfy particular recursive relationships:

$$\begin{aligned}
&\mathrm{v}_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \\
&= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s \right] \\
&= \textstyle\sum_a \pi(a \mid s) \sum_{s'} \sum_r p\left(s', r \mid s, a\right) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} \mid S_{t+1} = s' \right] \right] \\
&= \textstyle\sum_a \pi(a \mid s) \sum_{s', r} p\left(s', r \mid s, a\right) \left[r + \gamma v_\pi\left(s'\right) \right]
\end{aligned} \tag{11}$$

The final expression is the ***Bellman equation*** for v_π. It states that the value of the start state s must be equal to the discounted value of the expected next state, plus the immediate reward.

The Bellman equation is one of the essential building blocks of RL since it allows us to describe the relationship between the value of a state ***s***, $v_\pi(s)$, and the values of its successor state ***s'***, $v_\pi(s')$. As we will see later, this recursive property allows us to compute the values of all states using an iterative approach.

Notice how it can be read as an expected value. It is a sum over all possible values of a, s and r. For each combination of these three variables, we compute its probability, $\pi(a|s)p(s', r|s, a)$, weight the quantity in brackets by that probability and then sum over all possibilities.

3.1.4 Optimal Value functions

In broad terms, we can say that solving a reinforcement learning task means finding a policy that will yield more rewards to the agent than all other policies.

A policy π is considered to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}$ for all $s \in S$.

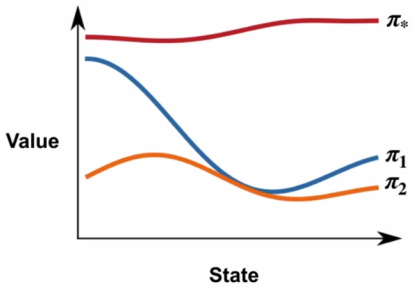


Figure 2: Optimal policy representation

A policy that is better than or at least the same as all other policies is called *optimal policy*.

Notice that there is always a policy that is better or equal to all other policies, and we define it as $\pi*$. Consider a scenario where we have a policy π_1 which does well in some states while a policy π_2 does well in other states, in such case we can always combine these policies into a third policy π_3 that consistently selects actions according to whichever policy π_1 or π_2 has the highest value in that state. π_3 will always have a value greater than or equal to both π_1 and π_2 in every state

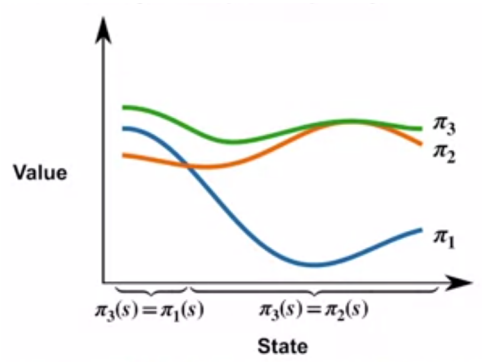


Figure 3: There is always an optimal policy

In a MDP there is always at least one optimal policy, but there may be more than one. These optimal policies are collectively denoted by $\pi*$. They share the same state-value function, called the *optimal state-value function*, denoted v_*, and they are defined as

$$v^{\pi^*} = v_*(s) = \max_\pi v_\pi(s) \tag{12}$$

for all $s \in S$. Optimal policies also share the same *optimal action-value function*, denoted q, and defined as

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a), \tag{13}$$

for all $s \in S$ and $a \in A(s)$.

Because v_* is the value function for a policy, it must satisfy the Bellman equation for state values.

$$v_*(s) = \sum_a \pi_*(a \mid s) \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma v_*\left(s'\right)\right]$$

Since we are dealing with an optimal policy we can rewrite the equation in a special form which doesn't reference any specific policy.

$$v_*(s) = \max_a \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma v_*\left(s'\right)\right] \tag{14}$$

This is because there always exists an optimal deterministic policy that selects an optimal action in every state. We call this equation ***Bellman optimality equation for v****.

We can make the same replacement for the action-value function and obtain the *Bellman optimality equation for* q^*:

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')] \tag{15}$$

The Bellman optimality equation is actually a system of equations, one for each state, so if there are n states, then there are n equations in n unknowns. But taking the maximum over actions is not a linear operation, so standard techniques from linear algebra for solving linear systems won't apply.

Recall that our goal is to find π^*, but, if we can find the values of v_* then it is relatively easy to determine the optimal policy π^*, having q^* makes choosing optimal actions even easier.

3.2 Dynamic Programming

"The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP)." [37]

Is important to keep in mind that DP algorithms are of limited use in RF since the assumption of perfect knowledge of the environment's model is unrealistic in a practical scenario, and also for their computational costs. Anyway, an understanding of DP methods is crucial for the understanding of the algorithm that we will see later. Indeed, all of these other methods can be seen as attempts to achieve much the same effect as DP, but with reduced computational demands and without the assumption of a perfect model of the environment.

The main idea is the use of value functions to organize and structure the search for good policies. As we said, if we can find the optimal value function then is easy to obtain the optimal policy.

In solving this problem we can distinguish two distinct tasks, *policy evaluation* and *policy improvement*.

Policy evaluation is the task of determining the value function for a specific policy.

Policy Improvement is the task of finding a policy to obtain as much reward as possible. In other words, finding a policy which maximizes the value function.

DP algorithms can be applied to solve these tasks, by turning Bellman equations (3.1.3) into an iterative updated rule for improving approximations of the desired value functions.

Finding a policy to obtain as much reward as possible is the ultimate goal of reinforcement learning. However the task of policy evaluation is usually a necessary first step.

3.2.1 Policy Evaluation

Policy evaluation is the task of determining the state value function v_π for a particular policy π (prediction problem).

Recall that the value of a state under a policy π is the expected return from that state if we act according to π.

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{16}$$

We have seen how, if the environment's dynamics are completely known, the Bellman equation reduces the problem of finding v_π to a system of linear equations, one equation for each state. So the problem of policy evaluation reduces to solving this system of linear equations. In principle, we could approach this task with various methods from linear algebra. In practice, the iterative solution methods of dynamic programming are more suitable for general MDPs.

The first DP algorithm that will see to solve this problem is called ***iterative policy evaluation***.

3.2.2 Iterative policy evaluation

Remember the Bellman equation gives us a recursive expression for v_π. The idea of iterative policy evaluation is to take the Bellman equation and directly use it as an update rule.

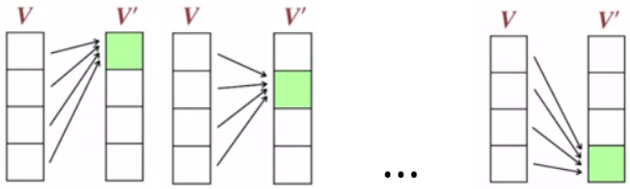
$$v_{\kappa+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\kappa(s')] \tag{17}$$

Now, instead of an equation which holds for the true value function, we have a procedure we can apply to iteratively refine our estimate of the value function (k is the index of each iteration step). This will produce a sequence of better and better approximations to the value function.

Let's consider a series of approximate value functions, $v_0, v_1, v_2, ...$, and so forth, each mapping S^+ to R. The initial approximation, v_0, is chosen arbitrarily (with the exception that the terminal state, if it exists, must have a value of 0). Subsequent approximations are determined by applying the Bellman equation (17) as an update rule. It can be demonstrated that, this sequence, converges to v_π as $k \longrightarrow \infty$.

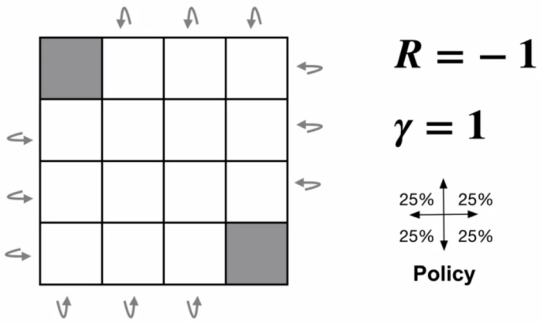
Let's look at an example to make everything clearer

To implement iterative policy evaluation, we store two arrays, each has one entry for every state. One array, which we label V stores the current approximate value function. Another array, V', stores the updated values. By using two arrays, we can compute the new values from the old one state at a time without the old values being changed in the process.



At the end of a full sweep(i.e. when all states have been updated), we can write all the new values into V; then we do the next iteration.

It is also possible to implement a version with only one array, in which case, some updates will themselves use new values instead of old. This single array version is still guaranteed to converge, and will usually converge faster. This is because it gets to use the updated values sooner. But for simplicity, let's start focusing on the two array version. Consider an example with the following characteristics.



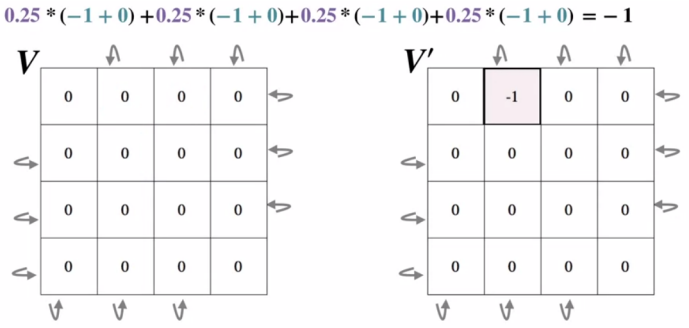
We are in the case of an episodic task with the terminal states located in the top left and bottom right corner.

The order through which we updated the states is not important when we use the 2 arrays approach, let's assume we sweep the states from left to right, and then from top to bottom. Values are all initialized to 0 (remember that final states must be equal to 0 and they will not be updated).

Let's see the first iteration, applying the equation (17).

Consider the first addend of the sum, corresponding to the action "go to left" :

- 0.25 corresponds to $\pi(a|s)$, the probability of taking that action in the current state
- -1 is the reward
- 0 is the value function of the state s' (the state in which the agent will be after taking the action)



In this case, the dynamic function is deterministic, so only the reward and value for one s' contribute to the sum. In the equation to calculate the updated value of the state we will have as many addends as there are possible actions. And we repeat the process for all the states, concluding one *sweep*.

After completing the sweep, we copy the updated value from V' to V, and we repeat the process.

Here is the complete algorithm:

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated

$V \leftarrow \vec{0}, V' \leftarrow \vec{0}$

Loop:

$\Delta \leftarrow 0$

Loop for each $s \in \mathcal{S}$:

$$V'(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)[r + \gamma V(s')]$$

$$\Delta \leftarrow \max(\Delta, |V'(s) - V(s)|)$$

$V \leftarrow V'$

until $\Delta < \theta$ (a small positive number)

Output $V \approx v_\pi$

We can notice how the outer loop continues until the change in the approximate value function becomes small (smaller than θ, a constant defined by us).

3.2.3 Policy Improvement

We just saw how to compute the value function for a given arbitrary policy, the reason to do that is to help find better policies.

Now, we would like to know whether in some states we should change the policy to deterministically choose an action $a \neq \pi(s)$.

Recall that we showed how given the optimal value function $v*$ we can find the optimal policy by choosing the greedy action, greedy action maximizes the the Bellman's optimality equation in each state.

$$\pi_*(s) = \arg\max_a \sum_{s'} \sum_r p(s', r|s, a)[r + \lambda v_*(s')] \tag{18}$$

Imagine selecting an action which is greedy with respect to the value function v_π of an arbitrary policy π.

$$\arg\max_a \sum_{s'} \sum_r p(s', r|s, a)[r + \epsilon v_\pi(s\prime)] \tag{19}$$

This new policy that we are now following is greedy with respect to v_π (remember that the current policy is still the same that we used to calculate the value function, in the example before the random policy with 1/4 of going in each direction). The first thing to note is that this new policy must be different from π. If this greedification doesn't change π, then it means that π was already optimal, and so that v_π obeys the Bellman's optimality equation.

The new policy is a strict improvement over π unless π is already optimal. This is a consequence of a general result called ***Policy improvement theorem***:

$$q(s, \pi'(s)) \geq q(s, \pi(s)) \; for \; all \; s \in S \longrightarrow \pi' \geq \pi \tag{20}$$

$$q(s, \pi'(s)) > q(s, \pi(s)) \; for \; at \; least \; one \; s \in S \longrightarrow \pi' > \pi \tag{21}$$

This theorem formalizes the idea that if taking action a according to π' and then following policy π has a higher (or equal) value than taking the first action according to π, then π' must be better, and thus obtain a greater or equal expected return from all the states.

This process of improving an original policy by making it greedy with respect to the value function of the original policy is called ***policy improvement***.

$$\begin{aligned} \pi'(s) &= \arg\max_a q_\pi(s, a) \\ &= \arg\max_a \sum_{s'} \sum_r p(s', r|s, a)[r + \lambda v_\pi(s\prime)] \end{aligned}$$

Note that we are assuming a deterministic policy. In the general case of stochastic policy, the idea is still the same, we will just have that $q_\pi(s, \pi'(s)) = \sum_a \pi'(a|s) q_\pi(s, a)$

3.2.4 Policy Iteration

Once a policy, π, has been improved using v_π to yield a better policy, π', we can then compute $v_{\pi'}$ and improve it again to yield an even better π', and we can repeat these steps to find the optimal policy by iteratively evaluating and providing a sequence of policies.

We can evaluate π_0 using iterative policy evaluation to obtain the state value, V_0, this is called **evaluation step**.

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0}$$

We can then greedify with respect to v_{π_0} to obtain a better policy, π_1. We call this the **improvement step**.

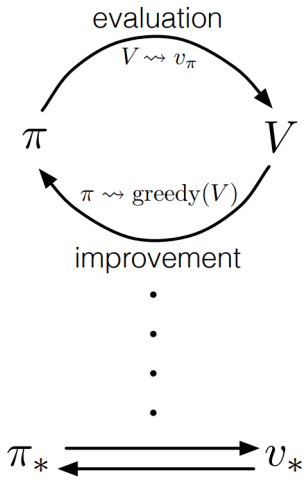
$$v_{\pi_0} \xrightarrow{\text{I}} \pi_1$$

Repeating this process we obtain a sequence of always better policies.

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

Each policy is guaranteed to be an improvement on the last unless the last policy was already optimal. Each policy generated in this way is deterministic, and since a finite MDP has only a finite number of policies, this process must converge to an optimal policy in a finite number of iterations.

This method of finding an optimal policy is called *policy iteration*.



This "dance" of policy and value proceeds back and forth, until we reach the only policy, which is greedy with respect to it's own value function, the optimal policy. At this point, and only at this point, the policy is greedy and the value

function is accurate. We can visualize this process as a bouncing back and forth between 2 lines, one representing the accurate value functions and the other one the greedy policies. These two lines intersect only at the optimal policy and value function.

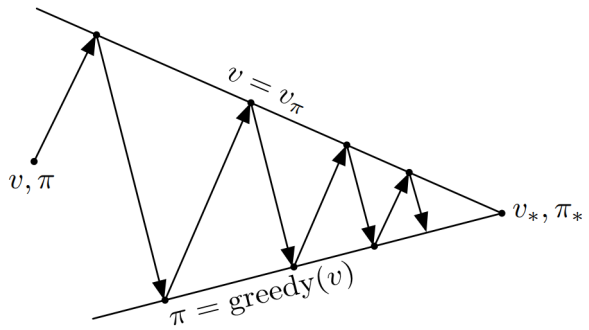


Figure 4: Policy Iteration Visualization

Here's what the complete procedure looks like in pseudo-code.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 5: Policy Iteration pseudo-code

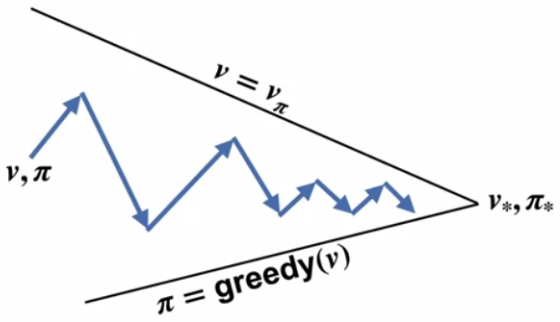
3.2.5 Generalized policy iteration

Right now we have seen policy iteration as a fairly rigid procedure where we alternate between evaluating the current policy and improving the evaluated policy, to eventually converge to the optimal policy. One drawback of this procedure is that each of its iterations involves policy evaluation, which is itself an iterative procedure, which may require multiple sweeps through the state set in order to precisely approximate the current value function.

The *Generalized policy iteration framework* allows to prove that the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration.

The policy iteration algorithm runs each step all the way to the best possible approximation of v_π and π. Imagine relaxing this condition of always converging to the best possible approximation. Now each evaluation step brings our estimate a little closer to the value of the current policy but not all the way, and, each policy improvement step makes our policy a little more greedy, but not totally greedy.

Visually we will move from a situation as shown in figure 4 to a situation like this:



This process will still make progress towards the optimal policy and value function.

We will use the term ***Generalized policy iteration*** to refer to all the ways we can interleave policy evaluation and policy improvement.

One of the most known Generalized policy algorithms is called *Value Iteration*, which can be written as a particularly simple backup operation that combines the policy improvement and truncated policy evaluation steps. We still sweep over all the states and greedify with respect to the current value function, however, we don't run policy evaluation to completion but we perform just one sweep over all the states.

3.3 Model Free Reinforcement Learning: Temporal-Difference Learning

When the idea behind Temporal-DIfference Learning came out, it represented a turning point in the world of RL. Roughly speaking, TD learning is a combination of Monte Carlo ideas and dynamic programming ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics (they are model-free approaches). Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).

Notice that we haven't, and we won't talk in-depth about Monte Carlo(MC) methods. All we need to know to completely understand the following pages is that Monte Carlo methods are a family of reinforcement learning algorithms that learn by averaging the returns they receive from each state-action pair, so their characteristic is that they have to wait until the end of each episode before updating the state-values estimates but they don't need a model of the environment's dynamics .[2]

Another thing to pay attention to is the fact that the theory related to reinforcement learning is very broad and growing, with more and more new approaches aimed at solving specific types of problems. From this chapter onward, we will focus more and more on the theoretical concepts that are most relevant to being able to understand the algorithm that was used to solve the problem that this thesis poses, inevitably going to neglect other concepts that are also fundamental to understanding other types of approaches.

3.3.1 TD Policy Evaluation

Let's start by focusing on the policy evaluation problem, this represents the main difference between DP, TD, and Monte Carlo methods, since for the control problem(finding an optimal policy) they all rely on some variation of the *Generalized Policy Iteration*.

As we already discussed, the Policy evaluation problem consists of finding the value function v_π for the given policy, for this reason, is also called the *prediction problem*.

Both TD and MC methods, are model-free, they do not need to know the transition probabilities or reward function of the environment, but they rather use **experience** they gain following π to solve the policy evaluation problem. The big difference comes from the fact that MC methods need to wait until the end of the episode, to then use the return G_t to update $V(S_t)$:

[2] For a detailed explanation of Monte Carlo methods refer to Chapter 5 of "Reinforcement Learning: An Introduction" (2018) Sutton, Barto .

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{22}$$

where α is a constant step-size parameter.

Here comes the big difference with respect to TD, while MC methods have to wait until the end of the episode to update $V(S_t)$, TD methods need only to wait until the next time step. At time $t+1$ they can already update the estimate of $V(S_t)$ using the observed reward R_{t+1} and the estimate $\mathrm{V}(\mathrm{S}_{t+1})$.

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{23}$$

Note that the quantity inside the square brackets is a sort of error, that measures the difference between the estimated value of S_t and the new estimate $R_{t+1} + \gamma V(S_{t+1})$. We will refer to this quantity as:

$$\delta_t = [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{24}$$

3.3.2 Q-learning: Off-policy TD Control

When it comes to using the TD prediction for the control problem (finding the optimal policy), there are different methods available, the most common are:

- SARSA
- Q-learning

We will focus only on the latter since it is crucial to understand DDPG

Q-learning is an *off-policy* control algorithm that was introduced by Chris Watkins in 1989.

The *off-policy* learning consists in learning one policy while following another. This approach was introduced to deal with the exploration-exploitation dilemma, in this way we have 2 policies, one that is learned about and that becomes the optimal policy(called *Target policy*), and one that keeps exploring and is used to generate behaviour (called *Beahviour policy*).

Off-policy methods usually have a higher variance and are slower to converge, but on the other hand, they are more powerful and general.

Q-learning is defined as:

$$Q\left(S_t, A_t\right) \leftarrow Q\left(S_t, A_t\right) + \alpha\left[R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a\right) - Q\left(S_t, A_t\right)\right] \tag{25}$$

In this case, the learned action-value function, Q, directly approximates $q*$ (optimal action-value function), independent of the policy being followed (Behaviour policy), which still has an effect of determining which state-action pairs

are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated.

The goal of Q learning is to find the optimal policy by learning the optimal Q values for each state action pair. The Q learning algorithm iteratively updates(value iteration) the Q values for each state action pair using the Bellman optimality equation equation until the Q function converges to the optimal Q:

$$q*(s,a) = E[R_{t+1} + \gamma \max_{a}^{'} q*(s',a')] \tag{26}$$

Notice that is just another way of writing the equation (15). It has been shown that Q converge with probability 1 to $q*$.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal,\cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 6: Q-Learning pseudo-code

However, when we have a lot of possible states, it becomes almost impossible to use Q-learning to calculate the optimal Q-value for all the states, we will instead use a function approximation to estimate the optimal Q function.

3.4 Approximate Solution Methods

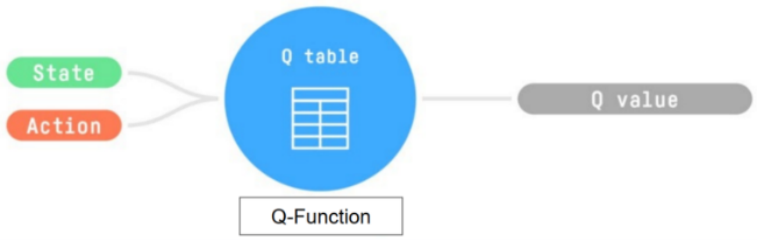
So far we have always considered the state space in tabular form, where it was possible to easily visualize the next possible states and actions. However, in many of the tasks to which we would like to apply reinforcement learning the state or action space is enormous. In these cases, traditional RL algorithms, such as Q-learning and policy iteration, can become intractable when dealing with such large or complex problems.

In these cases we can't expect to be able to find an optimal policy or optimal value function; our goal will be to replace the exact value function or policy with a parameterized function that best approximates the value function or the policy, this allows RL algorithms to learn from experience and improve

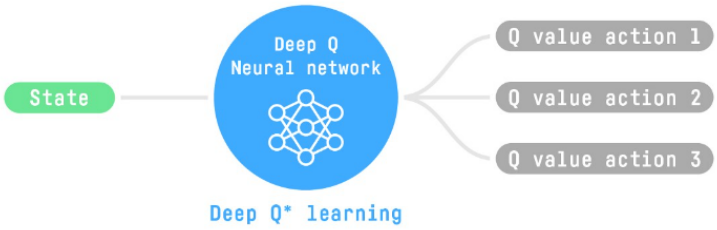
their performance without having to explicitly sore and evaluate the entire state space. Different techniques can be used to estimate the parameter of the function, ranging from simple linear methods to more complex machine learning and deep learning techniques.

3.4.1 Deep Q-Learning

As we discussed in chapter 3.3.2 Q-learning is an algorithm that allows us to learn the Q-function, an action-value function that determines the value of being in a particular state and taking a specific action. But we have the problem mentioned in the latter chapter, i.e., Q-Learning is a tabular method. This becomes a problem if the states and actions spaces are not small enough to be represented in a tabular form. In other words: it is not scalable.



The solution to this problem becomes to approximate the Q-values using a parametrized Q-function $Q_\theta(s, a)$. If we use a neural network to find the parameters of this function we have what is called ***Deep Q-Learning***.



Given a state, the neural network will approximate the Q-values for each possible state-action pair.

Unlike Q-learning, where we update the Q-value of a state-action pair directly (25), with Deep Q-Learning we create a loss function that compares our Q-value prediction and the Q-target and uses gradient descent to update the weights of our Deep Q-Network(DQN) to better approximate our Q-values.

Let's better explain the algorithm going through the pseudo-code.

We can identify two phases: ***sampling*** and ***training***. In the first one, the agent takes and action and store the observed experience (state, action, reward, next state) in a *replay memory*, then, in the training phase, a small batch of stored information is randomly selected, and a learn from this batch is performed using a gradient descent update step in order to find the parameters that better approximate the target Q-function (by minimizing the loss).

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**

Sampling

With probability ε select a random action a_t
otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
Execute action a_t in emulator and observe reward r_t and image x_{t+1}
Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

Training

Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

$$\text{Set } y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$$

Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
Every C steps reset $\hat{Q} = Q$

  End For
End For

[12]

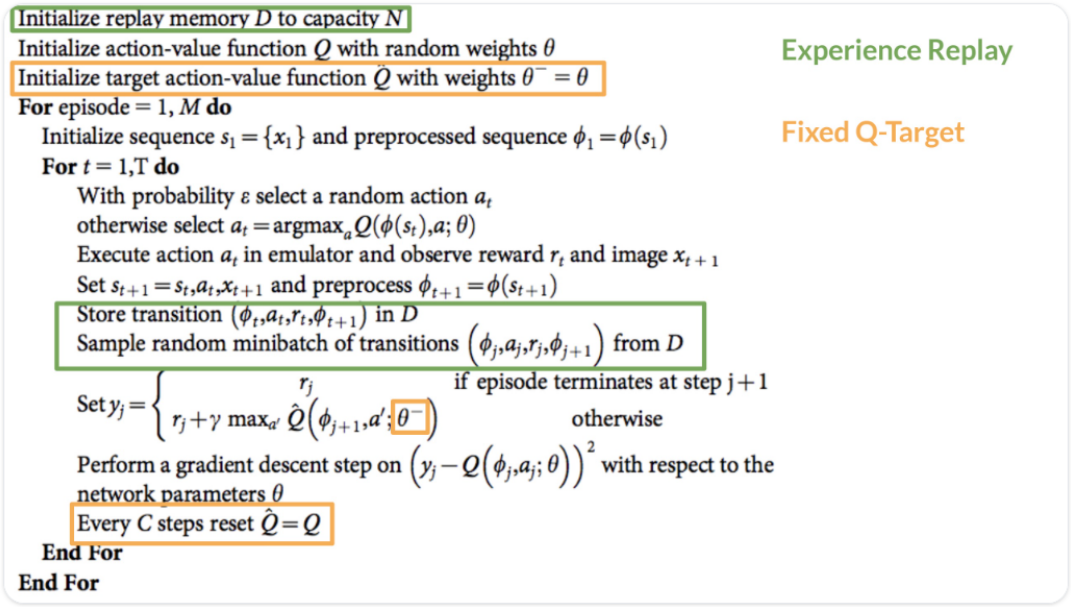
One of the key obstacles in applying Deep Q-Learning lies in the fact that Q-function is usually a non-linear function, which often leads to numerous local minima. This complexity can hinder the neural network's ability to effectively converge towards the accurate Q-function. Deep Q-learning might also suffer from instability because of combining a non-linear function Q-value function (Neural Network) and bootstrapping (since we update target with estimated values).

One technique that is used to address these problems is the use of the *Replay Buffer* (what we have just seen in the sampling phase). This helps to decorrelate the data and make the learning process more stable, it also allows to make a more efficient use of the experiences done during the training. Typically, in online reinforcement learning, the agent engages with the environment, gathers experiences (including state, action, reward, and next state), learns from them (by updating the neural network), and then discards them, which is inefficient. Saving the experience samples in the replay buffer to be then reused during the training, makes the training phase much more efficient.

Another reason that causes instability of results is the fact that when we

want to calculate the loss to minimise (i.e. TD error), that is the difference between the TD target (Q-Target) and the current Q-value (estimation of Q), we don't know the real TD target, but we need to estimate it using the Bellman equation. The problem is that we are using the same parameters (weights) for estimating the TD target and the Q-value. Consequently, there is a significant correlation between the TD target and the parameters we are changing. With each training step, both our Q-values and the target values shift. We're getting closer to our target, but the target is also moving. It's like chasing a moving target! This can lead to significant oscillation in training.

The technique proposed to address this issue consists of creating a separate *target network to estimate the target Q-values*. The target network is a copy of the main neural network with fixed parameters used to estimate the TD target. The target network is then periodically updated copying the parameters from our Deep Q-Network. This allows to stabilize the training process and mitigate issues related to target value estimation.

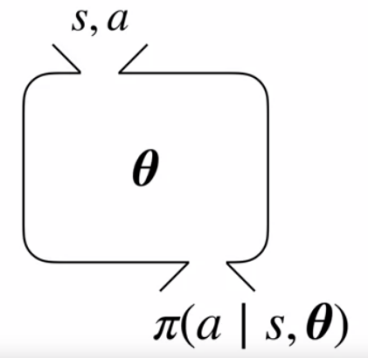


[12]

3.4.2 Policy Gradient Methods

We introduce here something completely new with respect to what we discussed before about RL theory. So far all the methods that we have seen, were *action-value methods*, which means, they learned the values of actions and selected the action based on their estimated action values. We will now talk about methods that instead learn a *parametrized policy* that can select actions without

consulting a value function. We can consider a parametrized policy which maps states directly to actions without first computing action values.



$\pi(a|s,\boldsymbol{\theta}) = Pr\{A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\}$ is the probability that action a is taken at time t given that the environment is in state s at time t, this mapping is controlled by parameter $\boldsymbol{\theta}$ ($\boldsymbol{\theta} \in R^{d'}$ is the *policy's parameter vector*).

The parameterized function has to generate a valid policy. This means it has to generate a valid probability distribution over actions for every state. Specifically, the probability of selecting an action must be greater than or equal to zero. For each state, the sum of the probabilities over all actions must be one.

Like with Deep Q-Learning, the basic idea to learn and improve a parameterized policy is to specify an objective, and then figure out how to estimate the gradient of that objective from an agent's experience.

The learning of the policy parameters is based on the gradient of some scalar performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameter $\boldsymbol{\theta}$.

These methods seek to maximize performance, so their updates approximate gradient ascent in J:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\widehat{\nabla J\left(\boldsymbol{\theta}_t\right)} \tag{27}$$

Where $\widehat{\nabla J\left(\boldsymbol{\theta}_t\right)} \in R^{d'}$ is a stochastic estimate whose expecatition approximates the gradient of the performance measure with respect to its argument $\boldsymbol{\theta}_t$.

We can use the discounted reward as an objective for policy optimization. We will discuss how to optimize this objective from sampled experience. [36]

All methods that follow this general schema are called *policy gradient methods*. Some of these methods also include learning the value function, these methods that learn both policy and value function are called *actor-critic methods*, where 'actor' is a reference to the learned policy, and 'critic' refers to the value function.

3.4.3 Actor-Critic Algorithm

Even within policy gradient methods, value-learning methods like TD can still have an important role. In the so-called actor-critic setup, we use both the parameterized policy and the value function estimation. The parameterized policy plays the role of an actor, while the value function plays the role of a critic, evaluating the actions selected by the actor.

- **Actor**: learns an optimal policy by exploring the environment
- **Critic**: assesses the value of each action taken by the Actor to determine whether the action will result in a better reward, guiding the Actor for the best course of action to take.

After we execute an action, we use the TD error to decide how good the action was compared to the average for that state. If the TD error is positive, then it means the selected action resulted in a higher value than expected. Taking that action more often should improve our policy. That is exactly what this update does. It changes the policy parameters to increase the probability of actions that were better than expected according to the critic. Correspondingly, if the critic is disappointed and the TD error is negative, then the probability of the action is decreased. The actor and the critic learn at the same time, constantly interacting. The actor is continually changing the policy to exceed the critics expectation, and the critic is constantly updating its value function to evaluate the actors changing policy.

Actor-Critic (continuing), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a \mid s, \boldsymbol{\theta})$

Input: a differentiable state-value function parameterization $\hat{v}(s, \mathbf{w})$

Initialize $\bar{R} \in \mathbb{R}$ to 0

Initialize state-value weights $\mathbf{w} \in \mathbb{R}^d$ and policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g. to 0)

Algorithm parameters: $\alpha^{\mathbf{w}} > 0$, $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\bar{R}} > 0$

Initialize $S \in \mathcal{S}$

Loop forever (for each time step):

$A \sim \pi(\cdot \mid S, \boldsymbol{\theta})$

Take action A, observe S', R

$\delta \leftarrow R - \bar{R} + \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$

$\bar{R} \leftarrow \bar{R} + \alpha^{\bar{R}} \delta$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S, \mathbf{w})$

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \nabla \ln \pi(A \mid S, \boldsymbol{\theta})$

$S \leftarrow S'$

3.5 Deep Deterministic Policy Gradient

Introduced for the first time by Deepmind's publication "Continuous Control With Deep Reinforcement Learning" (Lillicrap et al, 2015) [21], *the deep deterministic policy gradient (DDPG) algorithm is a model-free, online, off-policy reinforcement learning method which use an actor-critic reinforcement learning agent that searches for an optimal policy that maximizes the expected cumulative long-term reward.*[1]

.

A DDPG agent can be trained in environments with continuous or discrete observation spaces and continuous action spaces. This makes it appropriate for asset allocation problems, which have a continuous observation space and a continuous action space. Note how DQN is not able to deal with continuous action spaces since it works with a neural network with the number of output nodes that need to be equal to the number of possible actions.

Nevertheless, DDPG is very similar to DQN, since it use the same technique that DQN use to resolves the instabilities of the convergence of the results:

- Replay buffer
- Target Network

That's why we can think of the DDPG as a sort of DQN for continuous action spaces.

In addition to these techniques, DDPG also use an Actor-Citric approach, so the algorithm uses in total four neural networks: a Q network (Critic), a deterministic policy network (Actor), a target Q network (Target Critic), and a target policy network (Target Actor), with the following parameters:

- θ^Q: Q network
- θ^μ: Deterministic policy network
- $\theta^{Q\prime}$: Target Q network
- $\theta^{\mu\prime}$: Target policy network

Here the pseudo-code for the algorithm:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, 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 observe 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 better understand how the algorithm work we can identify 3 stages:

1. Experience replay
2. Actor & Critic network updates
3. Target network updates

3.5.1 Experience replay

As in Deep Q-learning, DDPG employs a replay buffer to sample past experiences to update neural network parameters. Throughout each experienced episode, all experience tuples (state, action, reward, and next state) are preserved into a finite-sized memory known as a "replay buffer." Subsequently, random mini-batches of experiences are sampled from this buffer when updating the value and policy networks (the reason for using a replay buffer has already been discussed in chapter (3.4.1).

3.5.2 Actor (Policy) & Critic (Value) network updates

The actor-network takes the current state as input and outputs a deterministic action plus some random noise to favourite exploration, in then observe the reward r_t, and new state $s_t + 1$, and eventually the store transition (s_t, a_t, r_i, s_{t+1})

in the Replay buffer. The critic network takes the current state and the action chosen by the actor as input and outputs the estimated value of that state-action pair (Q-value) evaluating in this way the action. The goal is to update the parameters of the actor(θ^μ) to make him able to always select the action with the highest expected reward, in other words to learn a deterministic policy $\mu_\theta(s)$ which gives the action that maximizes $Q_{\theta^\mu}(s, a)$.

To do that, the algorithm goes ahead by sampling a minibatch of experiences from the replay buffer and computes the target Q-value. Similar to Q-learning the target Q-value is obtained through the Bellman equation:

$$y_i = r_i + \gamma Q' \left(s_{i+1}, \mu' \left(s_{i+1} \mid \theta^{\mu'} \right) \mid \theta^{Q'} \right) \tag{28}$$

It then updates the critic network parameters by minimizing the mean squared error between the predicted Q-values and the target Q-values, with a one-step gradient descent:

$$Loss = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i | \theta^Q))^2 \tag{29}$$

Note how the Q value is calculated with the critic network, not with the target critic.

To update the actor-network, we want to learn a deterministic policy $\mu_\theta(s)$ which gives the action that maximizes $Q_{\theta^\mu}(s, a)$. Because the action space is continuous, and we assume the Q-function is differentiable with respect to action, we can just perform gradient ascent (with respect to policy parameters only) to solve:

$$\max_\theta E[Q(s, a) | s = s_t, a_t = \mu(s_t)] \tag{30}$$

Since the actor policy function is differentiable, the gradient can be calculated by using the chain rule of calculus to backpropagate the gradients through the critic network.

$$\nabla_{\theta^\mu} J(\theta) \approx \nabla_a Q(s, a) \nabla_{\theta_\mu} \mu(s | \theta^\mu)$$

since we are updating the policy in an off-policy way with batches of experience, we take the mean of the sum of gradients calculated from the mini-batch:

$$\nabla_{\theta^\mu} J(\theta) \approx \frac{1}{N} \sum_i \left[\nabla_a Q \left(s, a \mid \theta^Q \right) \big|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu \left(s \mid \theta^\mu \right) \Big|_{s=s_i} \right]$$

And then update the parameter of the actor network using the the gradient ascent method to maximize the expected return.

3.5.3 Target network updates

Slowly updates the Target Networks parameter via *soft updates*:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'} \tag{31}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\theta)\theta^{\mu'} \tag{32}$$

During each time step, a portion of the parameters from the Actor-Critic network are gradually transfered onto the target network. The degree of this transfer is controlled by a hyperparameter τ known as the target update rate (where $\tau << 1$).

In conclusion, this chapter has provided an overview of the theoretical foundations of the Deep Deterministic Policy Gradient algorithm, highlighting its key components and mechanisms.

Understanding the underlying principles of DDPG is essential for the practical implementation of the algorithm in real-world applications. In the next chapter, we will delve into the practical aspects of applying DDPG to the problem of asset allocation, where we will discuss data preparation, model architecture, training procedures, and performance evaluation. By leveraging the insights gained from this theoretical foundation, we aim to develop and deploy a DDPG-based agent capable of making informed investment decisions in dynamic market environments.

4 Experiment

4.1 Problem Statement

As we said, portfolio management is the action of continuous reallocation of capital into a number of different financial assets. The goal of the investors is to modify the weights assigned to each asset to obtain the maximum return possible given a certain risk aversion. In our case, the investor will be what we introduced in the previous chapter as the *Agent*. The agent will rebalance the portfolio on a daily base.

Before describing the details of how our approach tries to achieve this goal, it is proper to introduce a mathematical notation:

- M: Number of stocks
- $\mathbf{w}_t$: Portfolio vector, its i-th component represents the ratio of the total budget invested to the i-th asset at time t, such that:

$$\mathbf{w}_t = [w_{1,t}, w_{2,t}, ..., w_{M,t}]^T \in^M and \sum_{i=1}^{M} w_{i,t} = 1$$

  with

$$w_{i,t} \in [0, 1]$$

- $h_{i,t}$: Is the number of stocks i in the portfolio at time t

$$h_{i,t} = \left[\frac{V_t \times w_{i,t}}{p_{i,t}} \right]$$

- c_t: Cash owned at time t
- V_t = Portfolio value at time t

$$V_t = \sum_{i=1}^{M} h_{i,t-1} \times p_{i,t} + c_{t-1}$$

4.2 Asset Selection and Data Preparation

For the purpose of this study, 10 years of data with daily granularity was collected, from 2014 to the beginning of 2024. We choose five stocks, one for each of the five largest sectors of the S&P 500 index.

- Apple (AAPL) for the Technology sector
- JP Morgan (JPM) for Financial Services sector
- UnitedHealth Group (UHN) for HealthCare sector

- P&G (PG) for Consumer Discretionary
- ExxonMobil (XON) for Energy Sector

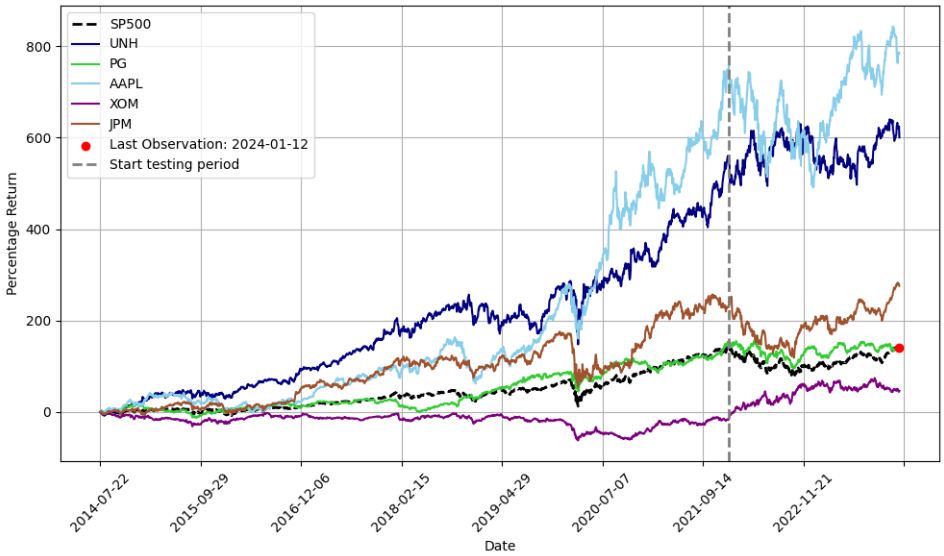


Figure 7: Percentage returns of the managed stocks

Data are downloaded from Yahoo Finance, and for each stock they include the basic financial information such as opening price, closing price and volumes. In the realm of practical trading, in addition to these raw stock data, it's crucial to consider a range of additional information, such as stock returns, technical indicators, and current macroeconomic variables. So, in addition to the usual daily just mentioned other useful information such as log returns (daily, weekly, monthly, and half-yearly) and price standard deviations (half-yearly and yearly) are calculated. Also, a number of technical financial indicators have been calculated. Such as:

- **Moving Average Convergence/Divergence (MACD):** Moving average convergence/divergence is a trend-following momentum indicator that shows the relationship between two exponential moving averages (EMAs) of a security's price. The MACD line is calculated by subtracting the 26-period EMA from the 12-period EMA.

  MACD has a positive value whenever the 12-period EMA is above the 26-period EMA and a negative value vice-versa. If the distance between the MACD and 0 increases, it indicates that the distance between the two EMAs is growing.

  Traders use MACD to identify changes in the direction or strength of a stock's price trend, helps traders detect when the recent momentum in a

stock's price may signal a change in its underlying trend. This can help traders decide when to enter, add to, or exit a position.

- **Relative Strength Index (RSI)**: The relative strength index (RSI) is a momentum indicator used in technical analysis to measure the speed and magnitude of an asset's recent price changes to evaluate overvalued or undervalued conditions in the price of that asset.

  The RSI is represented as an oscillator on a scale of zero to 100. Traditionally, a value of 70 or above indicates an overbought situation. A reading of 30 or below indicates an oversold condition. This information can be used to decide when to buy or sell. (It has been used a period of 30 days to calculate the RSI)

- **Bollinger Bands (boll ub, boll lb)**: Bollinger Bands are a technical analysis tool defined by a set of trendlines. They are represented as two standard deviations, both positively and negatively, away from a simple moving average (SMA) of a security's price and they can be adjusted to user preferences. They are a technical analysis tool to generate oversold or overbought signals.

  When the asset price continually touches the upper band, it can indicate an overbought signal, while when it touches the lower band it can indicate an oversold signal.

- **Directional Movement Index**: The directional movement index (DMI) is an indicator that identifies in which direction the price of an asset is moving. A value of the DMI bigger than 25 indicates a strong trend, either up or down. [3]

We add a variable "day" that tells us which day of the week the data refers to. Thus, for each stock in our portfolio, for each day when the markets are open, we will have 12 variables, representing the information that our agent can use to make informed decisions.

4.3 RL Components: Practical Implementation for Asset Allocation

At the beginning of the chapter regarding the fundamental theory of Reinforcement Learning, we introduce the main components of each RL framework. Let us now review them with regard to our specific application case.

4.3.1 The action

At every time step t (i.e. every trading day) the agent takes an action a_t. In our case, the action taken by the agent corresponds to deciding the ratio of the total worth to allocate to each of the assets in the portfolio, i.e.

$$a_t = \mathbf{w}_{t+1}$$

[3] All these financial definition are taken from https://www.investopedia.com/

4.3.2 The State

A state in reinforcement learning represents the observations that the agent has at his disposal, at a specific point in time, to take his decision (action).

We use the notation s_t to refer to the state at time t. In our case, the state consists of a matrix where each row corresponds to each stock [4], while each column corresponds to specific information about the stocks. So in the end the matrix will have dimension [x, y] where:

- x = number of stocks
- y = information available for each stock

We model the asset allocation process as a Markov Decision Process (MDP), so we assume that at time t, the agent, looking a the information contained in s_t has at his disposal all the information he needs to take action a_t and find himself in the state s_{t+1}.

4.3.3 The environment

The environment serves as the framework within which the reinforcement learning agent operates. In the context of algorithmic asset allocation, the environment is represented by the financial market and its behaviour. It encapsulates the dynamics of the asset market and defines the state space, action space, and reward structure. For reinforcement learning projects, especially in finance, it is very common to use predefined environments made available by OpenAI, called gym environments. These environments are designed to facilitate the development, testing, and benchmarking of RL algorithms. Anyway for this thesis project it was chosen not to rely on one of these environments made available, but to create one from scratch. This allows for much more freedom of movement and the ability to tailor each detail to the specific project. This was done especially with a view to a possible continuation of this thesis in the field of scientific research.

In our implementation, the environment is represented by the class `trade_env`. Let's analyze his components:

Initialization: Upon instantiation, the environment is initialized with parameters such as the financial dataset (`env`), initial portfolio worth (`worth`) and the length of each trading cycle (`cycle`). The action space is determined by the number of stocks available plus one to take in account the possibility to holding cash in the portfolio. Similarly, the state space is calculated based on the stock information contained in the dataset, excluding irrelevant information such as the stock's ticker and the date.

Reset Function: The reset function initializes the environment at the beginning of each episode. It sets the initial portfolio worth to 20.000,00 $, selects a random starting date for trading, and retrieves initial asset information.

[4] Each stock belonging to the portfolio

Step Function: The step function executes a single step within the environment. It takes an action (investment distribution across assets) as input and returns the resulting state, reward. Key steps include normalizing actions (to be sure the sum of the action element is equal to 1), updating portfolio values based on asset prices and investment decisions, calculating rewards, and determining if the episode has terminated.

4.3.4 The Agent

As already mentioned, the agent is the entity that interacts with the environment in order to maximize the cumulative reward. The agent is the learning component of the RL framework, responsible for making decisions and taking actions based on its observations of the environment and its internal state.

The agent in the context of our asset optimization framework is responsible for making decisions on portfolio allocations based on the information received from the environment, we can say that it will play the role of the investor or better of the software portfolio manager, and his goal will be to obtain the highest possible return from his portfolio of assets over a fixed time period.

The agent that this thesis work proposes to solve the asset allocation problem is a DDPG agent, namely, an agent that follows the DDPG framework described in subsection subsection 3.5. This choice was made due to its ability to handle continuous action spaces and demonstrated performance in similar domains.

We now elucidate the implementation and workings of the Deep Deterministic Policy Gradient (DDPG) agent, which we remember utilizes actor-critic architecture to learn optimal policies.
The agent's functionality are encapsulated in the `DDPG` class.

Initialization: the DDPG agent class initializes its components, including a replay buffer, actor and critic networks, target networks, and an Adam optimizers with learning rate equal to 0.003.

Action Selection: The `select_action` method takes the current state as input and returns an action based on the actor network's output. It employs the actor network to map states to actions.

Networks Update Mechanism: The `update` method orchestrates the training process of the actor and critic networks. It samples 64 batches from the replay buffer, computes target Q-values using the target networks, and optimizes the actor and critic networks using gradient ascent and descent, respectively. When calculating the target Q-values, the critic target network utilises a parameter $\gamma = 0.99$ in this way the agent takes future rewards more strongly into consideration, becoming more farsighted.

Soft updates (31) are employed to periodically update the target networks with parameters from the main networks, using a parameter $\tau = 0.001$.

Model Saving: Eventually, the `save_model` method enables the agent to save its trained actor and critic network weights to disk for future use or analysis.

4.3.5 Actor and Critic Networks

Both the Actor and the Critic are function approximations to learn the policy function and the Q-function, for this reason, we can use neural networks to construct them. The Actor takes action given the states and the Critic evaluates the action taken by the Actor.

We need to keep in mind that the DDPG is an off-policy method that uses target networks to stabilize the learning process, anyway both the target critic network and the target actor network will have the same architecture of the respective "non-target" counterparty. Therefore we will need just 2 classes for the networks; `Critic` and `Actor`.

For the Critic, we have chosen a fully connected neural network with two hidden layers with layer normalization and ReLU activation functions. The output layer is also a fully connected layer with ReLu activation function which returns a single scalar value representing the Q-value, i.e. the expected total reward for the current state-action pair.

The input passed to the network corresponds to a flattened matrix of dimensions (num of stock in the portfolio, num of variables per each stock) = (5, 16) corresponding to the state s_t. The hidden size (number of nodes) of the first and second layers is 64 respectively, with the addition of concatenating the actions with the output of the first hidden layer before processing through the second hidden layer.

For the Actor we use a fully connected neural network with two hidden layers with layer normalization and ReLU activation functions and an output layer with a softmax activation to ensure a valid probability distribution.

The input corresponds to the same passed to the critic corresponding to the state s_t. The hidden size of the first and second layers is 64 respectively, followed by the final linear layer which maps the output of the second linear layer to an output space of dimension `action_dim` (The portfolio vector w_t).

4.3.6 The Replay Buffer

The replay buffer plays a pivotal role in enabling the agent to learn from past experiences and mitigate issues associated with the correlation of consecutive samples. We elucidate the functionality and implementation of the replay buffer utilized in our reinforcement learning framework.

Initialization: The replay buffer is initialized with a maximum capacity (`max_size = 1.000.000`) to store transition tuples comprising state, next state, action, reward, and episode termination flag. The circular nature of the buffer ensures that old memories are replaced when the buffer reaches its maximum capacity.

Push Operation: The `push` method appends new transition tuples to the buffer. When the buffer is at full capacity, older memories are overwritten,

maintaining a fixed memory size.

Sampling: The `sample` method allows the retrieval of a batch of experiences from the buffer for training purposes. Random sampling ensures that experiences are drawn uniformly from the buffer, reducing the potential for biases in the learning process.

4.3.7 The Reward

The reward function plays a crucial role in reinforcement learning, guiding the agent's learning process by providing feedback on its actions.

In this work, the reward at each time step t is computed as the difference between the absolute return of the portfolio and the return of the same amount of money all invested in the S&P 500 index. Mathematically, the reward r_t at time t is defined as:

$$r_t = (V_t - V_{t-1}) - (SP\ Portfolio_t - SP\ Portfolio_{t-1})$$

Where $SP\ Portfolio_t$ is the value of a portfolio fully invested in the S&P 500 index.

This reward function captures the performance differential between the agent's portfolio and the benchmark represented by the S&P 500 index. A positive reward indicates that the portfolio outperformed the benchmark, while a negative reward suggests underperformance relative to the benchmark. By maximizing cumulative rewards over time, the agent aims to optimize its asset allocation strategy to achieve superior returns compared to the market index. By incorporating market dynamics into the reward signal, the agent learns to adapt its strategy to varying market conditions and exploit opportunities for value generation.

4.4 Training Process

The training process orchestrates the iterative learning of the agent within the reinforcement learning framework. This involves updating its decision-making strategy based on feedback received from the interactions with the environment. In this section, we explore how the agent learns and improves its policy for managing asset portfolios observing how the DDPG algorithm is implemented in practice.

The first steps before starting the actual training process consist of the instantiation of the trading environment object (`trade_env` , which is an instance of the `env` class) and the DDPG agent (which is an instance of the `DDPG` class).

Once the Agent and the Environment are initialised the agent start iteratively interacting with the environment across multiple training episodes. The duration of each episode was set to 7 days, which corresponds to, 5 days of trading.

Within each episode, the agent uses the `select_action` method to select the best action given the current state and the current exploration policy.

To balance exploration and exploitation, the agent incorporates Gaussian noise[5] into its selected actions, promoting exploration of the action space while leveraging learned policies to exploit promising strategies. This exploration strategy enhances the agent's ability to discover optimal asset allocation policies and adapt to changing market conditions.

The perturbed action is then passed to the `step` method of the environment, which based on the selected action returns the resultant reward and next state.

During each episode, the agent accumulates rewards and experiences, storing them in the replay buffer for subsequent learning. After the predefined number of trading days or when some termination conditions are met [6] the episode ends and the agent updates the network weights using a batch of samples from the replay buffer. It is worth remembering that while the Actor and the Citric networks' weights are normally updated [7], the target networks are updated via **soft updates** 31. The soft updates use a $\tau = 0,001$.

Throughout the whole training process, metrics such as episode returns, policy loss, and value loss are monitored to gauge the agent's performance and convergence.

The training process continues until a specified number of episodes are completed. The results shown in this chapter have been obtained by training the agent over 1900 episodes.

4.5 Training Performances

The training phase of the reinforcement learning framework yielded valuable results. In this section, we present key results and visualizations to elucidate the training process and outcomes.

4.5.1 Monitoring Value Loss and Policy Loss

A very important step while training a DDPG is to look at the losses; the value loss and policy loss play crucial roles in guiding the learning process of the agent. Monitoring the value and policy loss provides insights into the progress of the training process. A decreasing trend in both losses indicates that the agent is effectively learning from experiences and improving its decision-making strategy. Ensuring stability and convergence of value loss and policy loss is essential for the successful training of the agent. Unstable or diverging loss trends

[5] The authors of the original DDPG paper recommended time-correlated OU noise, but more recent results suggest that uncorrelated, mean-zero Gaussian noise works perfectly well. Since the latter is simpler, it is preferred [30].

[6] The agent loses more than 10% of initial portfolio value or the trading day is equal to the last day in the training dataset

[7] Using gradient ascent/descent method

may indicate issues with the model architecture, hyperparameters, or training methodology that need to be addressed.

Value Loss

Recall that the value loss measures the discrepancy between the predicted value of state-action pairs by the critic network and the actual observed returns. It is important as it helps the agent to learn the expected long-term rewards associated with taking specific actions in different states. By minimizing the value loss, the critic network learns to accurately estimate the Q-values, which are used by the actor network to make informed decisions.

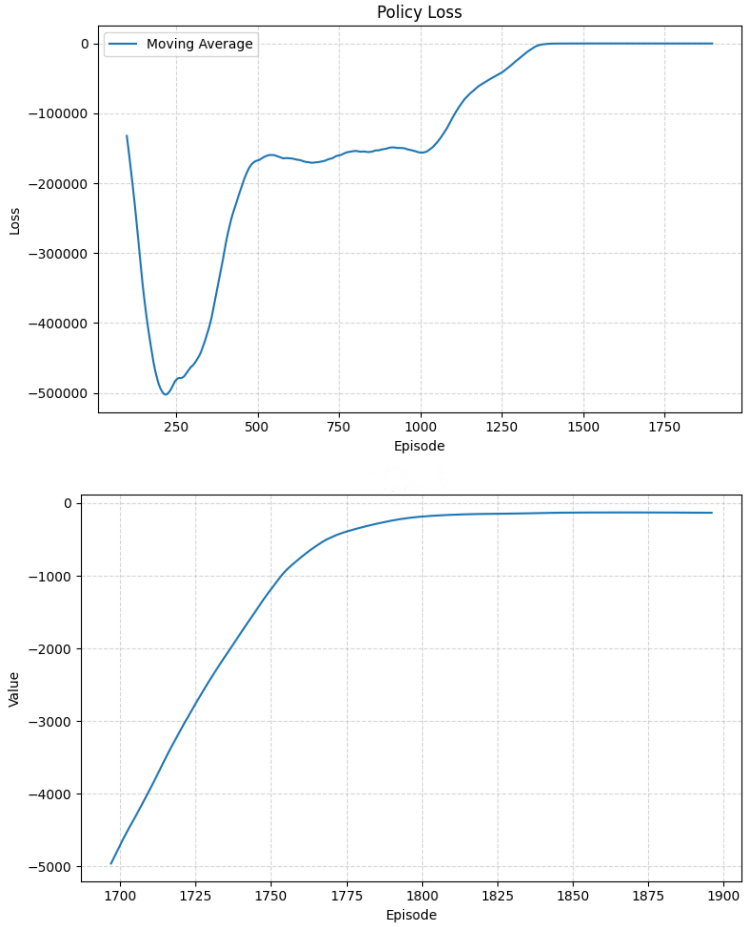


Figure 8: Policy Loss Moving Avarage

The above figures show the evolution of the Policy Loss through the training

process, with the second plot focusing on the last episodes, this is because the scale of the y-axis in the first plot is very wide, and it wouldn't be possible to observe the true convergence of the loss. The same is done also for the Value Loss.

Policy Loss

The Policy Loss quantifies the difference between the actions suggested by the actor network and the actions that maximize the expected long-term rewards estimated by the critic network.

By minimizing the policy loss, the actor network learns to improve its decision-making strategy and generate actions that lead to higher rewards in the long run.

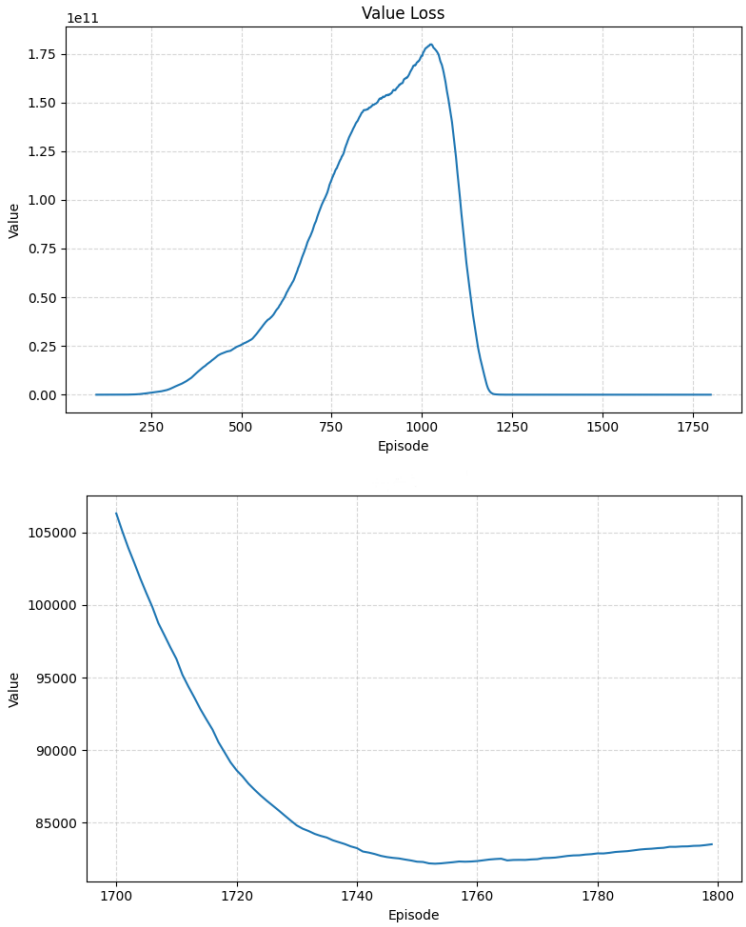


Figure 9: Value Loss Moving Avarage

The plot illustrates the training progress of the DDPG algorithm by showing the Policy Loss and Value Loss over episodes. Initially, both losses exhibit an increasing trend, indicating the early stages of learning where the agent's policy and value estimations are unstable. As training progresses, both losses begin to converge, indicating that the agent is learning and improving its performance. Notably, the Policy Loss converges to zero, suggesting that the agent's policy becomes increasingly optimal over time. However, the Value Loss converges to higher values, indicating that the agent's value estimations may not be perfectly accurate but remain stable. Overall, the convergence of both losses signifies the successful training of the agent, albeit with some remaining uncertainty in value estimations.

As we can see in figure 9 and 8 both policy and value loss exhibit a clear trend of convergence over the training episodes, indicating the progressive refinement of the agent's decision-making strategy and value estimation accuracy. Initially, both losses exhibit an increasing trend, indicating the early stages of learning where the agent's policy and value estimations are unstable. As training progresses, both losses begin to converge, indicating that the agent is learning and improving its performance. Notably, the Policy Loss converges to zero, suggesting that the agent's policy becomes increasingly optimal over time. However, the Value Loss converges to higher values, indicating that the agent's value estimations may not be perfectly accurate but remain stable. Overall, the convergence of both losses signifies the successful training of the agent, albeit with some remaining uncertainty in value estimations.

Notice how in 9 and 8 the x-axis doesn't start from zero but rather from 100, this is because the plotted function represents the moving average (with a window of 100 values) of the original losses. The same goes for the graph in the figure 20.

4.6 Episode Return

The episode returns provide a comprehensive view of the agent's performance over training episodes, reflecting the cumulative rewards obtained during each episode.

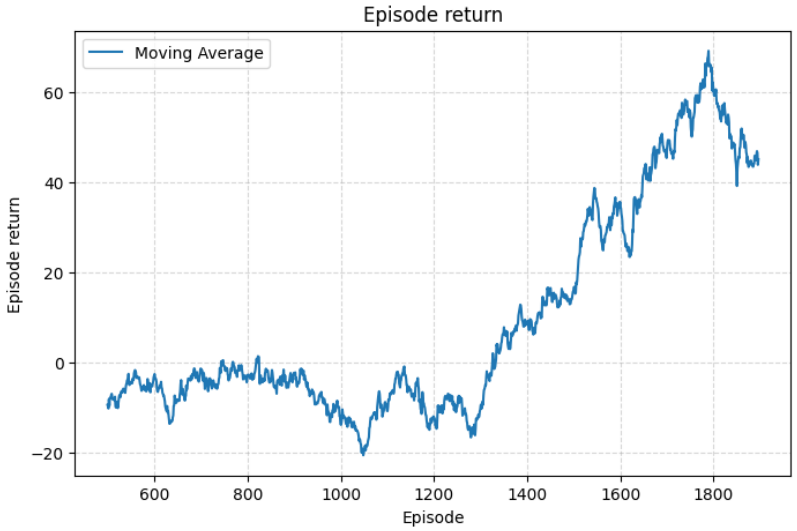


Figure 10: Moving Average of the returns for each training episode

The graph above illustrates the trend of episode returns over the training duration, highlighting the agent's ability to achieve favourable outcomes over time. The figure demonstrates a positive trend in episode returns, indicating that the agent is consistently improving its decision-making strategy and generating higher returns as training progresses. The increasing episode returns validate the effectiveness of the reinforcement learning approach in optimizing asset allocation and maximizing portfolio performance.

4.7 Testing the agent

After completing the training phase of the reinforcement learning agent, the next crucial step is to evaluate the performance of the trained agent on unseen data. In this chapter we expose the results obtained from testing the agent's decision-making capabilities for asset portfolio management over different periods of time.

The methodologies followed by the agent is the same as described in the training chapter. Given a random starting date, the agent is provided with the current state s_t containing the daily information of the assets in the portfolio, it will then be up to the agent to choose what is the best allocation to maximize

the portfolio return.

The agent's ability to optimize the returns of the portfolio under management was tested over 3 different time periods: 1 month, 6 months, and 1 year. To evaluate the performance of the agent on the test dataset, we compare the return of the managed portfolio with the benchmark return, calculating the difference between the returns at the end of the training period, allowing us to understand whether the active management of the securities has outperformed the benchmark or not.

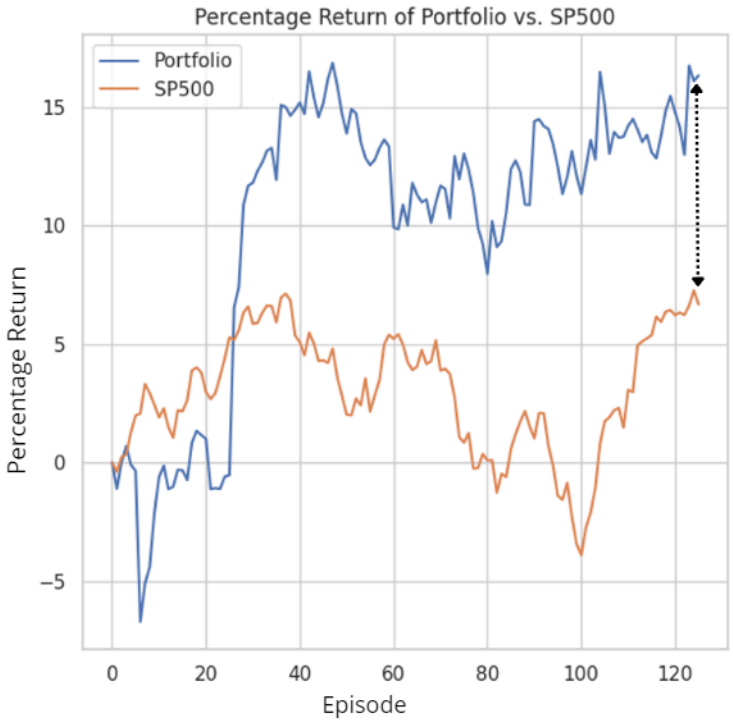


Figure 11: Portfolio vs SP500 returns over 120 trading days

Each trading period was subjected to testing through 100 independent runs to assess the robustness and consistency of the agent's performance[8]. The results were analyzed to gain insights into the agent's ability to optimize asset portfolios in real-world trading scenarios.

For each of the 3 trading period cases, the agent was tested on 100 episodes. For each of the three periods, different histograms are analysed, the first showing the returns obtained by the SP500, this will allow us to evaluate the average

[8]Ideally a much higher number of independent run would be suggested, but with just 2.5 years of test data, a bigger number of runs would most likely cause the agent to encounter the same testing period many times

trend of the benchmark. The second shows the portfolio returns, which will allow us to evaluate how our agent performed overall, and the last displays the difference between these SP500 returns and the portfolio return at the end of each trading period, allowing us to evaluate whether on average, in the same trading days, our agent overperformed or underperformed the benchmark.

Here are the results:

1 MONTH

The performance assessment for the 1-month trading period provides valuable insights into the agent's ability to manage the portfolio effectively within a short timeframe.

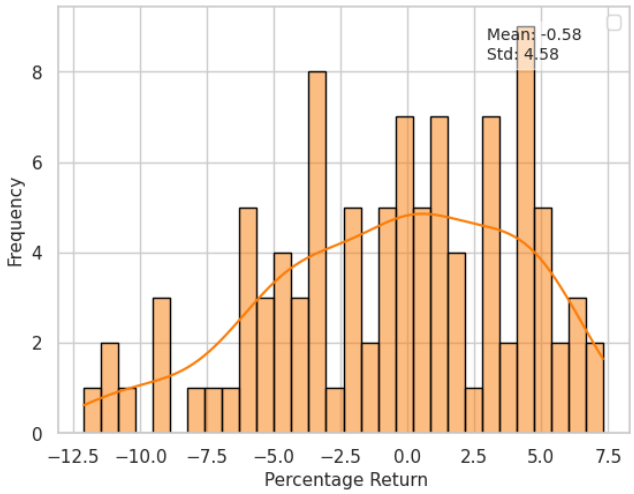


Figure 12: Distribution of S&P percentage return over 1 month period

The histogram above represents how in the period between the end of 2021 and beginning of 2024, for 1-month trading period, the S&P500 achieved returns of -0.58% on average, with a standard deviation of of 4.58%.

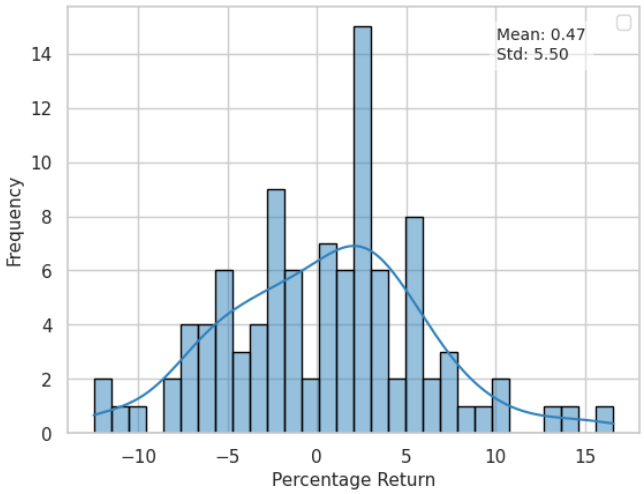


Figure 13: Distribution of Agent's portfolio percentage return over 1 months period

In contrast, the histogram illustrating the returns of the managed portfolio exhibits a slightly positive mean return of 0.47% with a higher standard deviation of 5.50%, suggesting that on average the agent was able to do better than the benchmark but taking more risks.

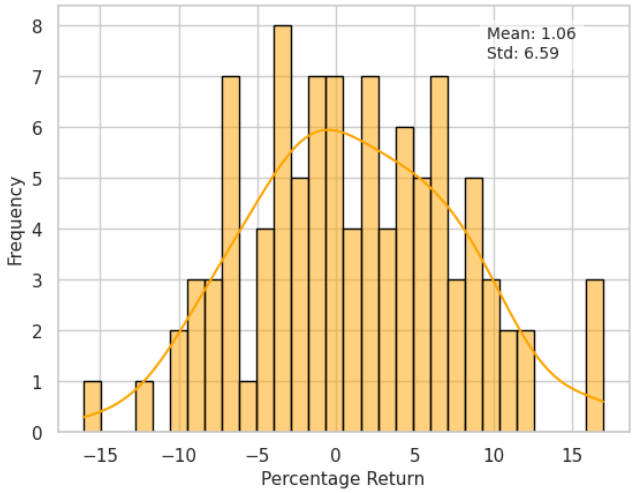


Figure 14: Distribution of percentage difference between Agent's portfolio returns and S&P returns over 1 month period

The final histogram, depicting the difference between the portfolio returns

and the benchmark returns, offers valuable insights into the agent's performance relative to the market index. The positive mean of the distribution suggests how on average the portfolio actively manged by the agent overperform by 1.06% the benchmark.

In conclusion, during the one-month trading period, the DDPG agent exhibited promising performance, outperforming the benchmark portfolio. The histogram depicting the distribution of results revealed a higher frequency of higher returns for the agent's portfolio compared to the benchmark. This indicates that the agent was able to capitalize on short-term market trends and make timely investment decisions, resulting in superior portfolio returns.

6 MONTHS

The assessment of the agent's performance over a 6-month trading period offers valuable insights into its ability to navigate mid-term market dynamics and capitalize on emerging opportunities.

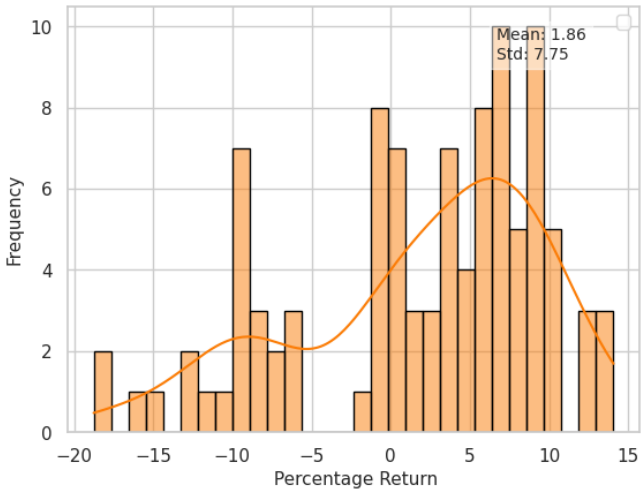


Figure 15: S&P percentage return over 6 months period

The histogram representing the returns of the SP500 index over 6-month periods indicates a moderate mean return of 1.86% with a standard deviation of 7.75%. The distribution suggests, on average, a relatively stable yet slightly bullish market environment over 6-month periods, characterized by moderate volatility.

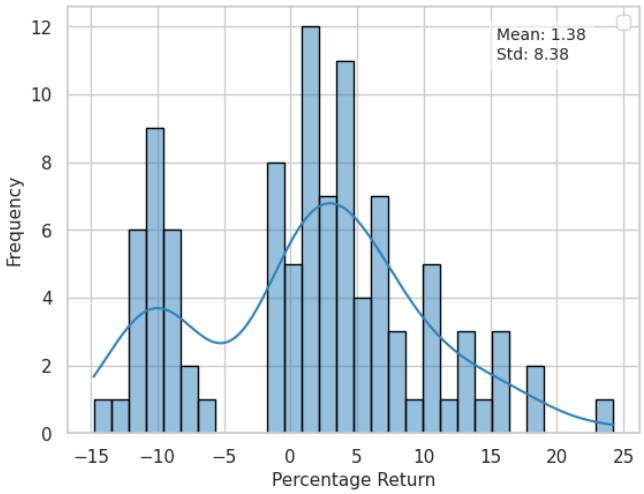


Figure 16: Agent's portfolio percentage return over 6 months period

In comparison, the histogram illustrating the returns of the managed portfolio exhibits a marginally lower mean return of 1.38% and a slightly higher standard deviation of 8.30%. The higher standard deviation suggests a more aggressive trading strategy employed by the agent to capitalize on medium-term market trends and fluctuations.

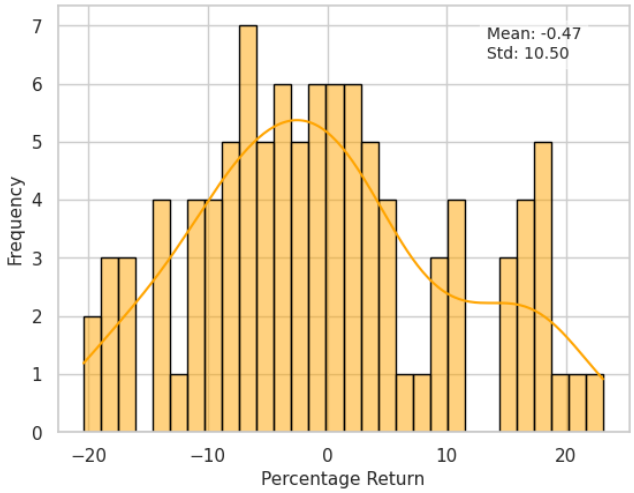


Figure 17: Percentage difference between Agent's portfolio returns and S&P returns over 6 months period

In contrast to the one-month trading period, the performance of the DDPG

agent during the six-month period showed mixed results. While the agent demonstrated competitive performance in some runs, on average the benchmark outperformed the agent in terms of overall returns.

1 YEAR

The evaluation of the agent's performance over a 1-year trading periods provides valuable insights into its ability to navigate longer-term market dynamics and sustain profitability over extended durations.

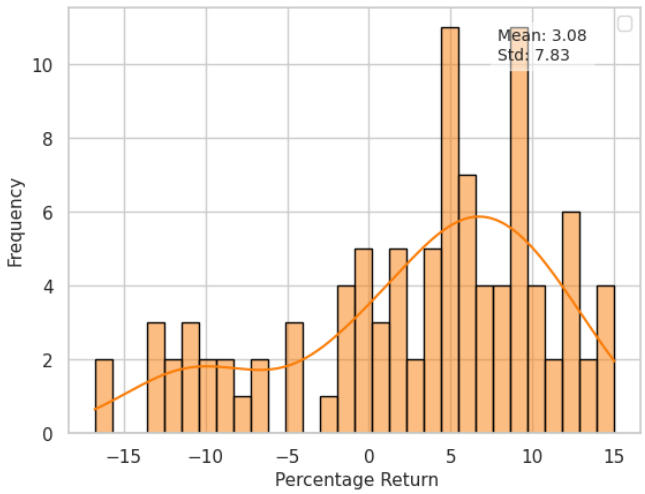


Figure 18: S&P percentage return over 1 year period

The histogram depicting the returns of the S&P 500 index during the 1-year period reveals a mean return of 3.08%, this means that in the last 3 years, on average, for long periods the SP500 achieved positive returns, with a moderate standard deviation equal to 7.83%.

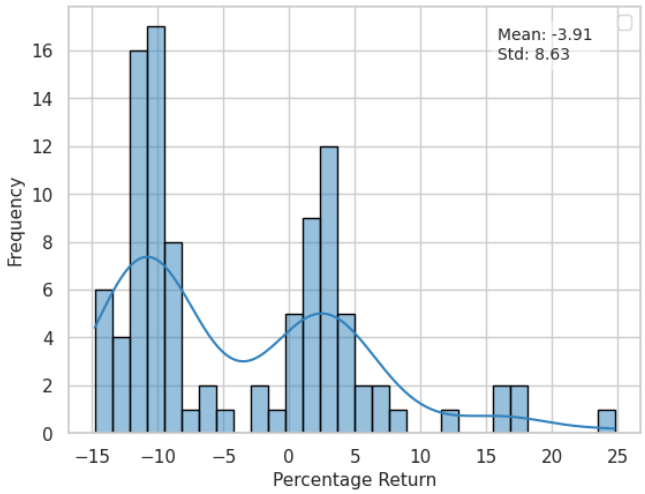


Figure 19: Agent's portfolio percentage return over 1 year period

In contrast, the histogram illustrating the returns of the managed portfolio exhibits a notably lower mean return of -3.91% and a comparable standard deviation of 8.36%. The distribution indicates a divergence from the benchmark index, with the agent struggling to deal with longer periods of time.

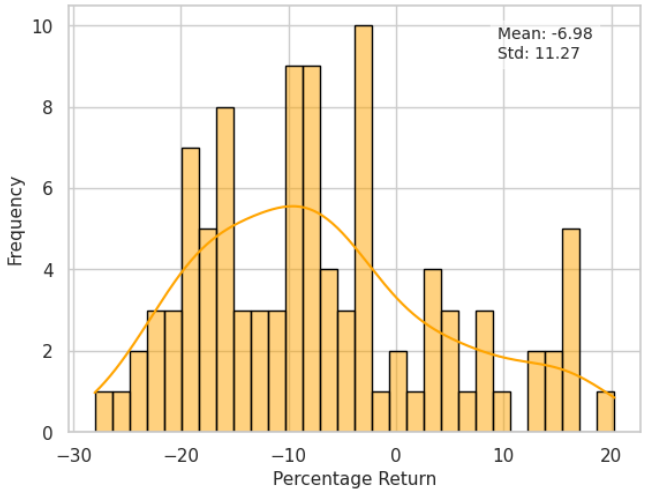


Figure 20: Percentage difference between Agent's portfolio returns and S&P returns over 1 year period

The last histogram confirms how extending the trading period to one year revealed further challenges for the DDPG agent. In this scenario, the man-

aged portfolio consistently underperformed the benchmark, achieving on average much lower returns. Factors such as market fluctuations, economic uncertainties, and unexpected events may have contributed to the divergence in performance between the portfolio and the market index.

Despite its ability to capture short-term trends, the agent appeared to face difficulties in sustaining profitability over extended investment horizons. The analysis of performance across different trading periods underscores the importance of evaluating the agent's robustness and adaptability in diverse market conditions. While the agent demonstrated promising results in shorter trading periods, its performance deteriorated in longer-term scenarios, where the benchmark portfolio proved to be better. The wider distribution of returns and higher volatility observed in the agent's portfolio suggest the need for further refinement and optimization of the reinforcement learning model to enhance its long-term viability and stability in real-world investment environments.

4.8 Assumptions and Limitations

4.8.1 Assumptions

In developing our model for asset allocation using Deep Deterministic Policy Gradient (DDPG), we make several fundamental assumptions to establish the framework within which the agent operates. These assumptions serve as foundational pillars that shape the dynamics of the model and guide its behaviour in managing the portfolio effectively. Let's have a closer look into each of these assumptions to gain a deeper understanding of their significance in our context.

- **Assumption 1:** The state S respects the Markov property, i.e., we are assuming that the state s provides the agent with all the necessary information to take the best asset allocation action.
- **Assumption 2:** Zero slippage, the difference between the expected price of a trade and the price at which is executed is equal to zero
- **Assumption 3**: The market is not influenced by the action taken by the agent
- **Assumption 4:**the assets in the portfolio always remain the same, new assets cannot be added during the portfolio management period
- **Assumption 5:**The volume of each stock is large enough, enabling the model to execute buy or sell orders for any stock on any given trading day.

4.8.2 Limitations

Despite the promising capabilities of the developed model, several limitations must be acknowledged. Firstly, the model's development was constrained by access to only freely available data. While this data sufficed

for initial testing and exploration, algorithmic trading and portfolio management typically rely on a broader range of indicators to construct more sophisticated models. Incorporating additional data sources and indicators could potentially enhance the model's predictive power and robustness. Secondly, a fundamental assumption underlying the model is that the state representation respects the Markov property. This assumption is quite stringent and implies that the state representation captures all relevant information necessary for decision-making. However, in reality, market dynamics are influenced by a myriad of complex factors, including market sentiment, macroeconomic indicators, and geopolitical events. Neglecting these factors may oversimplify the model and limit its effectiveness in capturing the true complexities of financial markets. Moreover, the performance of the model and its results may be sensitive to the selection of assets included in the portfolio. The model's efficacy could vary depending on the chosen assets, their historical performance, and their correlations with one another. Finally, an important assumption of the model is the neglect of transaction costs. While this simplification facilitates model development and analysis, it may lead to overestimated profitability and unrealistic trading frequency. In practice, transaction costs can significantly impact the performance of trading strategies, affecting both profitability and risk management. Recent research[44] has shown that incorporating transaction costs into reinforcement learning frameworks, such as DDPG, can lead to more realistic and effective trading strategies. However, addressing this limitation requires careful calibration and validation to ensure the model's stability and performance.

5 Conclusione and Future Work

5.1 Conclusion

The application of Deep Deterministic Policy Gradient (DDPG) reinforcement learning algorithm for asset portfolio management has been a fascinating journey, characterized by exploration, experimentation, and discovery. Through the course of this thesis work, we have delved into the intricacies of training a DDPG agent to optimize asset allocations in dynamic and uncertain financial markets. Our exploration has yielded valuable insights and raised important questions regarding the efficacy and applicability of reinforcement learning techniques in real-world investment scenarios.

Our investigation began with a comprehensive review of Reinforcment Learning theoretical and its underlying theory and fundamental concept, laying the groundwork for eventually understanding complex algorithms such as Deep Q-Learning a Deep Deterministic Policy Gradient. We then proceeded to implement and train the DDPG agent using historical market data, leveraging its ability to learn optimal investment strategies through trial and error.

The training phase provided invaluable insights into the behavior of the DDPG agent, as we observed the convergence of policy and value loss metrics, indicating the refinement and optimization of the agent's decision-making capabilities over time. However, we also encountered challenges and limitations, such as the impact of transaction costs and the need for further research into model robustness and adaptability.

The field of our research has significant implications for both academia and industry. The successful implementation of the DDPG algorithm demonstrates the potential of reinforcement learning techniques in optimizing asset portfolio management strategies. However, further research is needed to address key challenges and limitations.

However, in our research, we acknowledge that the Markov property assumption poses a significant challenge, particularly in the context of asset portfolio management. Financial markets are inherently dynamic and influenced by a multitude of factors, including economic indicators, geopolitical events, and investor sentiment. As a result, the state space of our model may not fully capture the complex interactions and dependencies present in the underlying market dynamics.

The failure to fully satisfy the Markov property assumption could have contributed to the suboptimal performance of our DDPG agent. In situations where past events and external factors exert a significant influence on future market conditions, the agent's ability to make optimal decisions based solely on the current state may be limited. This highlights the importance of further research into more sophisticated modelling techniques that can capture the non-Markovian nature of financial markets.

In conclusion, the journey of exploring the application of DDPG in asset portfolio management has been enlightening and thought-provoking. While our work has provided valuable insights, there is still much to explore and discover in this exciting field. By continuing to push the boundaries of innovation and research, we can unlock new opportunities and revolutionize the way we approach investment management in the digital age.

5.2 Future Work

While this master thesis has provided valuable insights to the field of reinforcement learning-based portfolio optimization, there are infinite avenues that can be explored starting from the foundation provided by this work. These potential areas of investigation include:

Transaction Costs Integration: For sure one notable limitation of the current study is the absence of transaction costs consideration. Future work could focus on incorporating realistic transaction costs models into the reinforcement learning framework. This would provide a more accurate representation of real-world trading scenarios and improve the practical applicability of the developed algorithms.

Dynamic Environment Modeling:In the current study, the states only include stock data. However, financial markets are subject to various external factors and events. One potential avenue for future research is the introduction of macro variables and other dynamic factors into the state representation.

By incorporating macroeconomic indicators, market sentiment analysis, geopolitical events, and other relevant factors into the state space, the agent can gain a more comprehensive understanding of the underlying market dynamics. This enhanced state representation enables the agent to adapt its decision-making process in response to changing market conditions and trends.

Considering different risk aversion : Asset optimization often involves multiple conflicting objectives, such as maximizing returns while minimizing risk or maintaining portfolio diversification. Future work could focus on developing multi-objective reinforcement learning frameworks capable of simultaneously optimizing across multiple objectives, providing more comprehensive and robust asset management strategies.

Explore different reward options: Future research should try to apply a different reward system. One possibility could be to use a baseline portfolio as a benchmark, where with *baseline portfolio* is mean a portfolio where we allocate the same amount of money to each stock. i.e $w = \frac{1}{5}[1, 1, 1, 1, 1]$ and we leave the weights unchanged for the whole trading period. This, in addition to exploring a new type of reward, would make this approach more general and applicable to any stock, no longer having to use the S&P 500 as a benchmark.

Exploration of Alternative Algorithms: While the DDPG algorithm has shown promising results in this field, the number of possible reinforcement learning algorithms is huge and in continuous development, exploring alternative reinforcement learning algorithms could offer valuable insights and potentially

superior performance. Algorithms such as Twin Delayed DDPG (TD3), Soft Actor-Critic (SAC), Proximal Policy Optimization (PPO) or ANormalized Advantage Functions (NAF) could be considered for comparison and evaluation.

By addressing these research directions, future studies can further advance the field of reinforcement learning-based asset optimization and contribute to the development of more sophisticated and effective financial decision-making tools.

References

[1] Deep deterministic policy gradient (ddpg) agents.

[2] Wentong Bao and Yang Liu. Multi-agent reinforcement learning in algorithmic trading. *arXiv preprint arXiv:1912.01597*, 2019.

[3] Eric Benhamou, David Saltiel, Sandrine Ungari, and Abhishek Mukhopadhyay. Bridging the gap between markowitz planning and deep reinforcement learning. *arXiv preprint arXiv:2010.09108*, 2020.

[4] Alvaro Cartea, Sebastian Jaimungal, and Jose Penalva. Reinforcement learning for automated trading in financial markets. *Annual Review of Financial Economics*, 12:361–383, 2020.

[5] Arkopal Chakraborty and et al. Robust multi-objective portfolio optimization using deep reinforcement learning. *arXiv preprint arXiv:1910.01578*, 2019.

[6] Vijay K. Chopra and William T. Ziemba. The effect of errors in means, variances, and covariances on optimal portfolio choice. *Journal of Portfolio Management*, 19(2):6–11, 1993.

[7] Yves Choueifaty and Lionel Coignard. Toward maximum diversification. *The Journal of Portfolio Management*, 35(1):40–51, 2008.

[8] Yves Choueifaty, Thomas Froidure, and Laurent Reynier. Properties of the most diversified portfolio. *Journal of Investment Strategies*, 1(3):49–70, 2012.

[9] Peter Christoffersen, Vihang Errunza, Kris Jacobs, and Hélène Langlois. Is the potential for international diversification disappearing? a dynamic copula approach. *Review of Financial Studies*, 23(4):1593–1621, 2010.

[10] Yuxin Deng and et al. Deep direct reinforcement learning for financial signal representation and trading. *IEEE Transactions on Neural Networks and Learning Systems*, 28(3):653–664, 2016.

[11] Ricard Durall. Asset Allocation: From Markowitz to Deep Reinforcement Learning. Papers 2208.07158, arXiv.org, July 2022.

[12] Hugging Face. The deep q-learning algorithm, n.d. Accessed on: 2024-03-09.

[13] Ziming Gao, Yuan Gao, Yi Hu, Zhengyong Jiang, and Jionglong Su. Application of deep q-network in portfolio management. In *2020 5th IEEE International Conference on Big Data Analytics (ICBDA)*, pages 268–275, 2020.

[14] Shixiang Gu and et al. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. *arXiv preprint arXiv:1610.00633*, 2017.

[15] Zhengyao Jiang and Jinjun Liang. Cryptocurrency portfolio management with deep reinforcement learning. In *2017 Intelligent Systems Conference (IntelliSys)*, pages 905–913, 2017.

[16] Zhengyao Jiang, Dixing Xu, Yifei Liang, and Steven Li. A deep reinforcement learning framework for the financial portfolio management problem. *IEEE Transactions on Neural Networks and Learning Systems*, 29(9):4257–4273, 2017.

[17] Zhongxing Jin, Yingyu Liang, Wei Cheng, Yiran Wang, and Jie Zhang. Deep portfolio optimization via distributional prediction. In *International Conference on Machine Learning*, pages 3149–3158. PMLR, 2019.

[18] Petter N. Kolm and Gordon Ritter. Algorithmic decision-making in financial markets: The time is now. *SSRN*, 2019.

[19] Sergey Levine and et al. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[20] Bo Li, Steven CH Hoi, and Vivek Gopalkrishnan. Portfolio management with reinforcement learning. *ACM Computing Surveys (CSUR)*, 51(2):1–36, 2018.

[21] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[22] Xiao-Yang Liu and et al. Deep reinforcement learning for portfolio management: An implementation perspective. *arXiv preprint arXiv:2005.11233*, 2020.

[23] Zhewei Lyu, Bo Li, Vishnu Sridharan, and Steven CH Hoi. Optimal trading with alpha predictors. *arXiv preprint arXiv:2102.06179*, 2021.

[24] S. Maillard, T. Roncalli, and J. Teiletche. The properties of equally weighted risk contribution portfolios. *Journal of Portfolio Management*, 36(4):60–70, 2010.

[25] Harry Markowitz. Portfolio Selection. *Journal of Finance*, 7(1):77–91, March 1952.

[26] Matlab. What is portfolio optimization?, n.d.

[27] Volodymyr Mnih, David Silver, and et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[28] Yingying Ning, Qiaomin Lin, and Sebastian Jaimungal. Deep reinforcement learning in portfolio management. *arXiv preprint arXiv:1811.04546*, 2018.

[29] Damian Ong, Faraz A Zohaib, Ruixun Goh, and Steven Li. Deep reinforcement learning in portfolio management. *arXiv preprint arXiv:1912.10930*, 2019.

[30] OpenAI. Deep deterministic policy gradient (ddpg), n.d. Accessed on: 2024-03-09.

[31] T. Roncalli and G. Weisang. Risk parity portfolios with risk factors. *Quantitative Finance*, 16(6):837–854, 2016.

[32] John Schulman and et al. Trust region policy optimization. *arXiv preprint arXiv:1502.05477*, 2015.

[33] David Silver and et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[34] David Silver and et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[35] David Silver, Guy Lever, Nicolas Manfred Otto Heess, Thomas Degris, Daan Wierstra, and Martin A. Riedmiller. Deterministic policy gradient algorithms. In *International Conference on Machine Learning*, 2014.

[36] Richard S Sutton and Andrew G Barto. *"Policy Gradient Methods, Chap. 13, "Reinforcement learning: An introduction.* MIT press, 2018.

[37] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[38] Josh Thomas, Dhruv Patel, Michael Booth, Marimuthu Palaniswami, and Hao Zhu. A reinforcement learning approach to portfolio optimization. In *Proceedings of the 34th International Florida Artificial Intelligence Research Society Conference*, pages 78–83, 2021.

[39] Oriol Vinyals and et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[40] X. Wang, W. Jia, and Q. Weng. Automatic lane-keeping system based on deep reinforcement learning. *Electronics Letters*, 54(2):110–112, 2018.

[41] C. Xiong and et al. Reinforcement learning-based financial portfolio management with augmented asset movement prediction. *IEEE Access*, 7:122506–122515, 2019.

[42] Yue Yu, Zhaoxi Liang, Zhengyao Jiang, and et al. Deep reinforcement learning for automated portfolio management. *Quantitative Finance*, 19(9):1275–1292, 2019.

[43] Hongyang Zhang, Stefan Zohren, and Stephen Roberts. Towards automated order sizing for multi-asset class trading via deep reinforcement learning. *arXiv preprint arXiv:1909.00285*, 2019.

[44] Huanming Zhang, Zhengyong Jiang, and Jionglong Su. A Deep Deterministic Policy Gradient-based Strategy for Stocks Portfolio Management. Papers 2103.11455, arXiv.org, March 2021.

[45] Jiang Zhengyao, Zhaoxi Liang, and et al. Deep reinforcement learning for order execution in algorithmic trading. *arXiv preprint arXiv:1706.10059*, 2017.