POLITECNICO DI TORINO

Master's Degree in Mathematical Engineering

Master's Degree Thesis

Portfolio Management: A Deep Reinforcement Learning Approach

Supervisors

Prof. Paolo BRANDIMARTE

Prof. Edoardo FADDA

Prof. Carlo SGARRA

Candidate

Yi Yu Ivan CHEN

November 2022

Summary

Portfolio management is the task of constantly redistributing the wealth of an investor into different financial products known as assets in order to maximize the overall profit during the trading period while maintaining an acceptable level of risk. Traditionally, this problem has been heavily studied using a static approach, leading to the creation of what is known as the Modern Portfolio Theory by economist H. Markowitz in 1952. However, more recent developments in the field of optimization of dynamical systems, especially via reinforcement learning, have sparked a growing interest in applying these new techniques to the portfolio optimization task.

In this thesis, we represent the trading environment as a discrete-time stochastic dynamical system and apply deep learning based reinforcement learning algorithms to train an agent to learn a trading strategy that optimizes a given objective. In particular, we assume that the agent has information only on the daily open, high, low and close prices and the trading volumes and, using these values, the trading agent has to decide, at the end of every trading day, what portfolio to build for the next day, keeping into account that all transactions are penalized due to the presence of a transaction cost equal to a small percentage of the total transaction value.

Regarding the reinforcement learning algorithms employed, we focused only on actor-critic methods using LSTM-based architectures to represent the value and policy functions. These techniques are further improved via a pre-training step by initializing the networks weights using the optimal weights obtained by training a network with the same structure to forecast portfolio returns.

Multiple experiments done using real market data consisting of a small set of stocks chosen from those included in the S&P500 index show the validity of applying the reinforcement learning framework to the task of portfolio optimization.

Acknowledgements

To everybody that has been with me during this hard and amazing adventure.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent times, there has been a growing interest in the field of artificial intelligence and in its ability to automate a variety of tasks. In particular, many reinforcement learning methods have been applied with success to applications involving decision making in dynamic systems.

1.1 Research Question

In this work, we analyse the dynamic asset allocation problem and how it can be addressed by state of the art reinforcement learning algorithms, with a focus on deep learning approaches that approximate the policy of the decision maker with a neural network. Moreover, the effectiveness of these algorithms will be compared to more standard approaches.

1.2 Thesis structure

Excluding this introductory chapter, the thesis is divided into two parts: the **Theoretical Background** and the **Contribution**. As the name implies, the first part focuses on all the financial and machine learning concepts that are needed to thoroughly understand the main objective of this work, while the second part dives deeper into the mathematical framework used to describe the trading environment, its implementation into Python code and the contribution of this thesis to the task at hand. Both parts are also organized into multiple chapters as follows:

- **Chapter 2 - Portfolio Management** introduces all the financial concepts that are needed for understanding the asset allocation problem;
- **Chapter 3 - Reinforcement Learning** explains the reinforcement learning framework and its main components. It also contains a description of the

neural network architectures used in the experiments and how they can be trained;

- **Chapter 4 - Methodology** introduces the mathematical framework used to define the trading environment and all the assumptions that were made regarding the market;
- **Chapter 5 - Implementation** contains a description of how the trading agent and the environment are implemented in Python;
- **Chapter 6 - Experiments** introduces the datasets used, shows all the experiments done on them and compares the results obtained;
- **Chapter 7 - Conclusion** is the final chapter and contains a brief summary of all the work done and possible future research directions.

Part I

Theoretical Background

Chapter 2

Portfolio Management

In this chapter an overview of the financial concepts that are used to tackle the problem of portfolio optimization will be given, with a focus on their mathematical aspects.

2.1 Assets and Portfolios

The word asset is a very generic term that refer to any kind of resource that holds an economical value, such as cash, company stocks, bonds, etc. In this thesis all experiments are done using only cash as the risk-free asset and stocks as the risky assets, although the results also hold for all other kinds of assets, as no restricting assumptions were made regarding the properties of the assets.

A portfolio is a collection of assets and, assuming that it contains M of them, it can be identified as a vector w_t of length M, where each component refers to how much is invested in a given asset at a given time t:

$$\boldsymbol{w}_t = \Big[w_{1,t}, w_{2,t}, \ldots, w_{M,t}\Big]^\intercal \in \mathbb{R}^M \tag{2.1}$$

The actual meaning of the portfolio weights, however, depends on the formulation used, as they may refer to either the number of shares the portfolio owner has or to the fraction of the total portfolio value the given stocks represent. In this thesis, the second interpretation is used and, as such, a second variable v_t is needed to represent the value of the whole portfolio over time. The complete mathematical representation of a portfolio at time step t is, therefore, given by:

$$\Big(v_t, \boldsymbol{w}_t\Big) \text{ s.t. } \sum_{i=1}^{M} w_{i,t} = 1 \ \forall t \ \text{ and } \ w_{i,t} \geq 0 \ \forall i, t \tag{2.2}$$

where the second constraint that limits the weights to only non negative value is due to the added assumption that the trader cannot make short sales.

Short selling, also known as shorting [8], refers to the process of selling an asset that is not owned, which is done by borrowing the asset from someone else, selling it immediately and, after some time, buying it back and returning it to the lender with interest with the hope that the borrowed asset lost value during this set of transactions so as to generate a profit for the investor. Denying the ability to perform short selling means that portfolio managers can only either buy assets or sell what they currently own. In financial terms, this is known as a long only portfolio.

In this work, this simplifying assumption was made to avoid situations in which the optimal portfolio weights might explode to infinity and also because, due to the asymmetrical effects of shorting in which investors can gain at most 100% of the shorted amount but can lose an unbounded amount of money, it can lead to extremely risky investment strategies.

2.2 Asset prices and returns

Assets are dynamic in nature, as their prices and trade volume change over time with high frequency due to a variety of factors. For this reason they can be modeled as discrete time series where the time-step t used can vary depending on the task at hand. Indeed, some applications, such as day trading and high-frequency trading require intraday data, which means that the asset prices' movements are recorded every hour or even minute, while others focus on a much longer time window and, therefore, only require the daily prices and volumes of the assets.

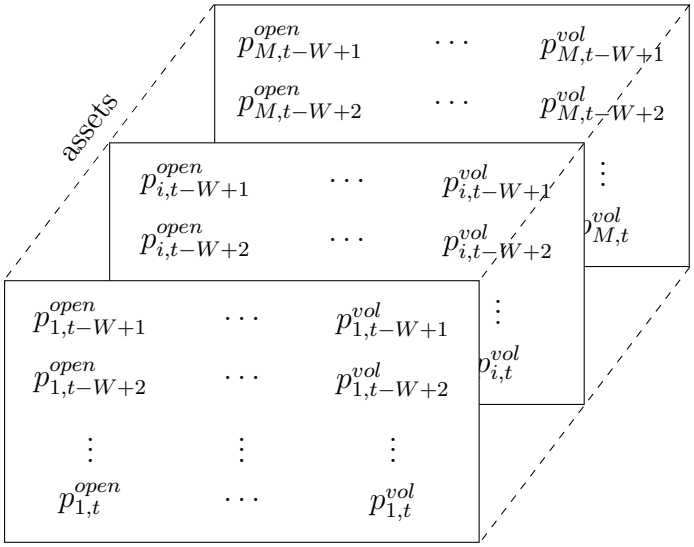
In this thesis, the focus is on the second type of applications and, in particular, uses the daily open, high, low, close and volume (OHLCV) data of all stocks taken into consideration, which, for a generic time-step t, can be denoted as:

$$\vec{\boldsymbol{p}}_t = \left[\boldsymbol{p}_t^{open}\ \boldsymbol{p}_t^{high}\ \boldsymbol{p}_t^{low}\ \boldsymbol{p}_t^{close}\ \boldsymbol{p}_t^{vol}\right] = \begin{bmatrix} p_{1,t}^{open} & p_{1,t}^{high} & p_{1,t}^{low} & p_{1,t}^{close} & p_{1,t}^{vol} \\ p_{2,t}^{open} & p_{2,t}^{high} & p_{2,t}^{low} & p_{2,t}^{close} & p_{2,t}^{vol} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ p_{M,t}^{open} & p_{M,t}^{high} & p_{M,t}^{low} & p_{M,t}^{close} & p_{M,t}^{vol} \end{bmatrix} \in \mathbb{R}_+^{M\times 5} \tag{2.3}$$

Similarly, if the focus is on the past W days of data points for a given asset i, the following two-dimensional matrix can be used:

$$\vec{\boldsymbol{p}}_{i,t-W+1:t} = \begin{bmatrix} p^{open}_{i,t-W+1} & p^{high}_{i,t-W+1} & p^{low}_{i,t-W+1} & p^{close}_{i,t-W+1} & p^{vol}_{i,t-W+1} \\ p^{open}_{i,t-W+2} & p^{high}_{i,t-W+2} & p^{low}_{i,t-W+2} & p^{close}_{i,t-W+2} & p^{vol}_{i,t-W+2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ p^{open}_{i,t} & p^{high}_{i,t} & p^{low}_{i,t} & p^{close}_{i,t} & p^{vol}_{i,t} \end{bmatrix} \in \mathbb{R}_+^{W\times 5} \tag{2.4}$$

which means that, to include all the M assets considered, a three-dimensional matrix $\vec{\boldsymbol{p}}_{t-W+1:t} \in \mathbb{R}_+^{M\times W\times 5}$, which will be referred to as **asset data matrix**, is needed:



(2.5)

Remark The reason why it is important to introduce the notation used for the values observed in a given time window will become clear in the section that will explain the reinforcement learning framework used to solve the problem of portfolio optimization.

To obtain the value of a portfolio using asset prices at time t, assuming that the market has just closed, a simple scalar product is needed:

$$v_t = \sum_{i=1}^{M} w_{i,t} p^{close}_{i,t} = \boldsymbol{w}_t^\intercal \boldsymbol{p}^{close}_t \in \mathbb{R}_+ \tag{2.6}$$

Regardless of the application, however, working directly with prices is usually not recommended, as not only are they not stationary time-series but different assets

can also have a significant difference in their values, which makes comparing them very difficult. Figure 2.1, which shows the daily closing prices of three different stocks, highlights these two issues.

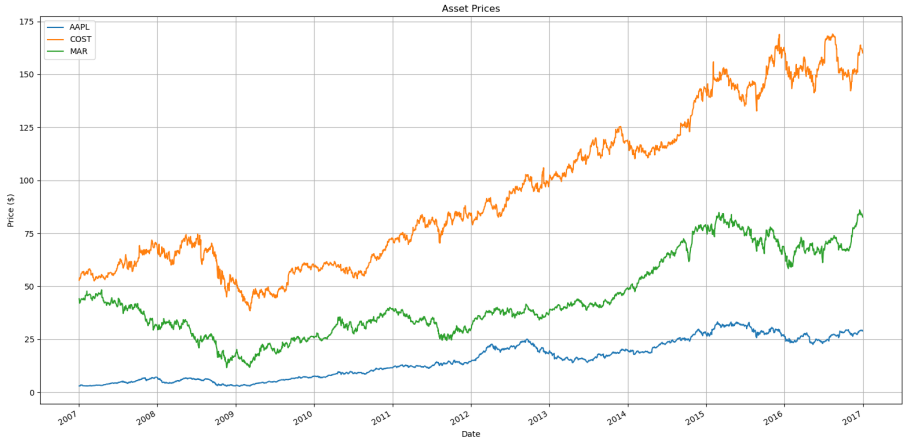


Figure 2.1: Closing stock prices of Apple Inc (AAPL), Costco Wholesale Corporation (COST) and Marriott International Inc (MAR) from start of 2007 to end of 2017

For these reasons, using asset returns, which measure the actual profitability of the portfolio caused by the change in price of its components over time, is the standard approach.

2.2.1 Simple returns

The most common definition of return is the simple return r_t, which represents the percentage change in the price of the asset from time-step $t-1$ to time-step t and is, therefore, defined by the formula:

$$r_{i,t} = \frac{p_{i,t} - p_{i,t-1}}{p_{i,t-1}} = \frac{p_{i,t}}{p_{i,t-1}} - 1 \in \mathbb{R} \tag{2.7}$$

where $p_{i,t}$ refers, in our case, to the closing price of day t of a single asset i.

Although this formula was introduced specifically for closing prices, the same transformation can be applied to open, high, low prices and trading volumes with the same meaning of relative change compared to their respective value the previous day. As a consequence, the **simple return matrix** can be defined by applying the simple return formula (2.7) on the asset data matrix defined in (2.5):

$$\vec{\boldsymbol{r}}_{t-W+2:t} = \frac{\vec{p}_{t-W+2:t}}{\vec{p}_{t-W+1:t-1}} - 1 \in \mathbb{R}^{M\times(W-1)\times 5} \tag{2.8}$$

where all operations are applied element-wise.

Figure 2.2 shows that simple returns indeed do not have the problems that prices have, as their values are all centered around 0 despite their prices having different price ranges.

Remark In case the objective is to calculate the return of a portfolio of assets, the formula can be easily obtained from equation 2.7 by replacing the asset value p_t with the portfolio value v_t.

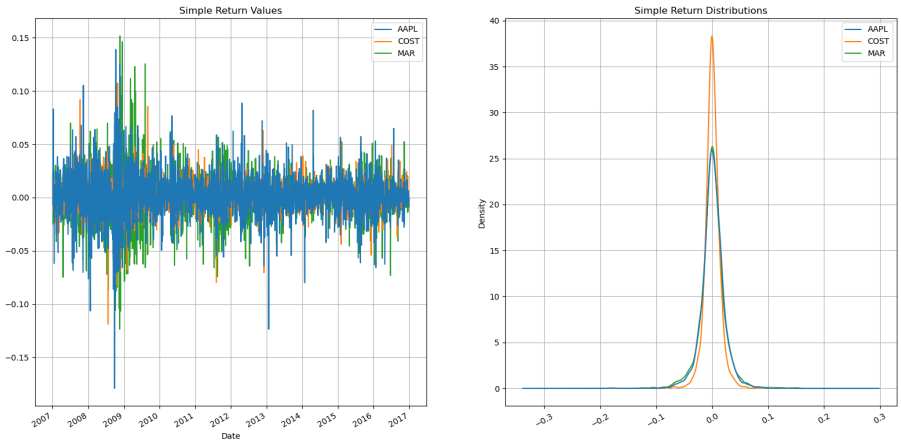


Figure 2.2: Simple returns of *Apple Inc* (AAPL), *Costco Wholesale Corporation* (COST) and *Marriott International Inc* (MAR) from start of 2007 to end of 2017

2.2.2 Log Returns

Despite the preference of simple returns over asset prices, this measure suffers from its asymmetric behaviour. To explain what this lack of symmetry means, a simple example can be given: given an asset that is now valued 100$, if it manages to achieve a simple return of 0.1 (10%) and then of -0.1 the day after, it would mean that its price increased to 110$ and then decreased to 99$, which is not the starting value despite the fact that the returns were both 0.1 in absolute value.

For this reason, the log return was introduced as a symmetric measure of change:

$$\rho_{i,t} = \ln(\frac{p_{i,t}}{p_{i,t-1}}) = \ln(p_{i,t}) - \ln(p_{i,t-1}) = \ln(1 + r_{i,t}) \tag{2.9}$$

Indeed, using the same example as before and replacing simple returns with log returns, it is clear that the two movements cancel each other out, bringing the value asset to its original value of 100$.

In the same way the simple return matrix was defined in equation (2.8), the **log return matrix** can be defined as:

$$\vec{\boldsymbol{\rho}}_{t-W+2:t} = \ln(1 + \vec{\boldsymbol{r}}_{t-W+2:t}) = \ln(\vec{p}_{t-W+2:t}) - \ln(\vec{p}_{t-W+1:t-1}) \in \mathbb{R}^{M\times(W-1)\times 5} \tag{2.10}$$

2.3 Performance Metrics

In order to determine whether a trading strategy is successful or not, a set of evaluation criteria that analyzes the value of the portfolio during the whole trading period needs to be defined.

For this section we will assume that the trading period goes from time-step 1 to time-step T and that the vector:

$$\boldsymbol{v}_{1:T} = \Big[v_1, v_2, \ldots, v_T\Big]^\intercal \in \mathbb{R}^T \tag{2.11}$$

represents the complete trajectory of the value of the portfolio during this period, with v_1 and v_T being respectively its first and last values.

2.3.1 Final Portfolio Value

The easiest way to evaluate the performance of a portfolio is simply to check its value at the end of the trading period used for the experiment and compare it to its initial value without considering how well it performed and the risks it was exposed to during the trading process.

It can be noted that the portfolio value at any time-step t can also be expressed as a function of all the returns realized until then as follows:

$$v_t = v_1 \prod_{t'=2}^{t} \Big(1 + r_{t'}\Big) = v_1 \exp\Big(\sum_{t'=2}^{t} \rho_{t'}\Big) \quad \forall t \tag{2.12}$$

where r_t and ρ_t are respectively the simple return and log return of the portfolio at time-step t obtained, assuming that empty products equal to 1 and empty sums equal to 0 in order for the formula to also work for $t = 1$.

2.3.2 Standard Deviation of Portfolio Returns

In order to take into consideration the risks involved, metrics should account for the volatility of the investments and the most direct way of doing so is by calculating the standard deviation of the rate of return of the portfolio using the vector of portfolio prices to obtain the samples needed to calculate the statistics. Using this metric, an asset is more preferable compared to another if the distribution of its returns has a lower standard deviation.

Standard deviation, however, should not be used as a standalone measure of performance, as minimizing it would only lead to investing everything into a risk-free asset and disregarding everything else. For this reason, a metric that balances the two main objectives of any investor of maximizing expected gains and minimizing risks is needed.

2.3.3 Sharpe Ratio

The **Sharpe ratio**, developed by economist W. Sharpe in 1966 [2], is a criterion that satisfies the aforementioned property. Indeed, it is defined as the ratio of the expected excess return with respect to a risk-free investment to its standard deviation:

$$\boldsymbol{SR} = \frac{\boldsymbol{E}[r_p] - r_f}{\sqrt{\boldsymbol{Var}(r_p)}} \tag{2.13}$$

where the expectation and volatility are calculated from the returns realized by the portfolio during trading and r_f is the risk-free rate of return.

This means that a portfolio strategy has a better performance the higher the Sharpe ratio is, as it means that the expected gain is large with respect to the amount of risk taken.

Sortino Ratio

Using standard deviation as a measure of risk, however, can be a double-edged sword as it does not differentiate between positive and negative variations. For this reason the **Sortino ratio**, which improves upon the Sharpe ratio by using a different denominator, was introduced [3]. Indeed, it replaces the total standard deviation of portfolio returns with the **downside deviation**, which is a measure of downside risk as it is given by the standard deviation of only negative returns. The formula for the Sortino ratio is therefore:

$$\boldsymbol{SortinoR} = \frac{\boldsymbol{E}[r_p] - r_f}{\sigma_d} \tag{2.14}$$

where the numerator is the same of the Sharpe ratio and σ_d is the downside deviation.

2.3.4 Value at Risk

Value at Risk ($\boldsymbol{V@R}$) is another asymmetric risk measure [4]. Given a value $\alpha \in (0,1)$ and the distribution of returns r, the $1-\alpha$ V@R is defined as its $1-\alpha$ quantile:

$$\mathbb{P}\Big(r \leq \boldsymbol{V@R}_{1-\alpha}\Big) = 1-\alpha \tag{2.15}$$

which represents, in simpler terms, the $100\,(1-\alpha)\,\%$ worst case scenario.

To obtain this value from the realizations of daily returns during a trading period of T days, it is sufficient to order them in ascending order and take the element in position αT.

Conditional Value at Risk

Value at risk, however, does not take into consideration what actually happens when the worst case scenario occurs. For this reason, the **conditional value at risk** ($\boldsymbol{CV@R}$) can be a better alternative [4], as it is defined as the expected value of the distribution conditional on the value being smaller (or bigger depending on the distribution) than the value at risk:

$$\boldsymbol{CV@R}_{1-\alpha} = \mathbb{E}\Big[r \;\mid\; r < \boldsymbol{V@R}_{1-\alpha}\Big] \tag{2.16}$$

which represents the expected outcome in case something worse than the $100\,(1-\alpha)\,\%$ worst case scenario happens.

2.3.5 Maximum Drawdown

Last but not least, another useful performance metric is the **Drawdown** ($\boldsymbol{DD}_t$), which is the decline of the portfolio value from when it reached a peak up to the current time-step t, represented as a percentage of the value at the peak:

$$\boldsymbol{DD}_t = \left(\frac{\max_{t' \in (0,t)} v_{t'} - v_t}{\max_{t' \in (0,t)} v_{t'}}\right)^+ \tag{2.17}$$

where v_t is the portfolio value at time-step t. For example, if a portfolio peaked at 1000\$ in value and then dropped to 900\$, its drawdown would be equal to 10%.

The **maximum drawdown** ($\boldsymbol{MDD}_t$) [5], instead, is the maximum value of the drawdown obtained during the whole trading period up to time-step t:

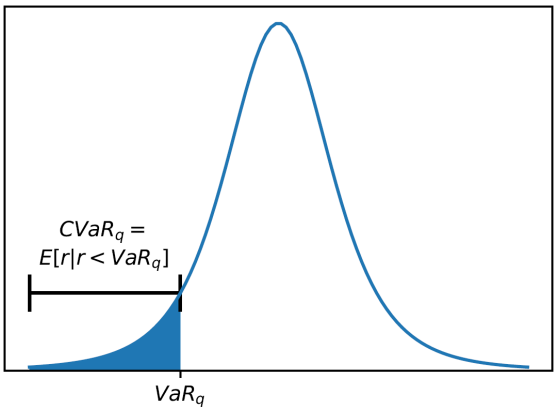


Figure 2.3: $\boldsymbol{V@R_q}$ and $\boldsymbol{CV@R_q}$ of a random distribution. The blue area represents fraction q of the whole area. With respect to their definitions given in 2.15 and 2.16, q takes the role of $1 - \alpha$.

$$\boldsymbol{MDD}_t = \max_{\tau \in (0,t)} \boldsymbol{DD}_\tau = \max_{\tau \in (0,t)} \left(\frac{\max_{t' \in (0,\tau)} v_{t'} - v_\tau}{\max_{t' \in (0,\tau)} v_{t'}} \right)^+ \tag{2.18}$$

2.4 Modern Portfolio Theory

Up until now, the basic concepts of asset and portfolio and the metrics used to evaluate them were introduced. This section will give an overview of the Modern Portfolio Theory, which is essential for the mathematical formulation of static portfolio management, and will motivate the use of Reinforcement Learning to address its dynamic version.

Modern Portfolio Theory (**MPT**), conceived by economist H. Markowitz in 1952 in a work for which he was later awarded the Nobel Price in Economics, introduces the Markowitz model, also called mean-variance model, to solve the problem of selecting the most optimal portfolio that maximizes the expected return of the resulting portfolio while minimizing the level of risk [1].

This model works under the following main assumptions:

- the portfolio risk is given by the volatility of its returns;
- investors are risk averse and rational;

which means that, between two portfolios with the same expected return, the one with lower volatility is preferable and that, given the same volatility, the better portfolio is the one with higher expected return.

2.4.1 Mathematical Formulation

In mathematical terms, the model can be formulated as the following quadratic optimization problem:

$$\begin{aligned} \max_{\boldsymbol{w}} \quad & \boldsymbol{\mu}^\intercal \boldsymbol{w} - \frac{1}{2}\gamma \boldsymbol{w}^\intercal \boldsymbol{\mathcal{E}} \boldsymbol{w} \\ \text{s.t.} \quad & \sum_{i=1}^{M} w_i = 1 \\ & w_i \geq 0 \quad \forall i \end{aligned} \tag{2.19}$$

where $\gamma > 0$ is a risk-aversion coefficient, M is the number of assets in the trading universe, $\boldsymbol{w}$ is the vector of portfolio weights, $\boldsymbol{\mu}$ is the vector of expected returns of the assets and $\boldsymbol{\mathcal{E}}$ is the covariance matrix of asset returns.

Regarding the two constraints, the first one is used to make sure that the percentages sum up to 1, while the second one is due to the assumption we made about not being able to short assets.

As the coefficient of risk-aversion can be difficult to interpret, there are two alternative formulation of the optimization problem that do not make use of such a value. The first one maximizes the expected return while enforcing a target variance of return:

$$\begin{aligned} \max_{\boldsymbol{w}} \quad & \boldsymbol{\mu}^\intercal \boldsymbol{w} \\ \text{s.t.} \quad & \boldsymbol{w}^\intercal \boldsymbol{\mathcal{E}} \boldsymbol{w} \leq \sigma^2_{target} \\ & \sum_{i=1}^{M} w_i = 1 \\ & w_i \geq 0 \quad \forall i \end{aligned} \tag{2.20}$$

while the second one does the opposite by minimizing the portfolio variance while enforcing a target expected return:

$$\begin{aligned} \min_{\boldsymbol{w}} \quad & \boldsymbol{w}^\intercal \boldsymbol{\mathcal{E}} \boldsymbol{w} \\ \text{s.t.} \quad & \boldsymbol{\mu}^\intercal \boldsymbol{w} \geq \mu_{target} \\ & \sum_{i=1}^{M} w_i = 1 \\ & w_i \geq 0 \quad \forall i \end{aligned} \tag{2.21}$$

Indeed, hyperparameters like σ_{target} and μ_{target} are much more interpretable compared to the risk-aversion coefficient γ despite fulfilling the same role of representing the trade-off between expected gain and risk level: high values of μ_{target} and

σ_{target} or low values of γ are used by greedy investors, while the opposite situation is for investors who want to avoid taking risks as much as possible.

2.4.2 Efficient Frontier

Solving the aforementioned optimization problems with different values for the hyperparameters will result in a set of portfolios, each with their own expected return and volatility. Displaying them on a risk-return plot will create a curve that is known in modern portfolio theory as the *efficient frontier* [6]. All portfolios that lay on this curve are said to be optimal since, according to this framework, there does not exist a single portfolio with the same standard deviation of return as any of them but with a higher expected return, which means that all other non optimal portfolios, including those composed by a single asset, lay below the frontier.

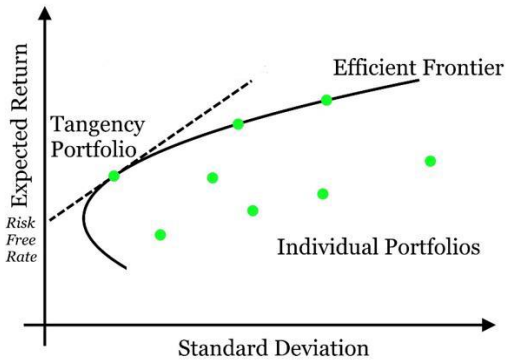


Figure 2.4: Representation of an efficient frontier curve

Optimizing the Sharpe ratio

Nevertheless, the formulations proposed in (2.19), (2.20) and (2.21) still require a hyperparameter to be set in order to obtain a single optimal portfolio. For this reason, a hyperparameterless formulation of the problem can be beneficial, which can be obtained by using the Sharpe ratio as the objective function to maximize:

$$\begin{aligned} \max_{\boldsymbol{w}} \quad & \frac{\boldsymbol{\mu}^\intercal \boldsymbol{w} - r_f}{\boldsymbol{w}^\intercal \boldsymbol{\mathcal{E}} \boldsymbol{w}} \\ \text{s.t.} \quad & \sum_{i=1}^{M} w_i = 1 \\ & w_i \geq 0 \quad \forall i \end{aligned} \tag{2.22}$$

This formulation results in a single optimal most commonly known as the *tangency portfolio* [7], due to it being the point of tangency on the efficient frontier

defined by the straight line that passes through the point $(0, r_f)$, which corresponds to a portfolio made up of only the risk free asset.

2.4.3 Multi-Stage Optimization

The mean-variance model introduced by Markowitz, however, is static and, as such, it is not applicable to the problem of dynamic portfolio management, in which the optimal weights might change at every time-step. One possible fix, which was proposed in [14], would be to include the transaction costs, which are assumed to be equal to a fraction of the whole transaction value, into the objective function, leading to the following formulation:

$$\begin{aligned} \max_{\boldsymbol{w}} \quad & \frac{\boldsymbol{\mu}^\intercal \boldsymbol{w} - r_f - \beta \|\boldsymbol{w} - \boldsymbol{w_0}\|_1}{\boldsymbol{w}^\intercal \boldsymbol{\mathcal{E}} \boldsymbol{w}} \\ \text{s.t.} \quad & \sum_{i=1}^{M} w_i = 1 \\ & w_i \geq 0 \quad \forall i \end{aligned} \tag{2.23}$$

where $\boldsymbol{w_0}$ is the vector of portfolio weights of the previous time-step and β is the transaction cost expressed as a percentage of the transaction value.

Remark Although $\beta \|\boldsymbol{w} - \boldsymbol{w_0}\|_1$ is not the exact formula for the transaction cost, it is an excellent approximation. More details will be given in chapter 4.

This new formulation, however, can lead to myopic solutions as it does not take into consideration the long term effect of its actions. For this reason, a reinforcement learning approach is preferable, since its main objective is to maximize a given objective in the long run. Nevertheless, the adapted Markowitz model with transaction costs can still be used as a comparison.

Chapter 3

Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm that is fundamentally different from other types of machine learning like supervised learning, as the learning is not achieved by observing i.i.d. observations, but by continuously choosing an action to perform in order to interact with the environment and observing how it responds to the stimulus.

This chapter will introduce the reinforcement learning framework and its main components. Furthermore, an overview of the deep learning approach to the RL problem will be given, as all algorithms employed in this work revolve around the use and training of neural networks.

3.1 Framework Definition

Figure 3.1 shows the configuration of a simple reinforcement learning system: at every time-step t the **agent**, also called the decision maker, observes the **state** $\boldsymbol{s}_t$ of the **environment** and interacts with it by applying an **action** $\boldsymbol{a}_t$. The environment then reacts to the action made by the agent by emitting a **reward** signal $\boldsymbol{r}_{t+1}$ and a new state $\boldsymbol{s}_{t+1}$, which will be observed by the agent and so on.

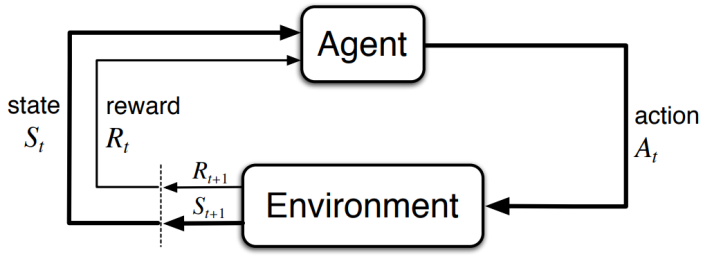


Figure 3.1: Interaction between agent and environment. Image taken from [10].

Action Space

As mentioned before, actions are the signals emitted by the agent at every time-step in order to interact with the environment. The set of all admissible actions is called the **action space $\boldsymbol{A}$** and can be either discrete or continuous depending on the application.

Environment State

Actions performed by the agents are chosen according to the observations received from the environment. The set of all observations is called **observation space**, which, however, does not necessarily coincide with the **environment space**, which refers to the set of actual states of the system. This is the case, for example, for trading agents that can only observe the current prices of the assets in the market.

In this case, the environment is said to be *partially observable* and, therefore, in order to perform the optimal actions, the agent is forced to build its own representation of the state of the environment by relying only on past observations, actions and reward signals. On the other hand, the environment is said to be *fully observable*.

Reward Signal

The **reward signal** is a scalar value $r_{t+1} \in \mathbb{R}$ that the agent receives after applying action a_t during the previous time-step, which indicates the quality of the action with respect to the current state of the environment. Assuming that this action made the environment transition from state s_t to s_{t+1}, the reward is a function of these three values:

$$r_{t+1} = r_{t+1}\left(s_t, a_t, s_{t+1}\right) \quad \forall t \tag{3.1}$$

The objective of the agent is to maximize the cumulative reward received, which means that the focus is on the long-run effects of the actions, rather than their immediate results. This is due to the **reward hypothesis**, which all of reinforcement learning is based on, that states:

All goals and purposes can be thought of as the maximization of the expected value of the cumulative sum of a received scalar signal called reward.

This hypothesis is based on the fact that, for an agent to choose an optimal action, all possible choices must be able to be compared against each other, which means that they must be able to be expressed as a scalar value, just like a reward signal. For this reason, it is imperative that an appropriate reward signal is chosen for the given tasks on which the RL framework is applied [9].

3.1.1 Rewards, Policies and Value Functions

In order for the agent to understand what is the best action to perform at any given moment using only reward signals, some key concepts used in reinforcement learning need to be defined first.

Future Discounted Reward

Given the series of reward signals $r_1, r_2, \ldots, r_t, \ldots$ and a discount factor $\gamma \in [0, 1]$, the **future discounted reward** at time-step t is defined as:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \tag{3.2}$$

which represents the present value of the total reward the agent will receive starting from time-step t if it were to apply the actions that produced those reward signals.

The coefficient γ is used to imply the preference of receiving a reward sooner rather than later and determines the present value of future rewards: values of γ close to 1 signify a farsighted agent that takes the future into account very strongly, while smaller values are representative of a myopic agent that cares more about immediate gains.

Policy

A **policy** $\boldsymbol{\pi}$ is a function that represents the behaviour of the agent by mapping environment states s_t to actions a_t. Depending on the situation, this function can be:

- deterministic: $a_t = \boldsymbol{\pi}(s_t)$;
- stochastic: $a_t \sim \boldsymbol{\pi}(\cdot \mid s_t)$.

which means that the action can either be the output of a standard function or sampled from a distribution conditioned on the current environment state.

Value Function

The **state-value function** for a given policy $\boldsymbol{\pi}$ of a state s, denoted $v_\pi(s)$, is the expected future discounted reward of an agent if the environment is in state s and the agent starts following the policy $\boldsymbol{\pi}$ from then on. Formally, it is given by:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi\left[G_t \mid s_t = s\right] \\ &= \mathbb{E}_\pi\Big[\sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \mid s_t = s\Big] \quad \forall s \end{aligned} \tag{3.3}$$

Q Value

Similarly to the state-value function that maps states to their value under a given policy, there exists a function that maps state-action pairs to their value. This function is called **action-value function**, or more colloquially **Q value**, for the policy $\boldsymbol{\pi}$, and is given by the formula:

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}_\pi\left[G_t \mid s_t = s, a_t = a\right] \\ &= \mathbb{E}_\pi\Big[\sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \mid s_t = s, a_t = a\Big] \quad \forall s, a \end{aligned} \tag{3.4}$$

Informally, $q_\pi(s,a)$ represents the value of state s if the agent were to perform action a and then follow the policy $\boldsymbol{\pi}$.

3.2 Markov Decision Processes

Basic reinforcement learning problems can be formulated as **Markov decision processes** (MDC), which are discrete-time stochastic dynamical systems that work under the assumptions that there is perfect knowledge about the **reward function** and the **transition probabilities** between environment states, and that the **Markov property** holds for all states.

State Transition Probabilities

Given the set of states $\boldsymbol{S}$ and set of actions $\boldsymbol{A}$, assuming that both of them are finite in size, the **state transition probabilities** can be represented by a three-dimensional matrix $P \in [0,1]^{|\boldsymbol{S}|\times|\boldsymbol{A}|\times|\boldsymbol{S}|}$ where:

$$P(s,a,s') = \mathbb{P}\left[s_{t+1} = s' \mid s_t = s, a_t = a\right] \quad \forall t, s \in \boldsymbol{S}, a \in \boldsymbol{A}, s' \in \boldsymbol{S} \tag{3.5}$$

Reward Function

As the system is stochastic, the **reward function** R_t at time-step t is defined as the expected value of the reward signal r_{t+1} if the environment is currently in state s and the agent performs action a, that is:

$$R_t(s,a) \doteq \mathbb{E}\left[r_{t+1} \mid s_t = s, a_t = a\right] \quad \forall t, s \in \boldsymbol{S}, a \in \boldsymbol{A} \tag{3.6}$$

Using the assumption that the state and action spaces are finite, the expectation can be expressed in simpler terms using the state transition probabilities:

$$R_t(s,a) = \sum_{s' \in \boldsymbol{S}} \mathbb{P}\left[s_{t+1} = s' \mid s_t = s, a_t = a\right] \cdot r_{t+1} \quad \forall t, s \in \boldsymbol{S}, a \in \boldsymbol{A} \tag{3.7}$$

where $r_{t+1} = r_{t+1}(s_t, a_t, s_{t+1})$ is the reward signal that the agent will receive if the environment were to transition from state s to state s' due to the effect of action a_t.

Markov Property

A state s_{t+1} is said to satisfy the **Markov Property** if and only if the transition to this state only depends on the current state s_t and the action a_t performed and not also on the previous ones, which means that the following condition must be satisfied:

$$\mathbb{P}\left[s_{t+1} \mid s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_1, a_1\right] = \mathbb{P}\left[s_{t+1} \mid s_t, a_t\right] \quad \forall t \tag{3.8}$$

Remark The validity of the Markov property heavily depends on what is considered as the state of the environment. Indeed, if, for example, we consider as a dynamical system the Fibonacci sequence:

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad \forall n > 1 \end{cases} \tag{3.9}$$

then it is clear that this system is not Markovian if the current state is defined only by the current value F_n, while it satisfies the Markov property if the state is defined by the pair of current value and previous value (F_n, F_{n-1}).

For this reason it is important to properly define what the environment state is, especially for more complex systems.

Having introduced all the components of a MDP, it can now easily be defined as the tuple $\langle \boldsymbol{S}, \boldsymbol{A}, P, R, \gamma \rangle$.

3.2.1 Bellman Equation

By applying the Markov property to the definitions of future discounted reward and of value function, it is possible to notice a recursive relationship:

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\Big[\sum_{i=0}^{\infty}\gamma^i r_{t+i+1} \mid s_t = s\Big] \\
&= \mathbb{E}_\pi\Big[r_{t+1} + \sum_{i=1}^{\infty}\gamma^i r_{t+i+1} \mid s_t = s\Big] \\
&= \mathbb{E}_\pi\Big[r_{t+1} + \gamma\sum_{i=0}^{\infty}\gamma^i r_{t+i+2} \mid s_t = s\Big] \\
&= \mathbb{E}_\pi\Big[r_{t+1} + \gamma G_{t+1} \mid s_t = s\Big] \\
&= \mathbb{E}_\pi\Big[r_{t+1} + \gamma v_\pi(s_{t+1}) \mid s_t = s\Big]
\end{aligned}
\tag{3.10}
$$

which implies that, for a Markovian system, the state-value function of any state $s \in \boldsymbol{S}$ can be decomposed into the immediate reward r_{t+1} and the discounted value of the successor state s_{t+1}. This equality is known as **Bellman equation**.

Using the same reasoning, the action-value function of any state-action pair (s, a) can also be decomposed into the same two components:

$$
\begin{aligned}
q_\pi(s,a) &= \mathbb{E}_\pi\left[r_{t+1} + \gamma v_\pi(s_{t+1}) \mid s_t = s, a_t = a\right] \\
&= \mathbb{E}_\pi\Big[r_{t+1} + \gamma q_\pi(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a\Big]
\end{aligned}
\tag{3.11}
$$

Remark Most of the steps of the proof to obtain the Bellman equation were skipped. If the reader is interested, the complete proof can be found in [10].

Bellman Optimality Equation

Solving a reinforcement learning task using the MDP framework consists in finding a policy $\boldsymbol{\pi}$ which is better than any other admissible policy $\boldsymbol{\pi'}$ and this happens if and only if the state-value function of all states $s \in \boldsymbol{S}$ under the policy $\boldsymbol{\pi}$ is better than the one under policy $\boldsymbol{\pi'}$, that is:

$$
v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in \boldsymbol{S} \tag{3.12}
$$

All policies that satisfy this condition are called **optimal policies** and are denoted by $\boldsymbol{\pi}_*$. They also share the same state-value function v_*, which is called the **optimal state-value function** and is defined as:

$$
v_*(s) \doteq \max_\pi v_\pi(s) \quad \forall s \in \boldsymbol{S} \tag{3.13}
$$

Likewise, the **optimal action-value function** q_* can also be defined as:

$$
q_*(s,a) \doteq \max_\pi q_\pi(s,a) \quad \forall s \in \boldsymbol{S}, a \in \boldsymbol{A} \tag{3.14}
$$

which is connected to the optimal state-value function by the following relationship:

$$v_*(s) = \max_{a \in \boldsymbol{A}} q_*(s,a) \quad \forall s \in \boldsymbol{S} \tag{3.15}$$

The same reasoning used to find the Bellman equation can also be applied to v_* and p_* in order to obtain recursive relationships that do not depend on any specific policy:

$$\begin{aligned} v_*(s) &= \max_{a \in \boldsymbol{A}} q_*(s,a) \\ &= \max_{a \in \boldsymbol{A}} \mathbb{E}_{\pi_*}\Big[\sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \mid s_t = s, a_t = a\Big] \\ &= \max_{a \in \boldsymbol{A}} \mathbb{E}_{\pi_*}\Big[r_{t+1} + \gamma G_{t+1} \mid s_t = s, a_t = a\Big] \\ &= \max_{a \in \boldsymbol{A}} \mathbb{E}\Big[r_{t+1} + \gamma v_*(s_{t+1}) \mid s_t = s, a_t = a\Big] \end{aligned} \tag{3.16}$$

and

$$q_*(s,a) = \mathbb{E}\Big[r_{t+1} + \gamma \max_{a' \in \boldsymbol{A}} q_*(s_{t+1}, a') \mid s_t = s, a_t = a\Big] \tag{3.17}$$

which hold for all $s \in \boldsymbol{S}$ and $a \in \boldsymbol{A}$ and $s' \in \boldsymbol{S}$. These equations are called the **Bellman optimality equations** for v_* and q_* respectively.

3.3 Dynamic Programming

The appeal of modeling stochastic dynamical systems as Markov decision processes is the possibility of obtaining an optimal policy by solving them using the Bellman optimality equations introduced in the previous section. Algorithms that use this method of computing the optimal solution belong to the **dynamic programming** (DP) framework.

From now on, we will assume that the state and action spaces are finite, which means that the environment dynamics are given by a set of probabilities. This is due to the fact that, in general, problems with continuous spaces cannot be solved exactly and approximations are required.

Iterative Policy Evaluation

The first step toward finding an optimal policy is finding the value of all states under a given policy $\boldsymbol{\pi}$. To do so, it is sufficient to apply the **iterative policy evaluation** algorithm, which simply consists in assigning a starting value v_0 as an initial approximation of the state-value function and then apply the following update rule inspired by the Bellman equation:

$$v_{k+1}(s) = \mathbb{E}_\pi\Big[r_{t+1} + \gamma v_k(s_{t+1}) \mid s_t = s\Big] \quad \forall s \in \boldsymbol{S} \tag{3.18}$$

This algorithm is proved to converge to the exact state-value function v_π for $k \to \infty$ as it is the fixed point of the Bellman equation. Of course, actual implementations of this algorithm cannot use an infinite number of iterations and stop when the difference between v_k and v_{k+1} becomes smaller than a given threshold for all states. With the state-value function, it becomes trivial to also evaluate the action-value function q_π for all states due to the relationship found in (3.11).

3.3.1 Policy Iteration

Policy Iteration is an algorithm that, starting from a given policy $\boldsymbol{\pi}$, is able to iteratively improve upon it by first computing its state-value function and then finding a better policy $\boldsymbol{\pi'}$ by relying on the following **policy improvement theorem**, whose proof can be found in [10]:

Let $\boldsymbol{\pi}$ and $\boldsymbol{\pi'}$ be any pair of deterministic policies such that $q_\pi(s, \boldsymbol{\pi'}(s)) \geq v_\pi(s)\ \forall s \in \boldsymbol{S}$, then it must hold that $v_{\pi'}(s) \geq v_\pi(s)\ \forall s \in \boldsymbol{S}$, which means that the policy $\boldsymbol{\pi'}$ must be at least as good as, if not better than, policy $\boldsymbol{\pi}$. Moreover if there is a strict inequality for the first expression at any state, then there must also be an inequality for the second expression at that state.

In particular, a new policy that satisfies the condition required by the theorem is the greedy policy $\boldsymbol{\pi'}$ that always chooses the action that maximizes the action-value function under policy $\boldsymbol{\pi}$, that is:

$$\begin{aligned} \boldsymbol{\pi'}(s) &\doteq \arg\max_{a \in \boldsymbol{A}} q_\pi(s, a) \\ &= \arg\max_{a \in \boldsymbol{A}} \mathbb{E}\Big[r_{t+1} + \gamma v_\pi(s_{t+1}) \mid s_t = s, a_t = a\Big] \quad \forall s \in \boldsymbol{S} \end{aligned} \tag{3.19}$$

By alternating the evaluation of the current policy and its improvement, the policy iteration algorithm will keep finding a better policy until it converges to the optimal policy $\boldsymbol{\pi}_*$

3.3.2 Value Iteration

The policy iteration algorithm works by directly updating the policies, which can be computationally expensive as it has to iterate through all states of the environment multiple times. One way to avoid this problem is to use only one update step during the iterative policy evaluation parts of the algorithm, which lead to a new

algorithm called **value iteration**. As the name implies, this algorithm works only with state-value functions and can be expressed as a simple update rule:

$$v_{k+1}(s) = \max_{a \in \boldsymbol{A}} \mathbb{E}\Big[r_{t+1} + \gamma v_k(s_{t+1}) \mid s_t = s, a_t = a\Big] \quad \forall s \in \boldsymbol{S} \tag{3.20}$$

Starting from an arbitrary state-value function v_0, this algorithm is shown to converge to the optimal state-value function v_*, which is then used to find the optimal policy $\boldsymbol{\pi}_*$ by using:

$$\boldsymbol{\pi}_*(s) = \arg\max_{a \in \boldsymbol{A}} \mathbb{E}\Big[r_{t+1} + \gamma v_*(s_{t+1}) \mid s_t = s, a_t = a\Big] \quad \forall s \in \boldsymbol{S} \tag{3.21}$$

3.4 Model-Free Learning

The main problem of dynamic programming and the reason why they are not heavily used is the MDP assumption that there is perfect knowledge about the dynamics of the environment, which is not the case for many real-world applications. For this reason, multiple algorithms that do not make any assumption about complete knowledge and instead rely only on interacting with the environment by sampling sequences of states, actions and rewards have been developed.

One of the most important framework used in this context is known as **temporal-difference** (TD) learning, which is based on iteratively updating the value $q(s, a)$ of state-action pairs (s, a) by using the value of another pair (s', a') where s' is the next state of the environment, obtained after applying action a from state s, hence the name temporal difference. The way in which the action a' is chosen identifies two main algorithms:

- **SARSA**: based on the policy iteration algorithm for MDP, it uses the temporal difference method to evaluate the action-value function q_π of a given policy $\boldsymbol{\pi}$, which means that $a' = \boldsymbol{\pi}(s')$. Starting from an initial approximation for the Q-values, the following update rule is iteratively applied to obtain a better approximation:

$$q_\pi(s, a) \leftarrow q_\pi(s, a) + \alpha\Big(\underbrace{r + \gamma q_\pi(s', a') - q_\pi(s, a)}_{\text{temporal difference}}\Big) \tag{3.22}$$

  where r is the reward signal received by transitioning from state s to state s' by performing action a and α is a coefficient known as **learning rate**.

  These approximations are then used to improve the current policy using a greedy approach: $\boldsymbol{\pi}_{new}(s) \in \arg\max_{a \in \boldsymbol{A}} q_\pi(s, a) \quad \forall s \in \boldsymbol{S}$, which is evaluated using the process defined in (3.22) and so on. This algorithm is said to follow

an **on-policy** approach, as the actions to be performed are always taken from the current policy.

- **Q-learning**: based on the value iteration algorithm for MDP, it aims at learning an approximate solution to the Bellman optimality equation (3.14) by using the temporal difference method to iteratively improve the current approximation of the Q-values [11]. To do so, the algorithm follows a greedy strategy, which means that $a' \in \arg\max_{a \in \boldsymbol{A}} q(s,a)$. Therefore, the update rule for the action-value function is given by:

$$q_*(s,a) \leftarrow q_*(s,a) + \alpha \Big(\underbrace{r + \gamma \max_{a' \in \boldsymbol{A}} q_*(s',a') - q_*(s,a)}_{\text{temporal difference}} \Big) \tag{3.23}$$

  Q-learning is said to be an *off-policy* method, due to the fact that it learns the optimal policy independently of the policy being followed.

This section contains a simple introduction to some of the algorithms used in reinforcement learning with discrete state and action spaces. If the reader is interested in a deeper analysis, they should read [10].

3.4.1 Exploration-Exploitation Trade-Off

It is clear that, in order to find the optimal strategy regardless of the type of algorithm used, a good approximation of the Q values for all state-action pairs is needed. Indeed, in a pure *exploitation* strategy in which all actions are chosen in a greedy manner, it might happen that particularly good actions with a bad initial approximation are discarded for other less optimal ones. On the other hand, a pure *exploration* approach that consists in selecting the actions at random can be particularly inefficient, especially for large action spaces. For this reason, there needs to be a good balance between exploiting the good actions and exploring the less chosen ones [12].

A very naive solution to this problem is the ϵ-greedy approach, which consists in selecting the most promising action with probability $1 - \epsilon$ and selecting a random one with probability ϵ, where the value of ϵ can either be fixed to a constants or decrease during the learning process as the estimations get better.

3.4.2 Episodic vs Continuous Tasks

It should also be noted that there are two different types of tasks:

- **episodic tasks**: they are tasks in which the interaction between agent and environment can be naturally separated into separate episodes, so that, at the

end of every episode, the environment is reset to its initial state and the next episode starts independently of how the previous one ended;

- **continuous tasks**: they are tasks in which the agent-environment interaction goes on continually without limit. They are the main reason why discounted rewards are used, since, without discounting, the expected future rewards would explode to infinity.

3.5 Deep Learning

Up until now, we have always assumed that the state and action spaces were finite, which allowed the use of a tabular formulation for state-value and action-value functions. Many applications, however, are not compatible with this assumption and, therefore, there is a need to find a representation for these functions so that they work with a continuous range of values for both inputs and outputs. Nowadays, the most common solution is the use of neural networks to approximate them [15].

3.5.1 Neural Networks

Originated from an attempt to find a mathematical model for the neural systems of living entities [16], **artificial neural network** can be considered as black-box models composed by a large collection of simpler units connected to each other via edges that represent the flow of information inside the network. Each unit is represented by a set of parameters, which control the output it will produce given an array of scalar values as input.

Depending on the application, there are different types of neural networks, which can also be combined to obtain even more complex structures. This section will only give a brief overview of the architectures used during the experiments.

Feed-Forward Neural Network

The **Feed-forward neural network**, also called multi-layer perceptron, is one of the simplest type of neural network that exists. It consists in a set of nodes, called neurons, that are connected in a feed-forward manner, meaning that they do not form any loops. These nodes can be categorized into three different types:

- input nodes: they are the nodes that receive the input data. In particular, each node corresponds to one specific feature of a data point;

- hidden nodes: they receive the data as a linear combination of the values of input nodes or other hidden nodes, where the linear combination coefficients

are given by the weights of the connections. They then output a value that will be used as input for other nodes;

- output nodes: they are just like hidden nodes, with the only difference being the fact that their outputs are used as the output of the whole neural network and are not fed into other nodes.

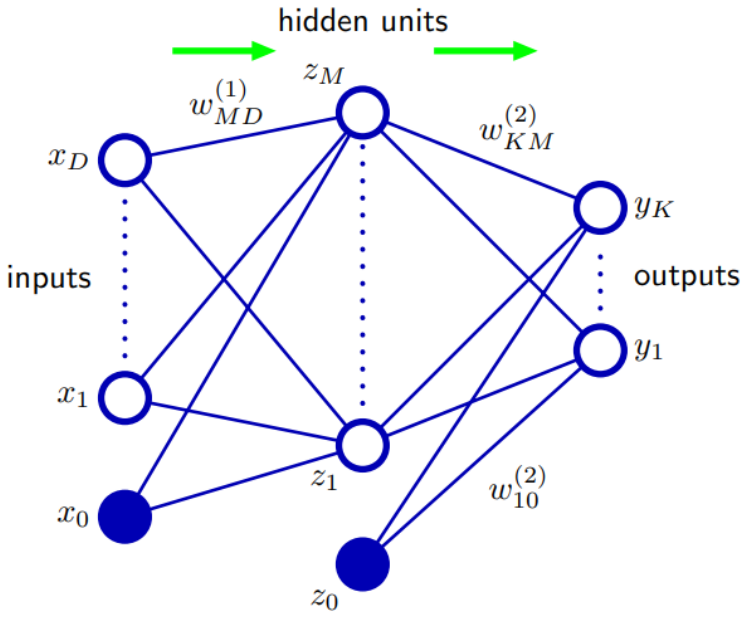


Figure 3.2: Example of a feed-forward neural network with a single hidden layer. Image taken from [16].

As shown in figure 3.2, these nodes are organized into layers in such a way that all nodes from one layer are connected to all nodes of the next layer, which is why these layers are also called *fully connected* (FC) layers. Furthermore, each layer excluding the output one also has a bias node, which is used to add a constant value to its output.

It should also be noted that, with this structure, the network would only be able to give as output a linear combination of the input values. For this reason, all linear combinations are followed by a non-linear transformation, called *activation function*. In mathematical terms, if $(x_1, \ldots, x_M)$ is the vector of values of a given layer and $(w_{i,1}, \ldots, w_{i,M})$ are the weights of the edges that connect this layer to the i-th node of the next layer, then its value is given by:

$$o_i = \sigma\Big(\sum_{j=1}^{M} w_{i,j} x_j + w_{i,0}\Big) \quad \forall i \tag{3.24}$$

where σ is the chosen non-linear transformation and $w_{i,0}$ is the weight of the edge that connect the bias node of the current layer to the i-th node of the next layer.

Recurrent Neural Network

Recurrent neural networks (RNN) are a family of neural networks designed to handle sequential data [13], such as time series and sentences. Unlike the standard multi-layer perceptron, this type of network uses feedback loops, which can be seen in figure 3.3.

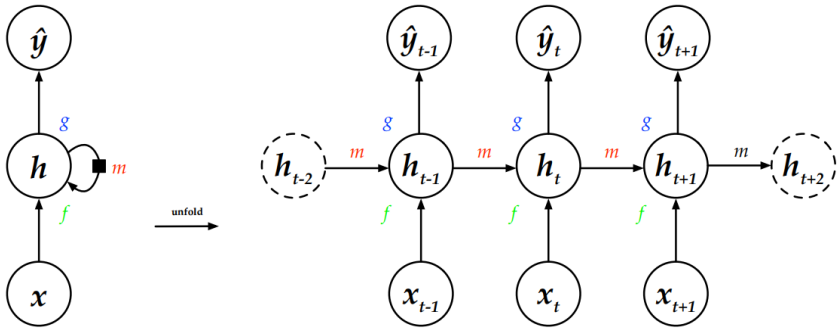


Figure 3.3: Computational graph of a generic recurrent neural network. The left part shows the simplified representation, which is unfolded on the right to show the exact flow of information for a single time-step. Image taken from [14].

Assuming that the input is a time series $\boldsymbol{x} = [x_1, \ldots, x_t, \ldots, x_T]$, the network recursively builds a hidden representation h_t of the input data x_t using the value of the data at that time-step and the hidden representation of the state at the previous time-step h_{t-1}:

$$h_t = f\left(h_{t-1}, x_t\right) \tag{3.25}$$

where f is a function that depends on the parameters of the network. The hidden representations h_t are then used to compute the output of the network y_t for all time-steps $t = 1, \ldots, T$ using a second function g:

$$y_t = g\left(h_t\right) \tag{3.26}$$

As both the figure and mathematical expression show, the transformations f and g that are applied to the data are the same for all time-step. This property

is known as *parameter sharing* and is a fundamental part of all recurrent neural network, as it allows them to avoid learning a different model for every single time-step, which would lead to much higher computational costs compared to using a multi-layer perceptron and possibly overfitting.

Standard RNNs, however, suffer from the *vanishing gradient problem* [17], which is encountered when training neural networks with gradient-based algorithms and consists in having a vanishingly small gradient that prevent the value of some of the parameteres from being changed. For this reason, other variants have been developed. The most popular one is the long short-term memory (**LSTM**) architecture, which is heavily used due to its ability to handle both long-term and short-term dependencies in the input data, as its name implies [18].

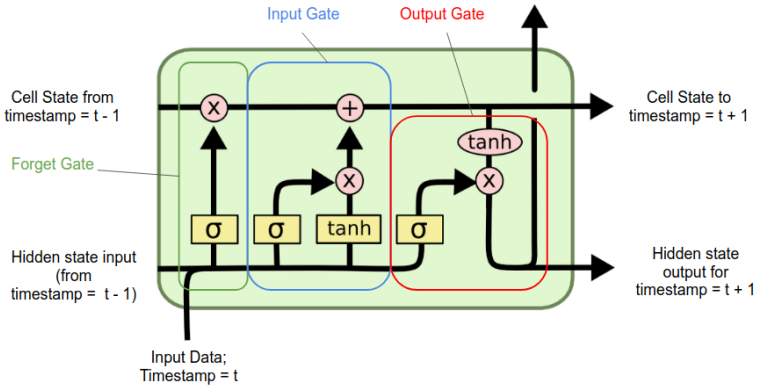


Figure 3.4: Diagram of an LSTM cell. Image taken from [19].

This is achieved thanks to the way its components interact:

- hidden state: it contains information on previous inputs and is also used as the output of the network for all time-steps;
- cell state: it works as the memory of the network as it stores information from both earlier and more recent time-steps according to the behaviour of the gates;
- forget gate: it decides what information should be kept in the cell state and what should be forgotten;
- input gate: it handles the role of updating the cell state using the current input data and the previous hidden state of the network;
- output gate: it decides what the next hidden state should be by using the current input and the previous hidden state.

Figure 3.4 shows the diagram of a single LSTM cell and highlights all of its components. In mathematical terms, this structure can be expressed using the following set of equations:

$$\begin{cases} f_t = \sigma\left(W_f x_t + U_f h_{t-1} + b_f\right) & \text{(forget gate)} \\ i_t = \sigma\left(W_i x_t + U_i h_{t-1} + b_i\right) & \text{(input gate)} \\ o_t = \sigma\left(W_o x_t + U_o h_{t-1} + b_o\right) & \text{(output gate)} \\ g_t = \tanh\left(W_g x_t + U_g h_{t-1} + b_g\right) & \\ c_t = f_t \odot c_{t-1} + i_t \odot g_t & \text{(cell state)} \\ h_t = o_t \odot \tanh\left(c_t\right) & \text{(hidden state)} \end{cases} \tag{3.27}$$

where $\odot$ refers to the element-wise product operator and W, U and b are parameters to be learnt.

3.5.2 Training

As mentioned when we introduced the concept of neural networks, their behaviour depend on the value of their parameters, which means that, in order for them to output optimal values in response to given inputs, they have to be trained using a sufficiently large amount of data relative to the number of parameters. The training process consists in the minimization of a loss function, which, in the context of reinforcement learning, does not have the same meaning as the loss function used in supervised learning, as it does not represent the performance of the policy [20].

Gradient Descent

The minimization of the loss function is usually achieved via a process called **gradient descent**, which, as the name implies, consists in calculating the gradient of the loss function obtained on all the training data with respect to the parameters of the network and then iteratively updating the parameters using the following update rule:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla L\left(\boldsymbol{w}\right) \tag{3.28}$$

where $\boldsymbol{w}$ is the vector of parameters of the network, $\nabla L\left(\boldsymbol{w}\right)$ is the gradient of the loss function and $\eta \in \mathbb{R}_+$ is a hyperparameter of the training process known as **learning rate**, which defines the step size at each iteration of the algorithm. Computing gradients using the whole training dataset at every iteration, however, can be too demanding in terms of computational resources required. For this reason, a batch learning variant, called **Stochastic Gradient Descent** (SGD) [21], has been proposed, which works in the same way as the original algorithm

with the only difference being that the gradient is computed using a fixed-size subset of data samples called minibatch.

In reality, most of the time, actual training of neural networks do not use the gradient descent algorithm or its batch learning version as is and, instead, opt for more complex variants that have been proved to have better performance in general, such as **Adam**, which relies on gradient scaling and the use of momentum [22].

Furthermore, in order to avoid overfitting on the training data as much as possible, these algorithms also make use of regularization, which consists in augmenting the original loss function by adding a regularization term so that the training process minimizes the sum of the two. One of the most commonly used regularization method is called L2 regularization, or **weight decay** [23], which can be explained with the following expression:

$$L(\boldsymbol{w}) = L_{old}(\boldsymbol{w}) + \lambda \|\boldsymbol{w}\|_2 \tag{3.29}$$

where L_{old} is the original loss function and λ is a hyperparameter that controls the degree of regularization. This method leads to smaller values for the parameters of the network, making it more robust to noise in the input.

3.6 Deep Learning Approach to Reinforcement Learning

Recent advances in the field of deep learning, especially applied to tasks related to computer vision and natural language processing, have proven the ability of neural networks to extract high-level features from raw data. For this reason, many researchers tried to adapt deep learning techniques to the reinforcement learning framework, which, at the time, mainly relied on linear combinations of hand-crafted features [24], leading to the creation of a variety of new algorithms as shown in figure 3.5.

Reinforcement learning algorithms can be divided into two groups: *model-based* approaches are based on trying to learn the behaviour of the environment by interacting with it, while *model-free* ones try to directly optimize the policy or the value function without inferring anything about the environment they interact with. In this thesis we will mostly focus on the second class of RL methods, although we will also briefly rely on the first kind.

Just like algorithms used to solve Markov decision processes rely either on improving the policy or on finding the optimal value function to find the best strategy, model-free RL approaches can also be categorized into two classes:

- **Policy Optimization**: this family of algorithms works by directly trying

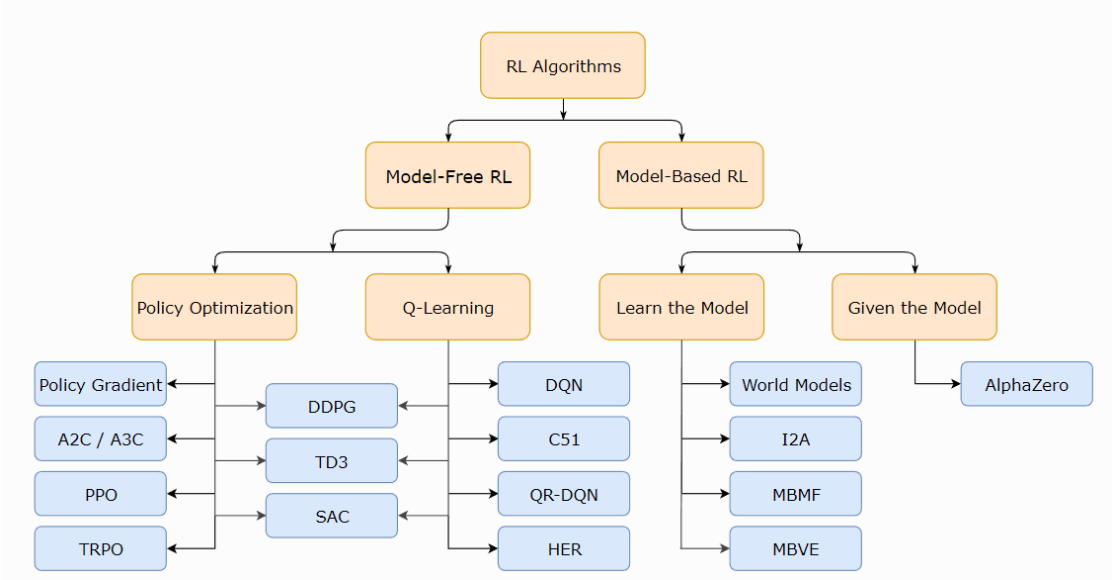


Figure 3.5: Non-exhaustive taxonomy of modern reinforcement learning algorithms. Image taken from [27].

to find the optimal parameters of the network used as the policy function. In particular, they usually perform on-policy optimization by learning the state-value function v_π of the current policy $\boldsymbol{\pi}$, which is then used to find a better one, just like in the policy iteration algorithm for MDP;

- **Q-Learning**: as the name implies, algorithms belonging to this group work in the same way as the Q-learning algorithm introduced in section 3.4, with the only difference being that it does not try to learn the Q-value for all state-action pairs directly and, instead, relies on finding the optimal value for the parameters of the neural network that approximates the action-value function.

Both types of algorithms have their own strengths and weaknesses. In particular, while policy optimization methods are often more reliable than Q-learning methods, which are prone to instabilities [29], they are less sample efficient than the counterpart [30]. As a solution, various algorithms that make use of both frameworks simultaneously have been developed.

3.6.1 Actor-Critic Methods

Using a different terminology, the term actor is used to refer to the policy function, while critic is used for the action-value function. **Actor-Critic methods**, therefore,

refer to algorithms that aim to combine the strong points of policy optimization methods and Q-learning methods, which are also called actor-only and critic-only methods respectively [31].

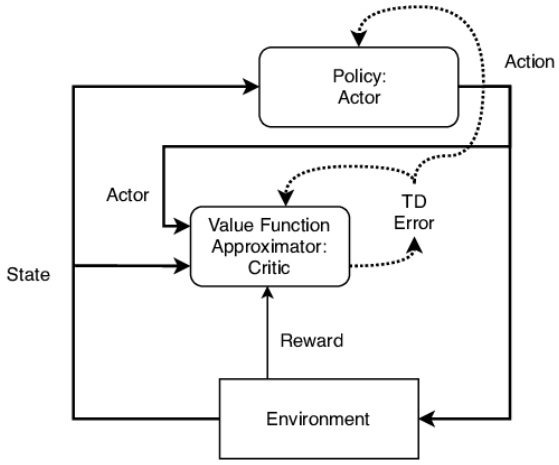


Figure 3.6: Dynamic of actor-critic algorithms. Image taken from [32].

In particular, the actor and the critic, which are represented by neural networks whose parameters have to be learnt, interact with each other and with the environment as shown in figure 3.6: the actor decides which action to perform depending on the current state of the environment, while the critic tells the actor how good its action was and how the parameters of the two networks should be adjusted based on the reward signal received from the environment.

A variety of actor-critic algorithms have been developed since their conception. We will now explain in details only the two that were used during the experiments performed for this thesis: DDPG and SAC.

DDPG

Short for **Deep Deterministic Policy Gradient**, DDPG is an off-policy algorithm developed as an adaptation of the *Deep Q Network* (DQN) algorithm [24], which is based on Q-learning, to deal with continuous action spaces [33]. Indeed, Q-learning algorithms struggle with this type of problems because they follow a greedy policy, which requires finding the optimal action at every time-step, and this can be impractical for high-dimensional continuous action spaces.

In particular, DDPG uses four networks: a deterministic actor network $\boldsymbol{\pi}_\phi$, a critic network Q_θ and two target networks $\boldsymbol{\pi}'_{\phi'}$ and $Q'_{\theta'}$, which are used to stabilize the training of the critic network [25]. Indeed, the training process consists in solving two optimization problem simultaneously:

- optimizing the parameters ϕ of the policy so that the expected future return is maximized, meaning that the objective function to maximize is given by:

$$J_\pi(\phi) = \mathbb{E}\Big[Q\left(s, \boldsymbol{\pi}_\phi(s)\right)\Big] \tag{3.30}$$

- finding the optimal parameters θ of the critic network so that it satisfies the Bellman equation 3.11 by minimizing the following loss function:

$$L(\theta) = \mathbb{E}\Big[\left(Q_\theta(s_t, a_t) - y_t\right)^2\Big] \tag{3.31}$$

where

$$y_t = r_{t+1}(s_t, a_t, s_{t+1}) + \gamma Q_{\theta'}\left(s_{t+1}, \boldsymbol{\pi}_{\phi'}(s_{t+1})\right) \tag{3.32}$$

In order to train the actor and critic networks using the aforementioned objectives, the algorithm applies gradient descent algorithms using samples obtained from interactions between the actor and the environment. In particular, it makes use of an *experience memory replay buffer* [26] to store the last $N \in \mathbb{N}$ agent-environment interactions in the shape of tuples $(s_t, a_t, s_{t+1}, r_{t+1})$, which are then used as samples to form the batches needed for the training process. This addresses the issue of not having i.i.d. data due to the samples being generated sequentially from interacting with the environment and allows the networks to be trained offline using batches rather than online with single samples, improving the computational efficiency of the training process. It should be noted that the use of batches allows the loss defined in (3.31) to be expressed as:

$$L(\theta) = \frac{1}{B}\sum_i \Big(Q_\theta(s_t, a_t) - y_t\Big)^2 \tag{3.33}$$

where B is the number of samples in the batch.

On the other hand, the target networks are not trained and, instead, have their parameters updated with the following soft update rule:

$$\begin{cases} \phi' \leftarrow \tau\phi + (1-\tau)\,\phi' \\ \theta' \leftarrow \tau\theta + (1-\tau)\,\theta' \end{cases} \tag{3.34}$$

where $\theta \ll 1$ is a hyperparameter to set.

Lastly, regarding the exploration-exploitation trade-off, DDPG deals with it by adding a noise $\mathcal{N}$ to the actor output during training, which can either be sampled from a simple distribution or also from a random process like the Ornstein-Uhlenbeck process that was used in the original paper [33].

Algorithm 1 Pseudo Code of Deep Deterministic Policy Gradient Algorithm

Input: Initial actor parameters ϕ and initial critic parameters θ
$\phi' \leftarrow \phi, \theta' \leftarrow \theta$ ▷ Initialize target network weights
$\mathcal{D} \leftarrow \varnothing$ ▷ Initialize an empty replay buffer
for episode $= 1, \ldots, M$ **do**
    Initialize a random process $\mathcal{N}$
    $s_t \leftarrow s_1$ ▷ Receive initial state
    repeat
        Observe state s_t and select action $a_t = \boldsymbol{\pi}_\phi(s_t) + \mathcal{N}$
        Execute action a_t and store tuple $(s_t, a_t, s_{t+1}, r_{t+1})$ in $\mathcal{D}$
        Sample a random minibatch of B transitions $(s_t, a_t, s_{t+1}, r_{t+1})$ from $\mathcal{D}$
        Compute $y_t = r_{t+1}(s_t, a_t, s_{t+1}) + \gamma Q'_{\theta'}\left(s_{t+1}, \boldsymbol{\pi}'_{\phi'}(s_{t+1})\right)$
        Update the parameters θ of the critic network by minimizing:
        $L(\theta) = \frac{1}{B}\sum_i \left(Q_\theta(s_t, a_t) - y_t\right)^2$
        Update the parameters ϕ of the actor network by maximizing:
        $J_\pi(\phi) = \mathbb{E}\left[Q(s, \boldsymbol{\pi}(s))\right]$
        $\phi' \leftarrow \tau\phi + (1-\tau)\phi'$ ▷ Update target actor
        $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ ▷ Update target critic
        $s_t \leftarrow s_{t+1}$ ▷ Receive next state
    until s_t is terminal
end for
Output: Policy $\boldsymbol{\pi}_\phi$ and Q-function Q_θ with optimal parameters

SAC

The DDPG algorithm, however, can be extremely brittle, as its performance is highly dependent on the initialization of the weights of the network and on the values of the hyperparameters [34]. For this reason, a new algorithm called **Soft Actor-Critic** (SAC) has been developed [35, 36].

Just like the DDPG algorithm, SAC uses a replay buffer to store all the interactions between agent and environment for offline training. This method, however, relies on a stochastic policy, which means that, rather than outputting an action directly, the actor generates the distribution of a random variable, from which an action will be sampled. This fact implies that exploration is intrinsic to the architecture of the policy and no external noise has to be added. To remove the exploration component during testing time, the policy simply uses the expected value of the output distribution as the action to perform.

Furthermore, it does not use the standard RL strategy of training the policy network to maximize the expected sum of rewards and, instead, augments the standard objective $\mathbb{E}_{\pi}\Big[\sum_{t=0}^{\infty}\gamma^{t} r_{t+1}\left(s_t, a_t, s_{t+1}\right)\Big]$ by adding the expected entropy of the policy to the reward signals, where the entropy of a random variable x with density function P is denoted by:

$$\mathcal{H}\left(P\right) = \mathbb{E}_{x\sim P}\Big[-\log P\left(x\right)\Big] \tag{3.35}$$

and represents, in simple terms, the level of randomness of the outcome of the random variable. This new term changes the objective of the policy optimization part of the algorithm to:

$$\begin{aligned} J_{\pi}\left(\phi\right) &= \mathbb{E}_{\pi}\Big[\sum_{t=0}^{\infty}\gamma^{t}\Big(r_{t+1}\left(s_t, a_t, s_{t+1}\right) + \alpha\mathcal{H}\left(\boldsymbol{\pi}_{\phi}\left(\cdot \mid s_t\right)\right)\Big)\Big] \\ &= \mathbb{E}\Big[Q\left(s_t, a_t\right) - \alpha\log\boldsymbol{\pi}_{\phi}\left(a_t \mid s_t\right)\Big] \end{aligned} \tag{3.36}$$

where ϕ is the set of parameters of the policy network $\boldsymbol{\pi}$, $Q\left(s_t, a_t\right)$ is the current approximation of the value of the state-value pair $\left(s_t, a_t\right)$ and α is a parameter called *temperature*. Likewise, the equations for the state-value and action-value functions under policy $\boldsymbol{\pi}$ become [38]:

$$V_{\pi}\left(s\right) = \mathbb{E}_{\pi}\Big[\sum_{t=0}^{\infty}\gamma^{t}\Big(r_{t+1}\left(s_t, a_t, s_{t+1}\right) + \alpha\mathcal{H}\left(\boldsymbol{\pi}_{\phi}\left(\cdot \mid s_t\right)\right)\Big) \mid s_t = s\Big] \tag{3.37}$$

and

$$Q_{\pi}\left(s, a\right) = \mathbb{E}_{\pi}\Big[\sum_{t=0}^{\infty}\gamma^{t} r_{t+1}\left(s_t, a_t, s_{t+1}\right) + \alpha\sum_{t=1}^{\infty}\gamma^{t}\mathcal{H}\left(\boldsymbol{\pi}_{\phi}\left(\cdot \mid s_t\right)\right) \mid s_t = s, a_t = a\Big] \tag{3.38}$$

respectively, from which it is possible to prove that:

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi\Big[Q_\pi(s,a) + \alpha\mathcal{H}\left(\boldsymbol{\pi}_\phi(\cdot \mid s)\right)\Big] \\ &= \mathbb{E}_\pi\Big[Q_\pi(s,a) - \alpha\log\boldsymbol{\pi}(a \mid s)\Big] \end{aligned} \tag{3.39}$$

and

$$\begin{aligned} Q_\pi(s,a) &= \mathbb{E}_\pi\Big[r(s,a,s') + \gamma\Big(Q_\pi(s',a') + \alpha\mathcal{H}\left(\boldsymbol{\pi}_\phi(\cdot \mid s)\right)\Big)\Big] \\ &= \mathbb{E}_\pi\Big[r(s,a,s') + \gamma\Big(Q_\pi(s',a') - \alpha\log\boldsymbol{\pi}_\phi(a' \mid s')\Big)\Big] \end{aligned} \tag{3.40}$$

In order to approximate the value of Q_π, SAC makes use of two Q-value functions, denoted by their set of parameters θ_1 and θ_2. In particular, the two networks are trained simultaneously in order to satisfy equation (3.40) by minimizing:

$$L(\theta_i) = \mathbb{E}\Big[\Big(Q_{\theta_i}(s,a) - \Big(r(s,a,s') + \gamma\Big(Q(s',a') - \alpha\log\boldsymbol{\pi}_\phi(a' \mid s')\Big)\Big)\Big)^2\Big] \tag{3.41}$$

where:

$$Q(s,a) = \min_{i=1,2} Q_{\theta_i}(s,a) \tag{3.42}$$

which means that the estimate of the action-value function is given by the minimum of the outputs of the two critic networks. This is done in order to reduce the overestimation problem that plagues value-based reinforcement learning algorithms [39]

Lastly, regarding the temperature parameter, it is used to determine the importance of the entropy term with respect to the reward signal, which means that it controls the stochasticity of the policy [35] and, therefore, the trade-off between exploration and exploitation. Indeed, adding entropy to the objective function for the optimization of the policy was shown to improve exploration by discouraging premature convergence to sub-optimal deterministic policies [37]. The value of the temperature can either be fixed to a constant or learnt like the parameters of the networks, which is the preferable strategy, as the policy should be more stochastic in more uncertain regions and more deterministic in states where there is a clear separation between good and bad actions [36]. In this case, the optimization problem to solve, which was formulated in [36], is:

$$J(\alpha) = \mathbb{E}_\pi\Big[-\alpha\log\boldsymbol{\pi}(a \mid s) - \alpha\bar{\mathcal{H}}\Big] \tag{3.43}$$

Algorithm 2 Pseudo Code of Soft Actor-Critic Algorithm

Input: Initial actor parameters ϕ and initial critic parameters θ_1 and θ_2
$\theta_1' \leftarrow \theta_1, \theta_2' \leftarrow \theta_2$ ▷ Initialize target network weights
$\mathcal{D} \leftarrow \varnothing$ ▷ Initialize an empty replay buffer
for episode $= 1, \ldots, M$ **do**
  $s_t \leftarrow s_1$ ▷ Receive initial state
  repeat
    Observe state s_t and sample action $a_t \sim \boldsymbol{\pi}_\phi(\cdot \mid s_t)$
    Execute action a_t and store tuple $(s_t, a_t, s_{t+1}, r_{t+1})$ in $\mathcal{D}$
    Sample a random minibatch of B transitions $(s_t, a_t, s_{t+1}, r_{t+1})$ from $\mathcal{D}$
    Compute $Q(s,a) = \min_{i=1,2} Q_{\theta_i}(s,a)$
    Update the parameters θ_1 and θ_2 of the two critics by minimizing:
    $L(\theta_i) = \mathbb{E}\Big[\Big(Q_{\theta_i}(s,a) - \Big(r(s,a,s') + \gamma\Big(Q(s',a') - \alpha \log \boldsymbol{\pi}_\phi(a' \mid s')\Big)\Big)\Big)^2\Big]$
    Update the parameters ϕ of the actor network by maximizing:
    $J_\pi(\phi) = \mathbb{E}\Big[Q(s_t, a_t) - \alpha \log \boldsymbol{\pi}_\phi(a_t \mid s_t)\Big]$
    Update the temperature parameter by minimizing:
    $J(\alpha) = \mathbb{E}_\pi\Big[-\alpha \log \boldsymbol{\pi}(a \mid s) - \alpha\bar{\mathcal{H}}\Big]$
    $\theta_i' \leftarrow \tau\theta + (1-\tau)\theta_i' \quad i = 1{,}2$ ▷ Update target critics
    $s_t \leftarrow s_{t+1}$ ▷ Receive next state
  until s_t is terminal
end for
Output: Policy $\boldsymbol{\pi}_\phi$ and Q-functions Q_{θ_1}, Q_{θ_2} with optimal parameters

Part II

Contribution

Chapter 4

Methodology

In chapter 2, the static portfolio management problem was introduced alongside the traditional method to solve it. We then presented the reinforcement learning framework in chapter 3 in order to tackle sequential decision making problems. In this chapter, we will formulate the dynamic asset allocation problem as a discrete time dynamical system so that it is compatible with the algorithms introduced in the previous chapter. In particular, section 4.1 will list all the assumptions that were made about the trading environment and the agent during the experiments, then section 4.2 will define the mathematical framework, which is heavily inspired by [40], and lastly sections 4.3 and 4.4 will introduce all the network architectures used to represent the policy and the Q-value functions and what can be done to speed up their training.

4.1 Assumptions

In this thesis, all experiments are done exclusively via back-test trading using historical data, which means that the trading agent pretends to be at a certain point in the past and then does paper trading without using any information from the "future" [40]. Therefore, in order for this method of testing to make sense, various assumptions regarding the market have to be made, which are:

- there is no bid-ask spread , meaning that all assets can be bought and sold at the same price;
- all assets must be sufficiently liquid, so that transactions involving any asset can take place immediately and at any time as long as the market is open with no difference between the expected price and the actual one;
- buying and selling fractions of shares is possible;

- all transactions have a cost equal to 0.2% of their value;
- the amount of money the trading agent can move is not meaningful enough to have an impact on the market.

Furthermore, we also assume that the market is composed of only M assets, of which one is the risk-free cash, and that the trading agent re-balances the portfolio at the end of every trading day, which means the closing prices of the assets are used to calculate the costs.

4.2 Framework definition

In the dynamic portfolio optimization task, an investor, who covers the role of agent, continuously interact with the market, which functions as the environment of the system, by changing their investment strategy in order to maximize a predefined reward signal after having observed the behaviour of the assets in the market.

4.2.1 State and Action Spaces

At any time-step t, which in this framework refers to the moment the market closes at the end of day t, the agent can only observe the price vector $\vec{\boldsymbol{p}}_t$ of the assets included in the market, which, however, is not enough to determine the actual state of the environment [14], meaning that the environment is only partially observable. For this reason, as mentioned in section 3.1, the agent needs to build its own representation of the environment state by relying on the history of observations and interactions between agent and environment. Nevertheless, using the complete history from the start of the trading period to the current time-step t is not computationally feasible. As a middle ground, a solution employed by many [14, 40, 41, 42], which this framework will also utilize, is to use a rolling window setup in which at every time-step t only the observations from the past W time periods are considered, where W is a hyperparameter called *window length*, which is set to be equal to 50 for all of our experiments. Furthermore, since transaction costs are also included, the current portfolio weights should also be included in the environment state. As a consequence, at every time-step t, the state s_t used by the agent to decide the action a_t to perform is given by:

$$s_t = (\vec{\boldsymbol{p}}_{t-W+1:t}, \boldsymbol{w}_t) \tag{4.1}$$

where $\vec{\boldsymbol{p}}_{t-W+1:t}$ is the asset data matrix defined in (2.5) and $\boldsymbol{w}_t$ is the vector of portfolio weights at the end of day t. Since assets can, in theory, have any value as long as it is not negative, the state space is considered to be continuous.

Data Preprocessing

As mentioned before in section 2.2, working directly with asset prices is not recommended due to their non-stationarity. For this reason, before being used as input for the policy and Q-value networks, various preprocessing steps are applied to the asset data matrix $\vec{\boldsymbol{p}}_{t-W+1:t}$. Firstly, the difference of the logarithm of the prices are calculated, obtaining the log return matrix $\vec{\boldsymbol{\rho}}_{t-W+2:t}$ defined in (2.10). Then, since all transactions are done with closing prices, the log-differences of the opening prices are removed for all assets. Lastly, the remaining four features for all assets are standardized using mean and standard deviation of the training data. Using the same notation to avoid unnecessary complications, the networks' input can be expressed as:

$$s_t = (\vec{\boldsymbol{\rho}}_{t-W+2:t}, w_t) \tag{4.2}$$

Action Space

Regarding, instead, the actions performed by the agent, in this framework they refer directly to the new portfolio weights, which will be applied when the market closes for the day and will be the investment strategy until the re-balancing the next day. This means that, just like the state space, the action space is also not discrete.

4.2.2 Trading Dynamics

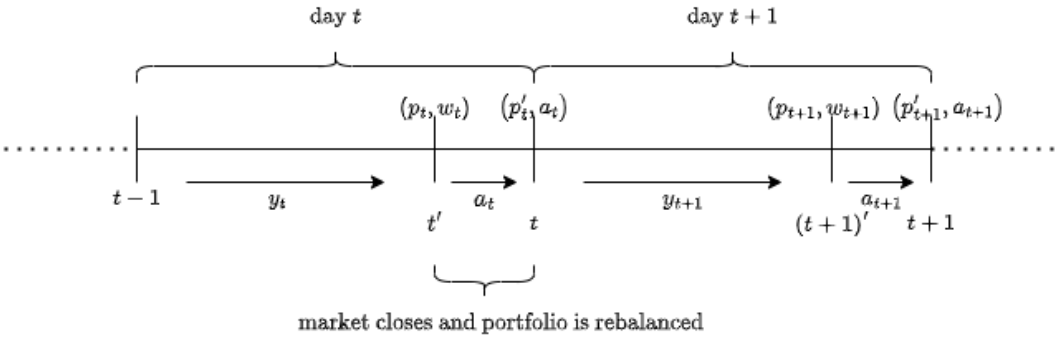


Figure 4.1: Dynamics of the portfolio

Following the assumptions made in section 4.1, the complete dynamics of the interactions between trading agent and market environment are represented by figure 4.1, which shows how the value and the weights of the portfolio change in response to the re-balancing actions of the agent and the changes in the prices of

the assets for one time period, which can also be explained by the following list of events:

1. at the end of day t (time instant t'), the market closes and the portfolio has weights w_t and a total value of p_t;

2. the agent, after having observed the price changes during day t and the current portfolio weights, decides to re-balance the portfolio by performing action a_t (time instant t), which changes the portfolio value to p'_t due to transaction costs. The new portfolio value is, therefore, given by:

$$p'_t = (1 - \mu_t)\, p_t \quad \text{with} \quad \mu_t \approx \mu \|a_t - w_t\|_1 \tag{4.3}$$

   where μ_t is the transaction cost expressed as a fraction of the portfolio value before being re-balanced, which is also used in (2.23) to express the transaction cost in the Markowitz model, and $\mu = 0.2\%$ as mentioned in section 4.1;

3. trading day $t+1$ starts and, at closing time (time instant $(t+1)'$), the portfolio has a total value of p_{t+1} with weights w_{t+1}, which are equal to:

$$\begin{cases} p_{t+1} = p'_t \,(y_{t+1} \cdot a_t) \\ w_{t+1} = \dfrac{y_{t+1} \odot a_t}{y_{t+1} \cdot a_t} \end{cases} \tag{4.4}$$

   where y_{t+1} is the vector of closing prices of the assets for day $t+1$ expressed as a fraction of the closing prices for day t, that is:

$$y_{t+1} = \frac{\boldsymbol{p}_{t+1}^{close}}{\boldsymbol{p}_{t}^{close}} \tag{4.5}$$

Regarding the first trading day, it is assumed that the starting portfolio consists of only 1\$ in cash, which will stay the same until the first closing day and the subsequent re-balancing, as the cash asset is used as the point of reference for all other assets and, therefore, does not change in value over time.

Remark It should be noted that the expression used in (4.3) to obtain the value of the portfolio after re-balancing is only an approximation. Indeed, in [40] it was proven that the exact formula for the transaction cost μ' is the solution of the following equation:

$$\mu_t = \frac{1}{1 - \mu a_{1,t}} \Big[1 - \mu w_{1,t} - \left(2\mu - \mu^2\right) \sum_{i=2}^{M} \left(w_{i,t} - \mu_t a_{i,t}\right)^+ \Big] \tag{4.6}$$

where $w_{1,t}$ and $a_{1,t}$ refer to the percentages invested in the cash asset, and that it can be solved using the fixed-point iteration method [43].

Nevertheless, figure 4.2 shows that the approximation is good enough that it does not warrant the loss in efficiency the use of this algorithm entails, as it only tends to slightly underestimate the exact value.

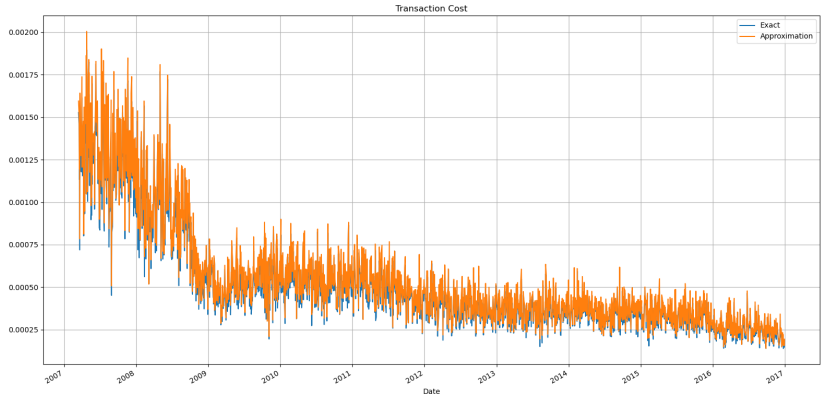


Figure 4.2: Comparison of exact and approximated transaction costs incurred by a trading agent that performs only random actions.

4.2.3 Reward Signals

Regarding the reward signals that are essential to the reinforcement learning framework, they have to be chosen so that they faithfully represent the effect of the most recent action. In order to do so, our framework takes inspiration from the various performance metrics introduced in section 2.3. In particular, three different reward signals will be used and compared: log returns, differential Sharpe ratio and differential downside deviation ratio.

Log Returns

Since the objective of RL is to maximize the sum of the rewards and that it is possible to express the value of the portfolio at time-step t using the sum of log returns using the formula in (2.12), one possible reward signal is the log return of the portfolio, that is:

$$r_{t+1} = \ln\left(\frac{v_{t+1}}{v_t}\right) = \rho_{t+1} \tag{4.7}$$

which translates to trying to maximize the final value of the portfolio without caring about the risks incurred.

Differential Sharpe Ratio

In order to include a measure of risk in the reward signal, using metrics such as the Sharpe ratio directly is not ideal, as its value is influenced by all actions performed from the start of the trading period to the current time-step and, as mentioned above, reward signals should only represent the effect of the most recent one. For this reason, one possible alternative is to use the **differential Sharpe ratio** (DSR) [44], which represents the influence on the Sharpe ratio of the return received at the current time-step and is defined as:

$$r_{t+1} \doteq \frac{B_t \left(\rho_{t+1} - A_t\right) - \frac{1}{2} A_t \left(\rho_{t+1}^2 - B_t\right)}{\left(B_t - A_t^2\right)^{3/2}} \tag{4.8}$$

with:

$$\begin{cases} A_t = A_{t-1} + \eta \left(\rho_t - A_{t-1}\right) \\ B_t = B_{t-1} + \eta \left(\rho_t^2 - B_{t-1}\right) \end{cases} \tag{4.9}$$

where η is a constant that controls the magnitude of the influence of the return on the Sharpe ratio.

Differential Downside Deviation Ratio

Just like the Sortino ratio was proposed as an alternative metric to the Sharpe ratio that takes into consideration the asymmetric preferences of most investors to price changes, the **differential downside deviation ratio** (D3R) [45] is proposed as an alternative reward signal to the differential Sharpe ratio that, as the name implies, uses the downside deviation as the risk measure. This value has the following form:

$$r_{t+1} \doteq \begin{cases} \dfrac{\rho_{t+1} - \frac{1}{2} A_t}{DD_t} & \rho_{t+1} > 0 \\ \dfrac{DD_t^2 \left(\rho_{t+1} - \frac{1}{2} A_t\right) - \frac{1}{2} A_t \rho_{t+1}^2}{DD_t^3} & \rho_{t+1} \leq 0 \end{cases} \tag{4.10}$$

with:

$$\begin{cases} A_t = A_{t-1} + \eta \left(\rho_t - A_{t-1}\right) \\ DD_t^2 = DD_{t-1}^2 + \eta \left(\left(\rho_t^+\right)^2 - DD_{t-1}^2\right) \end{cases} \tag{4.11}$$

4.3 Network Architectures

In this thesis, the neural networks used to represent the policy and value functions are heavily inspired by [40, 14, 42], in which they aim to build a universal model that reduces the agent model complexity and is able to generalize across different trading universes by exploiting the principle of parameter sharing [46].

4.3.1 Asset Value Module

In order to achieve this universality, we propose the **Asset Value Module** (AVM) shown in figure 4.3, which is a neural network that takes as input a multivariate time-series of fixed length and generates a score that summarizes its properties. In our case, the input time-series is the standardized log differences of the OHLCV data of a single asset i in a given time window $\vec{\rho}_{i,t-W+2:t}$, which, in turns, means that the generated score $v_t \in \mathbb{R}$ refers to how well the asset is doing in that period of time.

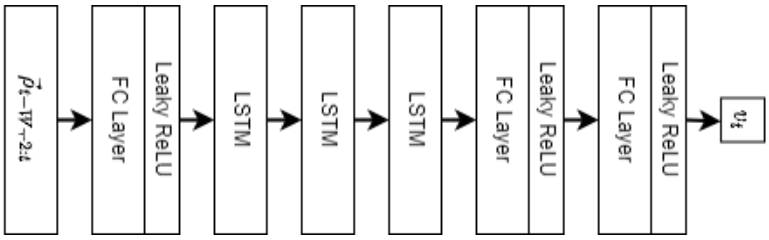


Figure 4.3: Asset Value Module architecture. The first fully connected layer construct a feature map, which are then elaborated by three LSTM layers. Lastly, two final dense layers combine the output of the last LSTM layer into a single scalar value.

Regarding the nonlinear transformations, the network uses leaky ReLUs with negative slope equal to 0.1 after every fully connected layer. Furthermore, the dimension of the feature map is a hyperparameter called *hidden size* that is kept constant throughout the whole network.

Since there are M assets in the market environment, the input matrix $\vec{\rho}_{i,t-W+2:t}$ of each asset is passed through the AVM separately and independently, which means that the parameters of the networks are shared for all assets, hence the use of the parameter sharing principle. Furthermore, since one of the M assets is the cash asset, whose price and volume are set to be constant throughout the whole trading period, the network will calculate the value of a matrix of all zeros, which can be beneficial, since, in this way, the network has a point of reference to evaluate the rest of the assets [42].

4.3.2 Actor and Critic Networks

The output of the AVM network for all assets are then combined with the current portfolio weights, which are the second part of the state of the environment, using another set of dense layers to obtain the complete deterministic policy network. In order to satisfy the constraint that the sum of the weights of the portfolio must be equal to 1, the final fully connected layer is followed by a softmax as the non-linear activation function. The critic network, instead, takes as additional input a third vector, the action a_t performed by the policy, which is then combined with the rest to output the Q-value of the state-action pair (s_t, a_t) as shown in figure 4.4.

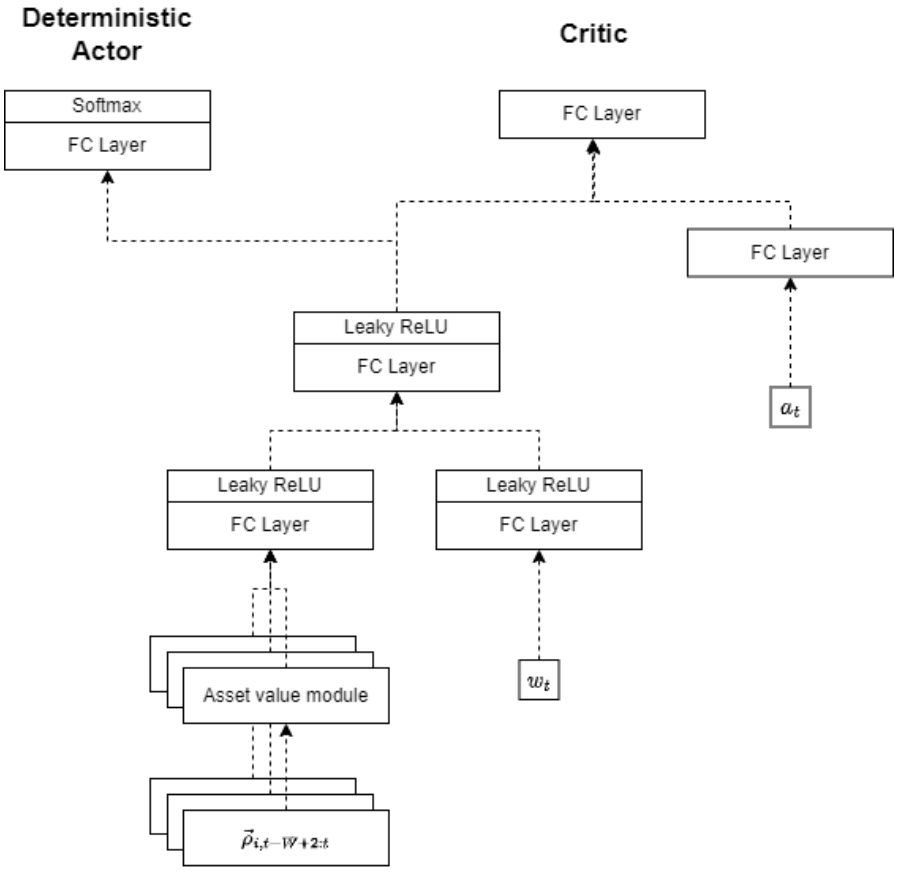


Figure 4.4: Diagram of the deterministic actor and critic networks. Note that, even if in this diagram the two networks are shown to share the initial part, this is done only for simplicity sake and that the parameters of the two networks are not actually shared. What is shared, instead, is the set of parameters of the AVM for each asset, separately for actor and critic.

Stochastic Policy Network

The two networks shown in figure 4.4 will be used in the DDPG algorithm introduced in section 3.6.1 to find the optimal policy with respect to the chosen reward signal. To use the SAC algorithm, instead, there is a need to define a network that takes the same input as the deterministic policy network, but outputs a distribution. To achieve this, inspired by the stochastic network used in [36], a slight modification to the deterministic actor is proposed: instead of a single dense layer as the output layer, it uses two separate fully connected layers, one for the mean of the distribution and the other for the standard deviation. The outputs of these two layers are then used to define a multivariate normal distribution, from which the action to perform will be sampled and, in order to satisfy the usual constraint on the portfolio weights, all components of the sampled vector are divided by their sum.

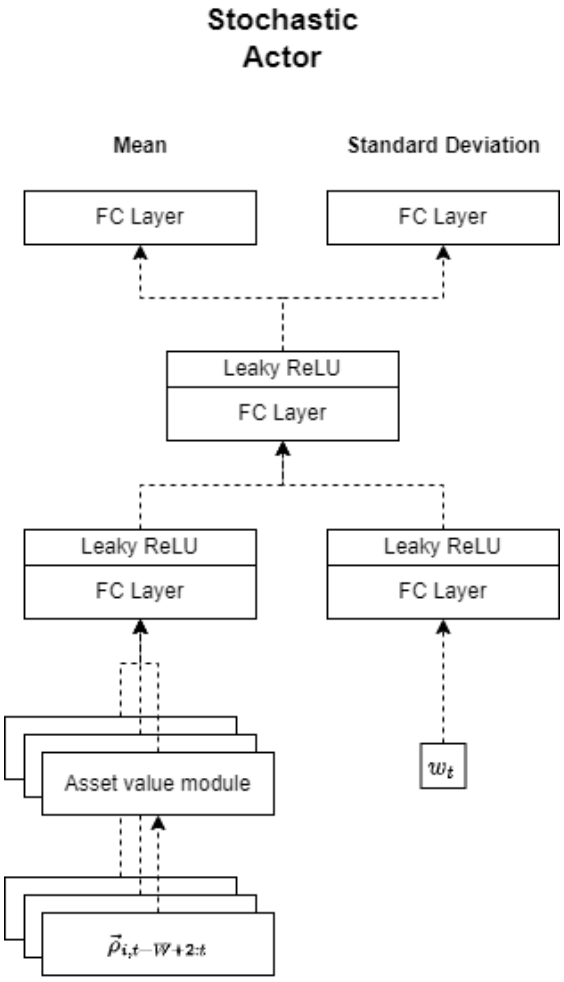


Figure 4.5: Diagram of the stochastic actor network.

4.3.3 Parameter Sharing

Until now, we have made use of the concept of parameter sharing without explaining why it is advantageous to do so, which is what this section will do, especially in the context of portfolio management. In order to explain the reasoning behind this choice, we will consider a variant of the networks proposed above, in which the AVM section of the networks is not shared and, instead, has a different set of parameters for each asset.

First of all, by using the same Asset Value Module for all assets, the total number of parameters of the network is significantly lower compared to the non-parameter-sharing variant, decreasing the probability of overfitting [47]. For the same reason, the parameters of the AVM are trained on a much larger amount of data points, which can be extremely important, as deep neural networks require a large amount of training data to obtain meaningful results. For example, if the training dataset consists of 10 years of daily OHLCV data of 5 assets, the parameter-sharing version trains its only AVM using $5 \cdot 10 \cdot 252 = 12600$ samples, while the alternative version has 5 different AVMs and each of them is trained on $10 \cdot 252 = 2520$ samples. Last but not least, the Asset Value Module is universe-agnostic, meaning that, assuming that is has been trained on a sufficient amount of data of different assets, it can also be used without training on a different set of assets. Indeed, if the actor and critic networks have already been trained on a specific market environment, when a different set of assets is given, it is possible to adapt the networks to the new data by simply freezing the parameters of the AVM and training only the following sections of the networks, drastically reducing the number of parameters to be trained [14].

4.3.4 Training

Regarding the training for the actor and critic networks, the process follows the algorithms shown in Algorithms 1 and 2 faithfully, with only a few exceptions:

- since the training data is obtained from historical observations, at the end of every episode, the environment is not reset to the oldest data point, but to a random one. The initial portfolio, instead, is always set to be equal to 1$ in cash;
- in order to use all observations an approximately equal number of times, rather than reaching the end of the trading period every episode, the number of agent-environment interactions per episode is fixed to 1000;
- since both DDPG and SAC are based on storing the interactions in a replay buffer and training the networks offline by sampling batches from it, the first episode is always spent on filling the buffer with agent-environment interactions

in which the agent performs random actions, since, otherwise, in the beginning there would not be enough samples in memory to form a complete batch.

4.4 Pre-Training

Nevertheless, despite the use of this particular architecture, training deep neural networks is still computationally expensive and require a lot of time, especially if there is a lack of a powerful GPU. For this reason, various techniques to reduce the training time are heavily used.

One such technique that is frequently used in the field of computer vision and natural language processing is the pre-training of neural networks, which consists in training the network on an auxiliary task and then fine-tune it for the actual task [49]. In particular, in the context of portfolio optimization, a sensible auxiliary task is the forecasting of future asset returns using the same standardized matrix of log returns used as input for the actor and critic networks. Indeed, the current framework only works with past data in order to decide a trading strategy for the future, which means that forecasting future values might improve the training process.

4.4.1 Forecasting

Since the main features that determines the value of the reward signal in the reinforcement learning framework are the closing prices of the assets, the forecasting task we are interested in consists in predicting the value of the vector of log returns of closing prices at the next time-step $t+1$, that is $\boldsymbol{\rho}_{t+1}^{close}$, given the matrix $\vec{\boldsymbol{\rho}}_{t-W+2:t}$ as input.

To achieve this, we simply adapted the deterministic actor network by removing the section that processes the current portfolio weights and the softmax transformation at the end of the network, since the log returns do not follow the same constraints as the weights of the portfolio, leading to the network architecture shown in figure 4.6.

Regarding the training process, it uses the same historical data used for the portfolio management task and follows the typical supervised learning approach: after having processed the training data and obtained the inputs $X_t = \vec{\boldsymbol{\rho}}_{t-W+2:t}$ and the target outputs $y_t = \boldsymbol{\rho}_{t+1}^{close}$ for all time-steps t, stochastic gradient descent or one of its variants is used to minimize a chosen loss function, which, in this case, is the **root mean squared error** (RMSE) [50]:

$$RMSE = \sqrt{\frac{\sum_{t=1}^{T} (\hat{y}_t - y_t)^2}{T}} \tag{4.12}$$

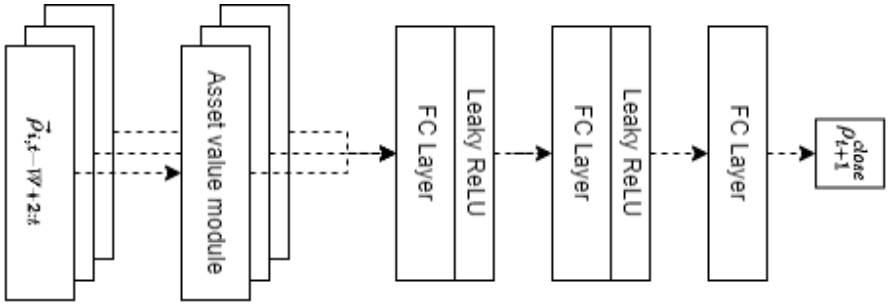


Figure 4.6: Diagram of the forecaster network.

where $\hat{y}_t$ is the prediction value. Actually, since the task is multivariate forecasting, the target outputs y_t and their predictions $\hat{y}_t$ are vectors and not scalar values. For this reason, the loss function used is calculated as the mean RMSE along the components of the vectors, which, in this case, represent the RMSE relative to the single assets.

4.4.2 Pre-Trained Networks

After having trained a neural network to forecast future log returns, the next step is to fine-tune it for the asset allocation task, which, however, is problematic, since the network used for forecasting and the actor and critic networks do not share the same architecture.

To solve this problem, we make use of the process proposed in [51], which works as follows:

1. consider only the part of the network that is shared among all networks involved, which, in this case, refers to the Asset Value Module and the first dense layer after it;
2. copy the parameters from the forecaster network to the actor and critic networks used in the RL algorithm;
3. freeze the copied parameters;
4. train the actor and critic network by optimizing only the unfrozen parameters.

In this way, the number of parameters to be optimized is heavily reduced.

Chapter 5

Implementation

This chapter will give a brief overview of how the multiple components of the framework introduced in the previous chapter were implemented in Python. This means that not all of the codebase will be explained, as otherwise it would result in too many uninteresting details that are not particularly relevant to the task at hand. In particular, only the main classes that are used to represent the agent, the environment and the training process will be presented. Indeed, this chapter is written under the premise that the reader also has the source code at hand, which can be found in the online repository of this project, in order to facilitate their understanding of the various pieces of code.

5.1 Market

As the name implies, the `Market` class handles the retrieval of historical market data from online resources. In particular, it relies on the *yfinance* package, which downloads the data using Yahoo's publicly available APIs, to access the data of all assets that belong to the trading universe in a particular period of time. In order to feed this data to the environment and the agent, the class has two methods:

- `step`: given the current time-step t, which is stored as an attribute of this class, it returns the asset data matrix $\vec{\boldsymbol{p}}_{t-W+1:t}$ and checks if the maximum number of steps has been reached;

- `reset`: it resets the current time-step to the initial value, which, as mentioned in section 4.3.4, it is a randomly chosen for each epoch of the training process.

5.2 Portfolio

The `Portfolio` class represents the environment in the reinforcement learning framework and is implemented as a subclass of the `Env` class of the *OpenAi Gym* package. Instances of this class have multiple important attributes, which are: an instance of the Market class, the current value and weights of the portfolio and the current time-step of the environment, which is shared with the step of the Market. Its methods, instead, are:

- `step`: it takes as input the action to perform and, using the market data obtained from the *step* method of the Market class, it computes the portfolio weights and value at the end of the next step just before the re-balance and returns the next state of the environment;
- `reset`: just like the `reset` method of Market, it is used to reset the environment to its initial and possible random state;
- `render`: as the name implies, when called, this method plots the trajectory of the portfolio value from the start to the current step.

In order to avoid mixing training and testing data, two instances of the Portfolio class, each of them with its own instance of the Market class, were defined.

5.3 Agent

To implement the multiple trading agents, we used a base class named `BaseAgent`, which is used to apply the reinforcement learning algorithms to train the policy networks and to produce the action to perform given the current state of the environment. To achieve this, this class stores an instance of the Portfolio class as one of its attribute to use as the training environment and defines multiple methods, which are:

- `predict_action`: it takes as input the current state of the environment and a boolean value that indicates if exploration noise has to be added and returns the action that the agent has to perform;
- `preprocess_data`: given the unprocessed asset data matrix $\vec{\boldsymbol{p}}_{t-W+1:t}$, it applies all the necessary transformations introduced in section 4.2.1;
- `get_reward`: given the interaction between agent and environment via the Portfolio's step method, it generates the appropriate reward signal;
- `train`: it applies a given reinforcement learning algorithm to train the actor and critic networks using the training environment;

- `eval`: it takes as input a second instance of the Portfolio class and uses it as a testing environment on which to evaluate the performance of the trained agent by printing various summarizing statistics and plotting the trajectory of the value and the weights of the portfolio over the testing period.

Using the BaseAgent class, multiple subclasses are defined, with each of them representing a specific agent, such as `DDPGAgent` and `SACAgent`, which represent the agents trained using DDPG and SAC respectively, and `MPTAgent`, which is an agent that does not need any training and, instead, follows the strategy defined in section 2.4.3 and simply solves the optimization problem (2.23) at every time-step. It should also be noted that, with the exception of `eval`, all other methods are redefined in the subclasses in order to add implementative details specific to the single agents. For example, the exploration noise for the DDPG algorithm is not compatible with the way SAC adds the exploration component to the actions, which means that a unique method for both of them is not feasible.

Furthermore, for the `DDPGAgent` and `SACAgent`, the neural networks used to define the various actors and critics are implemented and trained using the *PyTorch* package, which, alongside *Tensorflow* and *Keras*, is one of the most diffused open source packages for deep learning. Regarding the algorithms used to them, despite the fact that they were implemented from scratch, they are heavily inspired by the OpenAi implementations, which can be easily found online with a much more detailed documentation [52, 53]. For this reason, we will not explain the details of their implementation in this thesis.

Lastly, after multiple agents have been trained, in order to compare their performances, a new class called `AgentsEvaluator` is defined, which simply has the role of collecting the list of agents used and calling the `eval` method for all of them.

5.4 Forecaster

In regards to the training of the forecaster network introduced in section 4.4.1, the class `NNForecaster` handles the training of the network via the method `train`, which uses the data directly from a Market instance. Furthermore, in order to have a point of reference, we also implemented a more classical method of forecasting, which is based on the **Autoregressive-Moving-Average** (ARMA) model [54]. In particular, by using the *statsmodel* package, we defined the class `VARMAForecaster`, which implements a multivariate ARMA model (VARMA) and finds the optimal parameter via cross-validation, since the objective is to forecast the future returns of multiple assets at the same time.

In order to compare these two types of forecasting techniques, the method `plot_all` is used, which plots the true returns against the predicted ones and also

calculates the error both in and out of sample.

5.5 Pre-Training

As explained in section 4.4.2, the pre-training process involves copying a subset of the trained parameters of the forecaster network into the actor and critic networks and then freezing them for the fine-tuning part. In Pytorch this is easily achievable:

- for the copying part, since the set of parameters of a network is represented by a dictionary, it is sufficient to copy the key-value pairs of the section of interest;
- in order to freeze a part of the network, it is sufficient to know that all parameters of a network in Pytorch have the boolean attribute `requires_grad`, which defines if the parameter is trainable or not.

In our project, this is done in the `load_pretrained` method of both `DDPGAgent` and `SACAgent`.

5.6 Complete Framework

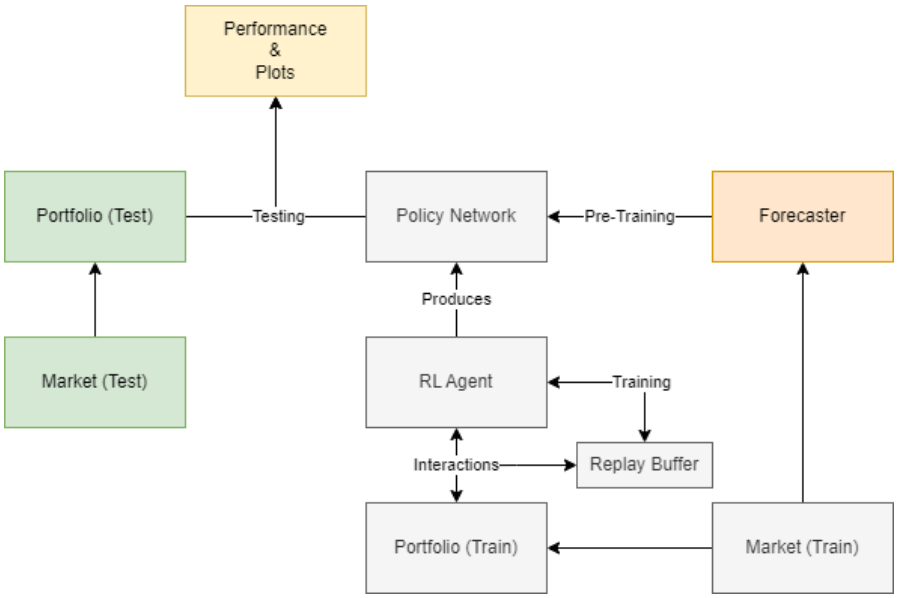


Figure 5.1: Complete framework of the training and testing process with pre-training using the forecasting task.

The complete training, testing and pre-training process can therefore be represented by the figure 5.1 and can be explained by the following steps:

1. one instance of the `Market` class downloads the market data relative to the training period and sends it to a `Portfolio` object;
2. the same market data is used in a `NNForecaster` object to train a forecaster to predict future log-returns of the assets;
3. a subset of the weights are copied into the policy and Q-value networks of the agent via the `load_pretrained` method;
4. the `train` method of the RL agent is called, which makes the agent and the portfolio interact with each other via the methods `predict_action` of the agent and `step` of the portfolio;
5. from the training, which is achieved by sampling from the replay buffer, the actor and critic networks have their parameters optimized;
6. a second instance of the `Market` class is used to download the market data relative to the testing period;
7. the actor network is used on another `Portfolio` object with the testing data to obtain the out of sample performance of the trading strategy.

Chapter 6

Experiments

In this chapter, first an introduction of the datasets used will be given, then all the results obtained from the various experiments will be shown along with some comments.

6.1 Datasets

All experiments carried out in this thesis are done using a small subsets of stocks chosen randomly from all the stocks tracked by the S&P500 index, which was created in 1957 and is regarded by many as one of the best gauge of the performance of the largest United States based companies [55]. In particular, we tested the reinforcement learning framework on two datasets, each composed by 4 stocks, that are:

- **Dataset** 1: *Apple Inc* (AAPL), *Amazon.com, Inc* (AMZN), *Costco Wholesale Corporation* (COST) and *Alphabet Inc Class A* (GOOGL);
- **Dataset** 2: *DISH Network Corp* (DISH), *Electronic Arts Inc* (EA), *Intel Corporation* (INTC) and *Marriott International Inc* (MAR).

and the cash asset (CASH) for a total of 5 assets each. Furthermore, as it should now be clear, two time intervals are used in order to create a train/test split so that the algorithms can be evaluated out of sample. In particular, our experiments use the period from 2007/01/01 to 2016/12/31 inclusive for training and the period from 2017/01/01 to 2020/12/31 inclusive for testing.

Remark 1 The reason why such a limited number of stocks was selected for each dataset is the lack of sufficient computational resources needed to manage a larger amount of data. Indeed, initial experiments were done with a set of 16 assets,

which resulted in extremely long training times of more than one hour per episode and in much smaller improvements in the performance of the resulting policy.

Remark 2 Until now we have used the terms *portfolio management* and *asset allocation* interchangeably. However, in reality, this is incorrect, since asset allocation is only one of the two steps of actual portfolio management strategies [56]:

1. *asset allocation*: the investor decides the percentages of the portfolio they would like to invest in the different types of assets, such as stocks and bonds;
2. *asset selection*: after having fixed the percentages, the investor decides the specific assets to buy.

The reason behind this imprecision is the fact that the framework introduced in this work combines the two steps by allocating the percentages directly to the single assets and not to the categories. Nevertheless, despite this difference, the results still provides valuable information regarding the applicability of reinforcement learning algorithms to the portfolio optimization problem.

6.2 Hyperparameter Setup

The hyperparameters exploited in the experiments for the DDPG and SAC algorithms are shown in tables 6.1 and 6.2 respectively, which are used for both datasets without change.
The hyperparameters used for the forecasting task, instead, are shown in table 6.3.

6.3 Results

Regarding the experiments done, we will apply DDPG and SAC algorithms and compare the results with an agent that simply solves the optimization problem (2.23) at every time-step, which we will call MPT agent from now on. In particular we will discuss the results obtained with the three reward signals proposed in section 4.2.3 and use the performance metrics introduced in section 2.3 to compare the different portfolios. Then, as a last experiment, we apply the pre-training step explained in section 4.4 and compare the results with the previous ones to see if adding a pre-training step has any effect on the training process.

6.3.1 DDPG Approach

From the results obtained using the DDPG algorithm on both trading universes, it is clear that using *log returns* or *differential Sharpe ratio* results in almost the same

Hyperparameter	Value
Actor Network	Optimizer: Adam Learning rate: 1×10^{-4} Hidden size: 64
Critic Network	Optimizer: Adam Learning rate: 1×10^{-4} Hidden size: 64
Exploration	Source: Ornstein-Uhlenbeck Process Parameters: $\mu = 0, \sigma = 0.2, \theta = 0.15$
Gamma (γ)	0.99
Tau (τ)	1×10^{-3}
Replay Memory Size	1×10000
Batch Size	64
Number of Episodes	100
Number of Steps per Episode	1000

Table 6.1: Hyperparameter setup for DDPG algorithim.

strategy in which the trading agent invests almost everything into a single asset for the whole duration of the trading period with all other assets having less than 0.1% invested into them. Moreover, the resulting strategies are basically static, since, even though the portfolio weights are not constant throughout the trading period, the amount by which they change is insignificant. Furthermore, due to the fact that these results were obtained after a single episode of training and that all subsequent episodes did not bring any change to the networks, we hypothesize that the training process gets stuck in a local optimum.

Regarding, instead, the use of the *differential downside deviation ratio* as the reward signal, we have found it to be particularly problematic. Indeed, regardless of the dataset used, the loss functions for the actor and critic networks would immediately start to diverge and overflow after a couple of episodes, causing the policy to output `NaN` as the action to perform independently of the input. Even after applying various numerical techniques such as adding a small value to the denominator to avoid division by 0 or scaling the reward signal to reduce its order of magnitude, we were unable to solve this problem. For this reason, from then on, we restricted the experiments to the other two reward signals.

Hyperparameter	Value
Actor Network	Optimizer: Adam Learning rate: 3×10^{-4} Hidden size: 64
Critic Networks	Optimizer: Adam Learning rate: 3×10^{-4} Hidden size: 64
Temperature	Optimizer: Adam Learning rate: 3×10^{-4} Initial value: 1
Gamma (γ)	0.99
Tau (τ)	1×10^{-3}
Replay Memory Size	1×10000
Batch Size	64
Number of Episodes	100
Number of Steps per Episode	1000

Table 6.2: Hyperparameter setup for SAC algorithim.

Hyperparameter	Value
Forecaster Network	Optimizer: Adam Learning rate: 1×10^{-4} Hidden size: 64
Batch Size	256
Number of Epochs	1000

Table 6.3: Hyperparameter setup for the forecasting task.

6.3.2 SAC Approach

Training the trading agent using the SAC algorithm seems to have solved some of the problems of DDPG. Indeed, figures 6.1 and 6.2 show that the agent is able to diversify its portfolio, while figures 6.2 and 6.2 show that using the differential Sharpe ratio as the reward signal results in an improved portfolio compared to using the log returns. Nevertheless, the problem of having an almost static strategy

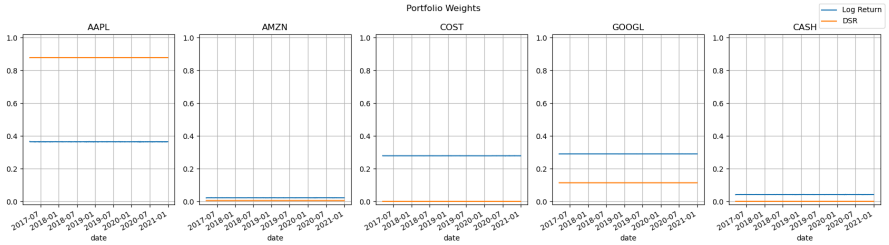


Figure 6.1: Dynamics of the portfolio weights during the testing period for SAC with *log returns* and *differential Sharpe ratio* as reward signals for dataset 1.

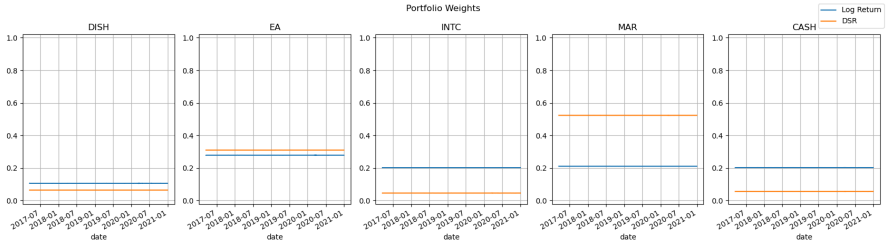


Figure 6.2: Dynamics of the portfolio weights during the testing period for SAC with *log returns* and *differential Sharpe ratio* as reward signals for dataset 2.

in which the weights of the portfolio do not change by any significant amount over time still stands.

6.3.3 Pre-Training Approach

As out last step, we try to apply pre-training to all the previous experiments in order to see whether it is able to speed up the training process and solve the problems encountered before.

First of all, we train the forecaster network on the training data of both datasets, yielding the results shown in figures 6.5 and 6.6, which are comparable to to the results obtained using the more standard multivariate ARMA forecaster shown in figures 6.7 and 6.8.

Then, as explained in section 4.4, we copy the parameters of the forecaster network into those of the actor and critic networks, freeze them and train only the remaining ones using the standard DDPG and SAC algorithms.

For DDPG, this process did not bring any kind of improvement, since the strategy simply changed to investing everything into another asset.

For SAC, instead, pre-training the networks had a positive impact on the results, as it is possible to see from figures 6.9 and 6.11, which show that the trading strategy is no longer static for either dataset, and figures 6.10 and 6.12, which

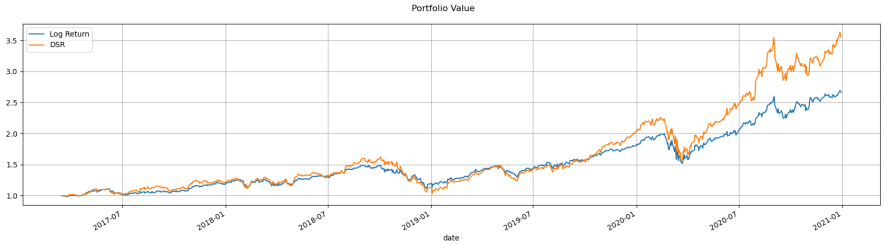


Figure 6.3: Dynamics of the portfolio weights during the testing period for SAC with *log returns* and *differential Sharpe ratio* as reward signals for dataset 1.

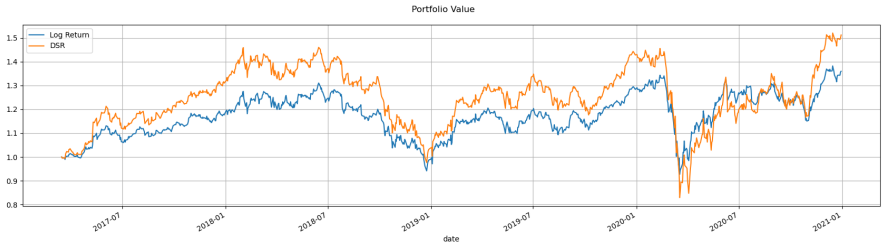


Figure 6.4: Dynamics of the portfolio weights during the testing period for SAC with *log returns* and *differential Sharpe ratio* as reward signals for dataset 2.

show the superiority of using differential Sharpe ratio instead of log returns as the reward signal for the SAC algorithm.

6.3.4 Comparison

Tables 6.4 and 6.5 present the performance metrics of the portfolios obtained using DDPG and SAC on the two datasets with and without pre-training and compare them with the MPT agent, which show the superiority of reinforcement learning based methods compared to an agent based on a one-step optimization problem.

Furthermore, comparing the metrics between the methods with and without pre-training, it is possible to notice that overall it had a positive effect on the performance of the portfolios with the exception of the DDPG algorithm on the first dataset, which is the only situation in which the performance dropped. Moreover, we also noticed a decrease in the training time for both algorithms: for DDPG, the training time dropped from 1 minute per episode to about 40 seconds, while for SAC the time required went from 80 seconds to 1 minute per episode.

Lastly, it should be noted that, even though DDPG seems to perform better than SAC, especially on the first dataset, this is most likely due to the fact that agents obtained from DDPG always invest everything into one asset combined with the presence in the dataset of an asset that performs much better than the others.

Figure 6.5: Results of training the forecaster network on dataset 1. The green section is in-sample, while the red section is out-of-sample.

Figure 6.6: Results of training the forecaster network on dataset 2.

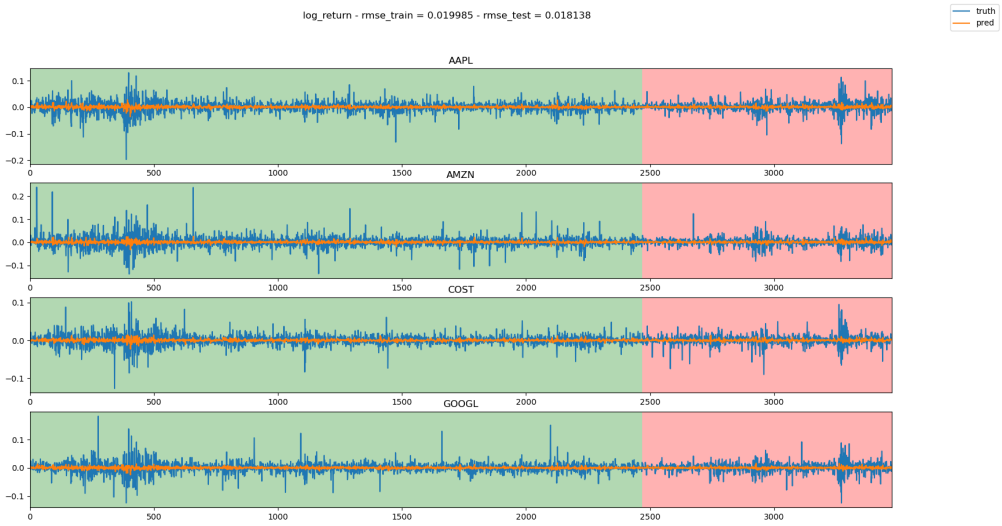


Figure 6.7: Results of training the multivariate ARMA forecaster on dataset 1.

Figure 6.8: Results of training the multivariate ARMA forecaster on dataset 2.

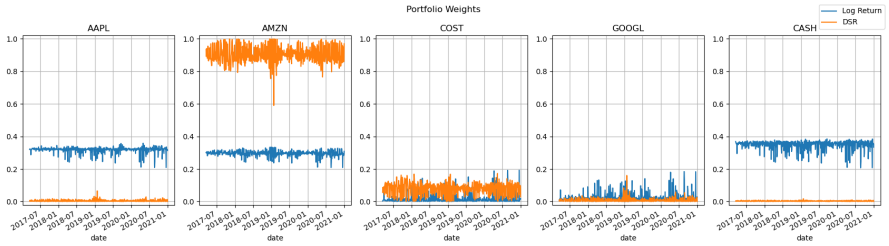


Figure 6.9: Dynamics of the portfolio weights for SAC with pre-training and with *log returns* and *differential Sharpe ratio* as reward signals for dataset 1.

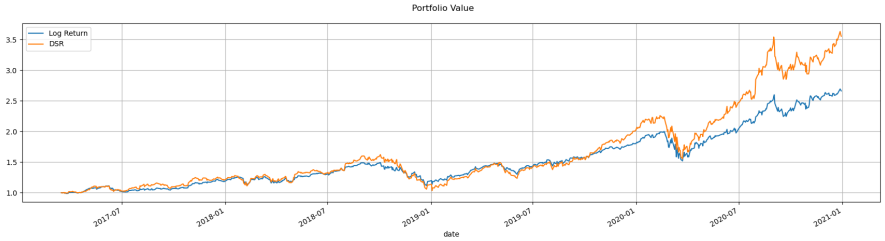


Figure 6.10: Dynamics of the portfolio weights for SAC with pre-training and with *log returns* and *differential Sharpe ratio* as reward signals for dataset 1.

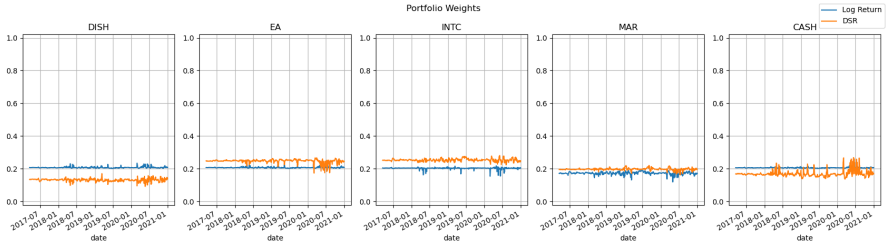


Figure 6.11: Dynamics of the portfolio values for SAC with pre-training and with *log returns* and *differential Sharpe ratio* as reward signals for dataset 2.

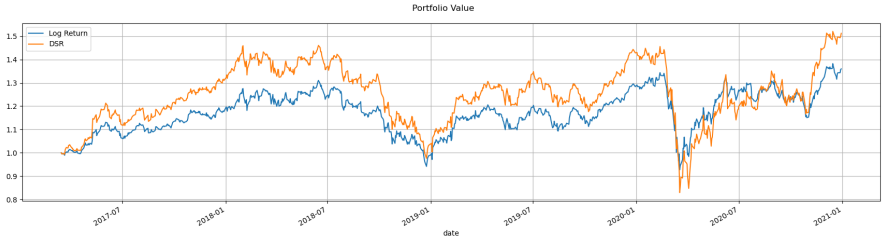


Figure 6.12: Dynamics of the portfolio values for SAC with pre-training and with *log returns* and *differential Sharpe ratio* as reward signals for dataset 2.

Method	FPV	σ	SR	SorR	MDD	V@R$_{0.95}$	CV@R$_{0.95}$
DDPG w/ Log Ret	3.81	0.020	0.081	0.12	0.34	0.031	0.044
(Above) + PT	3.75	0.020	0.078	0.12	0.39	0.030	0.048
DDPG w/ DSR	3.81	0.020	0.081	0.12	0.34	0.031	0.044
(Above) + PT	2.01	0.018	0.050	0.071	0.31	0.027	0.044
SAC w/ Log Ret	2.26	0.012	0.080	0.12	0.23	0.019	0.028
(Above) + PT	2.67	0.014	0.078	0.11	0.25	0.022	0.036
SAC w/ DSR	3.22	0.019	0.075	0.11	0.33	0.029	0.043
(Above) + PT	3.53	0.019	0.078	0.11	0.36	0.029	0.046
MPT	1.88	0.011	0.064	0.096	0.24	0.016	0.028

Table 6.4: Results obtained on dataset 1. PT stands for pre-training.

Method	FPV	σ	SR	SorR	MDD	V@R$_{0.95}$	CV@R$_{0.95}$
DDPG w/ Log Ret	1.42	0.023	0.027	0.039	0.36	0.030	0.053
(Above) + PT	1.50	0.025	0.029	0.044	0.61	0.034	0.059
DDPG w/ DSR	1.42	0.023	0.027	0.039	0.36	0.030	0.053
(Above) + PT	1.50	0.025	0.029	0.044	0.61	0.034	0.059
SAC w/ Log Ret	1.22	0.014	0.023	0.031	0.33	0.020	0.033
(Above) + PT	1.38	0.013	0.032	0.044	0.31	0.019	0.032
SAC w/ DSR	1.34	0.014	0.029	0.040	0.33	0.020	0.034
(Above) + PT	1.53	0.017	0.035	0.050	0.43	0.026	0.040
MPT	1.17	0.017	0.018	0.027	0.32	0.020	0.039

Table 6.5: Results obtained on dataset 2. PT stands for pre-training.

Chapter 7

Conclusion

The ever-growing interest in reinforcement learning approaches to solving many real-world applications combined with recent advancement in the field of deep learning has motivated the work done in this thesis. Indeed, reinforcement learning introduces a new framework to solve dynamic decision making problems, which were usually solved using more standard mathematical optimization methods.

7.1 Contributions

In order to solve the portfolio management problem with this novel framework, inspired by works like [14] and [40], we first modeled the trading universe as a discrete time dynamical system. Then, we proposed a neural network architecture that was able to process the market data while keeping in mind the overall computational complexity. Furthermore, using state of the art reinforcement learning algorithms, we were able to produce competitive results compared to more standard approaches, even though we had to limit the experiments to small datasets with a very limited number of assets. Lastly, we proposed a pre-training step using the forecasting task in order to try to improve the previous results, which led to a promising outcome, since the performance metrics indicated that this step had an overall positive effect on the training of the agents.

Nevertheless, all the results obtained in this work should be carefully considered, since the methods have only been tested via back-testing, meaning that there is no guarantee that the same results will be achieved in a live-trading environment.

7.2 Future Work

First of all, one possible extension of this work is to check, using a more powerful GPU, if the results obtained in this work still hold for larger environments or if, by

increasing the number of stocks in the market, the algorithms used are no longer able to obtain competitive results.

Secondly, all experiemnts were carried out used only the OHLCV data of the assets, which is very limited compared to the total amount of information available online regarding the various assets. Indeed, adding other sources of information such as news data to the reinforcement learning framework can be an interesting extension.

Lastly, more research should be done regarding the explainability of the resulting neural networks. Indeed, right now the agent is treated as a black-box, which is the complete opposite of actual investment strategies where being able to explain the reasoning behind the various actions is fundamental.

Bibliography

[1] Harry Markowitz. «PORTFOLIO SELECTION». In: *The Journal of Finance* 7.1 (Mar. 1952), pp. 77–91. DOI: 10.1111/j.1540-6261.1952.tb01525.x. URL: https://doi.org/10.1111/j.1540-6261.1952.tb01525.x (cit. on p. 12).

[2] William F. Sharpe. «Mutual Fund Performance». In: *The Journal of Business* 39.1 (1966), pp. 119–138. ISSN: 00219398, 15375374. URL: http://www.jstor.org/stable/2351741 (visited on 11/06/2022) (cit. on p. 10).

[3] Frank A. Sortino and Lee N. Price. «Performance Measurement in a Downside Risk Framework». In: *The Journal of Investing* 3.3 (Aug. 1994), pp. 59–64. DOI: 10.3905/joi.3.3.59. URL: https://doi.org/10.3905/joi.3.3.59 (cit. on p. 10).

[4] Philippe Artzner, Freddy Delbaen, Jean-Marc Eber, and David Heath. «Coherent Measures of Risk». In: *Mathematical Finance* 9.3 (July 1999), pp. 203–228. DOI: 10.1111/1467-9965.00068. URL: https://doi.org/10.1111/1467-9965.00068 (cit. on p. 11).

[5] Malik Magdon-Ismail, Amir F. Atiya, Amrit Pratap, and Yaser S. Abu-Mostafa. «On the maximum drawdown of a Brownian motion». In: *Journal of Applied Probability* 41.1 (2004), pp. 147–161. DOI: 10.1239/jap/1077134674 (cit. on p. 11).

[6] Mikkel Rasmussen. *Quantitative Portfolio Optimisation, Asset Allocation and Risk Management: A Practical Guide to Implementing Quantitative Investment Theory*. Springer, 2002 (cit. on p. 14).

[7] Gérard Cornuéjols, Javier Peña, and Reha Tütüncü. *Optimization Methods in Finance*. 2nd ed. Cambridge University Press, 2018. DOI: 10.1017/9781107297340 (cit. on p. 14).

[8] David G Luenberger. *Investment Science*. en. New York, NY: Oxford University Press, June 1997 (cit. on p. 5).

[9] David Silver. *Lectures on Reinforcement Learning*. 2015. URL: https://www.davidsilver.uk/teaching/ (cit. on p. 17).

[10] Richard S Sutton and Andrew G Barto. *Reinforcement Learning*. 2nd ed. Adaptive Computation and Machine Learning series. Cambridge, MA: Bradford Books, Nov. 2018 (cit. on pp. 16, 21, 23, 25).

[11] Christopher J. C. H. Watkins and Peter Dayan. «Q-learning». In: *Machine Learning* 8.3 (May 1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698 (cit. on p. 25).

[12] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. «Reinforcement learning: A survey». In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285 (cit. on p. 25).

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cit. on p. 28).

[14] Angelos Filos. *Reinforcement Learning for Portfolio Management*. 2019. DOI: 10.48550/ARXIV.1909.09571. URL: https://arxiv.org/abs/1909.09571 (cit. on pp. 15, 28, 41, 46, 49, 67).

[15] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. «An Introduction to Deep Reinforcement Learning». In: *Foundations and Trends® in Machine Learning* 11.3-4 (2018), pp. 219–354. DOI: 10.1561/2200000071. URL: https://doi.org/10.1561/2200000071 (cit. on p. 26).

[16] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738 (cit. on pp. 26, 27).

[17] Sepp Hochreiter. «The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions». In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 06.02 (1998), pp. 107–116. DOI: 10.1142/S0218488598000094. eprint: https://doi.org/10.1142/S0218488598000094. URL: https://doi.org/10.1142/S0218488598000094 (cit. on p. 29).

[18] Sepp Hochreiter and Jürgen Schmidhuber. «Long Short-term Memory». In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735 (cit. on p. 29).

[19] Ryan T. J. J. *LSTMs explained: A complete, technically accurate, conceptual guide with keras*. July 2021. URL: https://medium.com/analytics-vidhya/lstms-explained-a-complete-technically-accurate-conceptual-guide-with-keras-2a650327e8f2 (cit. on p. 29).

[20] *Part 3: Intro to policy optimization*. URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html (cit. on p. 30).

[21] David Saad. *On-Line Learning in Neural Networks*. Jan. 1999. ISBN: 9780521652636 (cit. on p. 30).

[22] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980 (cit. on p. 31).

[23] Ilya Loshchilov and Frank Hutter. *Decoupled Weight Decay Regularization*. 2017. DOI: 10.48550/ARXIV.1711.05101. URL: https://arxiv.org/abs/1711.05101 (cit. on p. 31).

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. *Playing Atari with Deep Reinforcement Learning*. 2013. DOI: 10.48550/ARXIV.1312.5602. URL: https://arxiv.org/abs/1312.5602 (cit. on pp. 31, 33).

[25] Juntao Gao, Yulong Shen, Jia Liu, Minoru Ito, and Norio Shiratori. *Adaptive Traffic Signal Control: Deep Reinforcement Learning Algorithm with Experience Replay and Target Network*. 2017. DOI: 10.48550/ARXIV.1705.02755. URL: https://arxiv.org/abs/1705.02755 (cit. on p. 33).

[26] Long-Ji Lin. «Self-improving reactive agents based on reinforcement learning, planning and teaching». In: *Machine Learning* 8.3 (May 1992), pp. 293–321. ISSN: 1573-0565. DOI: 10.1007/BF00992699. URL: https://doi.org/10.1007/BF00992699 (cit. on p. 34).

[27] *Part 2: Kinds of RL algorithms*. URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html (cit. on p. 32).

[28] *Soft actor-critic*. URL: https://spinningup.openai.com/en/latest/algorithms/sac.html.

[29] Csaba Szepesvari. *Algorithms for Reinforcement Learning*. Morgan and Claypool Publishers, 2010. ISBN: 1608454924 (cit. on p. 32).

[30] Chi Jin, Zeyuan Allen-Zhu, Sebastien Bubeck, and Michael I. Jordan. *Is Q-learning Provably Efficient?* 2018. DOI: 10.48550/ARXIV.1807.03765. URL: https://arxiv.org/abs/1807.03765 (cit. on p. 32).

[31] Vijay Konda and John Tsitsiklis. «Actor-Critic Algorithms». In: *Society for Industrial and Applied Mathematics* 42 (Apr. 2001) (cit. on p. 33).

[32] Luis A. Garrido, Rajiv Nishtala, and Paul Carpenter. «Continuous-Action Reinforcement Learning for Memory Allocation in Virtualized Servers». In: *Lecture Notes in Computer Science*. Springer International Publishing, 2019, pp. 13–24. DOI: 10.1007/978-3-030-34356-9_2. URL: https://doi.org/10.1007/978-3-030-34356-9_2 (cit. on p. 33).

[33] 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*. 2015. DOI: 10.48550/ARXIV.1509.02971. URL: https://arxiv.org/abs/1509.02971 (cit. on pp. 33, 34).

[34] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. *Benchmarking Deep Reinforcement Learning for Continuous Control*. 2016. DOI: 10.48550/ARXIV.1604.06778. URL: https://arxiv.org/abs/1604.06778 (cit. on p. 36).

[35] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. 2018. DOI: 10.48550/ARXIV.1801.01290. URL: https://arxiv.org/abs/1801.01290 (cit. on pp. 36, 37).

[36] Tuomas Haarnoja et al. *Soft Actor-Critic Algorithms and Applications*. 2018. DOI: 10.48550/ARXIV.1812.05905. URL: https://arxiv.org/abs/1812.05905 (cit. on pp. 36, 37, 48).

[37] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. «Asynchronous Methods for Deep Reinforcement Learning». In: (2016). DOI: 10.48550/ARXIV.1602.01783. URL: https://arxiv.org/abs/1602.01783 (cit. on p. 37).

[38] Pietro Michiardi, Elena Baralis, and Piero Macaluso. «Deep Reinforcement Learning for Autonomous Systems». In: () (cit. on p. 36).

[39] Scott Fujimoto, Herke van Hoof, and David Meger. *Addressing Function Approximation Error in Actor-Critic Methods*. 2018. DOI: 10.48550/ARXIV.1802.09477. URL: https://arxiv.org/abs/1802.09477 (cit. on p. 37).

[40] Zhengyao Jiang, Dixing Xu, and Jinjun Liang. *A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem*. 2017. DOI: 10.48550/ARXIV.1706.10059. URL: https://arxiv.org/abs/1706.10059 (cit. on pp. 40, 41, 43, 46, 67).

[41] Pengqian Yu, Joon Sern Lee, Ilya Kulyatin, Zekun Shi, and Sakyasingha Dasgupta. *Model-based Deep Reinforcement Learning for Dynamic Portfolio Optimization*. 2019. DOI: 10.48550/ARXIV.1901.08740. URL: https://arxiv.org/abs/1901.08740 (cit. on p. 41).

[42] Carlos Betancourt and Wen-Hui Chen. «Deep reinforcement learning for portfolio management of markets with a dynamic number of assets». In: *Expert Systems with Applications* 164 (Feb. 2021), p. 114002. DOI: 10.1016/j.eswa.2020.114002. URL: https://doi.org/10.1016/j.eswa.2020.114002 (cit. on pp. 41, 46).

[43] Richard L Burden and J Douglas Faires. «Numerical Analysis». In: 9th ed. Florence, AL: Cengage Learning, July 2010, pp. 56–64 (cit. on p. 44).

[44] John E Moody, Matthew Saffell, Y Liao, and L Wu. «Reinforcement Learning for Trading Systems and Portfolios.» In: *KDD*. 1998, pp. 279–283 (cit. on p. 45).

[45] John Moody and Matthew Saffell. «Learning to trade via direct reinforcement». In: *IEEE transactions on neural Networks* 12.4 (2001), pp. 875–889 (cit. on p. 45).

[46] Devendra Sachan and Graham Neubig. «Parameter Sharing Methods for Multilingual Self-Attentional Translation Models». In: *Proceedings of the Third Conference on Machine Translation: Research Papers*. Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 261–271. DOI: 10.18653/v1/W18-6327. URL: https://aclanthology.org/W18-6327 (cit. on p. 46).

[47] Xue Ying. «An Overview of Overfitting and its Solutions». In: *Journal of Physics: Conference Series* 1168.2 (Feb. 2019), p. 022022. DOI: 10.1088/1742-6596/1168/2/022022. URL: https://dx.doi.org/10.1088/1742-6596/1168/2/022022 (cit. on p. 49).

[48] Kevin P Murphy. *Machine Learning*. en. Adaptive Computation and Machine Learning series. London, England: MIT Press, Aug. 2012.

[49] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2013. DOI: 10.48550/ARXIV.1311.2524. URL: https://arxiv.org/abs/1311.2524 (cit. on p. 50).

[50] Rob J. Hyndman and Anne B. Koehler. «Another look at measures of forecast accuracy». In: *International Journal of Forecasting* 22.4 (2006), pp. 679–688. ISSN: 0169-2070. DOI: https://doi.org/10.1016/j.ijforecast.2006.03.001. URL: https://www.sciencedirect.com/science/article/pii/S0169207006000239 (cit. on p. 50).

[51] Tianqi Wang and Dong Eui Chang. *Improved Reinforcement Learning through Imitation Learning Pretraining Towards Image-based Autonomous Driving*. 2019. DOI: 10.48550/ARXIV.1907.06838. URL: https://arxiv.org/abs/1907.06838 (cit. on p. 51).

[52] *Source code for spinup.algos.pytorch.ddpg.ddpg*. URL: https://spinningup.openai.com/en/latest/_modules/spinup/algos/pytorch/ddpg/ddpg.html (cit. on p. 54).

[53] *Source code for spinup.algos.pytorch.sac.sac*. URL: https://spinningup.openai.com/en/latest/_modules/spinup/algos/pytorch/sac/sac.html (cit. on p. 54).

[54] George E. P. Box, Gwilym M. Jenkins, and Gregory C. Reinsel. *Time Series Analysis*. Nov. 1994. ISBN: 9780130607744 (cit. on p. 54).

[55] *S&P 500®*. URL: https://www.spglobal.com/spdji/en/indices/equity/sp-500/#overview (cit. on p. 57).

[56] Emmanouil Platanakis, Charles Sutcliffe, and Xiaoxia Ye. «Horses for courses: Mean-variance for asset allocation and 1/N for stock selection». In: *European Journal of Operational Research* 288.1 (2021), pp. 302–317 (cit. on p. 58).