Reinforcement Learning Techniques for Optimal Control in Financial Markets

Department of Economics and Finance

Master's Degree in Finance

Chair of Empirical Finance

Supervisor:

Prof. Antonio Simeone

Co-Supervisor:

Prof. Stefano Marzioni

Candidate:

Pietro Passarello

757201

Academic Year 2022/2023

To my family and my friends, who supported me throughout this journey.

Contents

List of Algorithms

List of Figures

List of Tables

Introduction

In today's fast-paced financial markets, the quest for superior trading strategies has intensified, driven by the ever-increasing complexity and volatility of global financial systems. Traders and financial institutions are continually exploring innovative methods to gain a competitive edge. One such innovation gaining traction in recent years is the application of reinforcement learning (RL) techniques to devise trading strategies.

This work embarks on a comprehensive exploration of the use of reinforcement learning for trading strategies, offering insights into the technical foundations, approximate solution methods, and practical methodology associated with this exciting and evolving field.

The motivation of this work is the great expansion that has characterized the world of RL in the past years, such as the Artificial Intelligence (AI) developed by DeepMind, Silver et al. (2016), that managed to defeat one of the best players of the Chinese game Go. What makes the game of Go particularly challenging to learn is the number of legal plays available: in fact, it has been calculated to be approximately 2.1×10^{170}. The astonishing results of RL continued with incredible achievements on all Atari games, presented in Mnih et al. (2013), where the Agent managed to establish new records and beat all human benchmarks. The achievements obtained in these applications have caused a rising interest in the applications of RL in all other fields. In the field of Finance, specifically in

Algorithmic Trading, the application of RL methods is extremely appealing: in fact, with the proper formulation, one can exploit anomalies and predict trends in financial time series using the power of Artificial Neural Networks, that can capture patterns within data that are not visible to the naked eye. Furthermore, an RL agent does not need labeled data, since its learning process is guided by the interaction with the environment, thus we are able to build a model that is completely uncorrelated with the views and thoughts of its developer. These characteristics make it so that when these techniques are applied correctly, RL Agents manage to outperform the benchmark in the field of finance, as in Huang (2018), yielding superior trading strategies.

In this work, we decided to experiment with a state-of-the-art RL technique, called Proximal Policy Optimization, Schulman et al. (2017b), in the field of Algorithmic Trading: in fact, we developed from scratch a Python implementation of the algorithm, which was later applied to empirical data from the stock markets, yielding our experimental results.

The foundation of this work is rooted in a strong technical background, as it is essential to comprehend the underlying principles and concepts governing reinforcement learning for trading. This journey begins with an in-depth examination of Markov decision processes (MDPs) in Section 1.1, where we delve into topics such as rewards, returns, policies, and value functions. We explore the notion of an optimal policy and value function, setting the stage for subsequent discussions on dynamic programming techniques, Section 1.2. Fixed point theory, Bellman policy operators, policy evaluation, policy improvement, and value iteration are the essential components of this crucial phase in our exploration.

Section 1.3 delves into the application of reinforcement learning in prediction and control to bridge the gap between theory and practical implementation. Here, we introduce Monte Carlo methods and Temporal Difference (TD) learning, which

serve as the building blocks for our journey into approximate solution methods in Chapter 2. We investigate various approaches such as function approximation, including value function approximation and linear function approximation. This work also explores the power of neural networks in nonlinear function approximation.

The methodology in Chapter 3 of our work provides a blueprint for implementing reinforcement learning-based trading strategies. This includes defining the model, data processing techniques, and a comprehensive overview of performance measures that enable us to assess the effectiveness of our strategies.

As we venture further into our work, Chapter 4 presents the results of our research. Preliminary remarks set the stage for a detailed data summary, followed by a discussion on hyperparameters, training processes, and, most importantly, on the test results that showcase the practical viability of reinforcement learning-based trading strategies.

In conclusion, this work embarks on an extensive exploration of reinforcement learning's application in the domain of trading strategies. By navigating through the technical foundations, approximate solution methods, and a comprehensive methodology, we aim to provide a comprehensive understanding of how reinforcement learning can revolutionize trading practices in the dynamic world of finance. Through rigorous analysis and empirical evidence, we aim to contribute to the growing body of knowledge in this field and provide insights that can empower traders and financial institutions in their quest for superior performance and profitability.

Chapter 1

Technical Background

1.1 Markov Decision Processes

A Markov Decision Process (MDP), Bellman (1957b), is a formalization of sequential decision-making, where actions influence both immediate rewards and subsequent states.

An MDP is a straightforward framing of the problem of learning from interaction to achieve a goal. In the context of an MDP, the learner and decision maker is called the *agent*, while everything that the agent interacts with is called the *environment*. Upon selecting an action, the agent receives a reward from the environment, which is a numerical value that the agent seeks to maximize over time through its actions.

In a more specific way, the interaction between the agent and the environment can be thought of as a sequence of discrete time-steps $t = 0, 1, 2, \ldots$, where at each time step the agent receives an observation that represents the environment's state $S_t \in \mathcal{S}$. Based on such observation, the agent selects an action $A_t in \mathcal{A}$, receiving a reward $R_t \in \mathcal{R}$. The environment then evolves to a new state S_{t+1} based on the previous state and the performed action. This procedure generates a *trajectory*

that evolves in the following way

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots.$$

We can now formally define a Markov Decision Process

Definition 1.1 (Markov Decision Process)**.** *A Markov Decision Process is defined by the tuple* $(\mathcal{S}, \mathcal{A}, P, R, \gamma, \mu)$*:*

- $\mathcal{S}$ *is a continuous or finite set of states;*
- $\mathcal{A}$ *is a continuous or finite set of actions;*
- $\mathbb{P}$ *is the transition probability function, that represents the probability of transitioning to the state* $s' \in \mathcal{S}$ *after taking action* $a \in \mathcal{A}$ *from state* $s \in \mathcal{S}$*;*
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ *is a reward function that, for each state-action pair* $(s, a) \in \mathcal{S} \times \mathcal{A}$ *returns a scalar value;*
- $\gamma \in [0, 1]$ *is a discount factor;*
- μ *is the distribution of the initial state.*

An MDP is said to be *finite* when the sets of states, the set of actions, and the set of rewards all have a finite number of elements. In this case, the random variables S_t and R_t have well-defined discrete probability distributions depending only on the preceding state and action, thus satisfying the Markov Property: that is, for particular values of these random variables, $s' \in \mathcal{S}$ and $r \in \mathcal{R}$, there is a probability of those values occurring at time t, given particular values of the preceding state and action; such a probability is given by the function p, which is said to define the *dynamics* of the MDP and is a function

$$p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1],$$

defined as

$$p(s', s|r, a) := \mathbb{P}\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}. \tag{1.1}$$

In an MDP, the probability given by p completely characterizes the environment's dynamics and thus has the *Markov Property.*

Definition 1.2 (Markovian Process)**.** *A process is said to be Markovian if and only if*

$$\mathbb{P}\{S_t|A_{t-1}, S_{t-1}, A_{t-2}, S_{t-2}, \dots\} = \mathbb{P}\{S_t|A_{t-1}, S_{t-1}\}.$$

We now define some convenient functions that can be derived from the dynamics function p, the first of which is the *state-transition probabilities* can be computed as a three argument function $P : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$, defined as:

$$P(s'|s, a) := \mathbb{P}\{S_t = s'|S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r|s, a). \tag{1.2}$$

We then have the expected reward for state-action pairs as a two-argument function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, defined as:

$$r(s, a) := \mathbb{E}[R_t|S_{t_1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a), \tag{1.3}$$

and the expected rewards for state-action-next-state triples as a three-argument function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, defined as:

$$r(s, a, s') := \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a, S_t = s] = \sum_{r \in \mathcal{R}} r \frac{p(s', r|s, a)}{s'|s, a}. \tag{1.4}$$

1.1.1 Rewards and Returns

The goal of the agent is to maximize the reward, or, to be more precise, to maximize the expected return, Sutton, Barto (2018), where the return is the sum of the reward between the time steps:

$$G_t = R_{t+1} + R_{t+2} + ... + R_T, \tag{1.5}$$

where R_T is the reward at the final time step T, in this case, the MDP is said to be *finite*, and the MDP task is said to be *episodic*. When it is not possible to define a terminal state, the MDP task is denoted as *continuous* and the final time step is equal to $T = \infty$. In this case, we use the discounted return, defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{1.6}$$

It must be noted that the discount factor γ represents how much we would weight the immediate reward compared to future rewards: in fact, a γ close to 0 will lead to a myopic agent and a γ close to 1 will lead to a farsighted agent. Furthermore, it is useful to notice that the discounted reward function in Equation 1.6 can also be expressed in a recursive relation as:

$$G_t = R_{t+1} + \gamma G_{t+1}. \tag{1.7}$$

1.1.2 Policy

In the general case, we can assume that the agent will perform a random action A_t, according to the probability distribution function of the current state S_t. We refer to this function as a *Policy*, which formally is a function, Rao, Jelvis (2022),

$$\pi : \mathcal{S} \times \mathcal{A} \to [0, 1],$$

defined as:

$$\pi(s, a) = \mathbb{P}\{A_t = a | S_t = s\},\ \forall s \in \mathcal{S}, a \in \mathcal{A}, \tag{1.8}$$

such that

$$\sum_{a \in \mathcal{A}} \pi(s, a) = 1, \forall s \in \mathcal{S}.$$

When we have a policy such that the action probability distribution for each state is concentrated on a single action, we refer to it as a *deterministic policy*, which is

a function

$$\pi_D : \mathcal{S} \to \mathcal{A},$$

such that

$$\pi(s, \pi_D(s)) = 1, \text{ and } \pi(s, a) = 0, \forall a \in \mathcal{A} \text{ with } a \neq \pi_D(s).$$

It is also worth pointing out that policy is said to be *stationary* if it is invariant of time t, otherwise, we say that the policy is *non-stationary*.

1.1.3 Value and Action-Value Function

The *value function* is a function that is used to estimate *how good* it is for the agent to be in a given state; clearly, the rewards the agent can expect to receive in the future, and thus the *goodness* of the current state, depend on the actions that it will take. For this reason, value functions are defined with respect to a policy.

Definition 1.3 (Value Function)**.** *The Value Function for an MDP evaluated with fixed policy π is a function*

$$V^\pi : \mathcal{S} \to \mathbb{R},$$

defined as:

$$V^\pi(s) = \mathbb{E}_\pi \left[G_t | S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right], \forall s \in \mathcal{S}. \tag{1.9}$$

In a similar fashion, we can define the value of taking action a in state s under policy π as the expected return starting from s, taking action a, and thereafter following policy π.

Definition 1.4 (Action-Value Function)**.** *The action-value function is a function*

$$Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R},$$

defined as:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[G_t|S_t = s, A_t = a] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s, A_t = s\right], \forall s \in \mathcal{S}. \tag{1.10}$$

It must be pointed out that an important property of value functions is that they satisfy recursive relationships as the one used in Equation 1.7; in fact, for any policy π and any state s, the following consistency condition holds:

$$\begin{aligned} V^{\pi}(s) &:= \mathbb{E}_{\pi}\left[G_t|S_t = s\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1}|S_t = s\right] \\ &= \sum_a \pi(a|s)\sum_{s'}\sum_r p(s', r|s, a)\left[r + \gamma\mathbb{E}[G_{t+1}|S_{t+1} = s']\right] \\ &= \sum_a \pi(a|s)\sum_{s,r} p(s', r|s, a)[r + \gamma V^{\pi}(s')],\ \forall s \in \mathcal{S} \\ &= \sum_{a\in\mathcal{A}} \pi(a, s)\left(r(s, a) + \gamma\sum_{s'\in\mathcal{S}} P(s'|s, a)V^{\pi}(s')\right). \end{aligned} \tag{1.11}$$

Equation 1.11 is known as the *Bellman equation* for V^{π}; it express a relationship between the value of a state and the values of its successor states: specifically, the Bellman equation averages over all the possibilities, weighting each by its probability of occurring. Note that the value function V^{π} is the unique solution to its Bellman equation. The same recursive relationship that holds for the Value Function in Equation 1.11 can be applied to the Action-Value function, yielding the following Bellman equation for $Q^{\pi}(s, a)$:

$$\begin{aligned} Q^{\pi}(s, a) &= \mathbb{E}_{\pi}\left[R_{t+1} + \gamma Q^{\pi}(s_{t+1}, a_{t+1})|s_t = s, a_t = a\right] \\ &= r(s, a) + \gamma\sum_{s'\in\mathcal{S}} P(s'|s, a)V^{\pi}(s'). \end{aligned} \tag{1.12}$$

It is of extreme importance to point out that the Bellman equations can also be

represented in matrix form, thus making it easier to notice the closed form solution:

$$V^\pi = R^\pi + \gamma P^\pi V^\pi = (I - \gamma P^\pi)^{-1} R^\pi. \tag{1.13}$$

In Equation 1.13, the superscript π refers to the functions obtained when applying a fixed policy to the MDP, so keeping the actions fixed.

1.1.4 Optimal Policy and Optimal Value Function

Roughly speaking, solving a reinforcement learning task consists in finding a policy that achieves a high cumulative reward in the long run. A policy π is defined to be better than or equal to another policy π' if its expected return is greater than or equal to the one of π' for all states:

$$\pi \geq \pi' \text{ if and only if } V^\pi(s) \geq V^{\pi'}(s) \, \forall s \in \mathcal{S}.$$

There is always at least one policy that is better than or equal to all other policies, such a policy is referred to as the *optimal policy*; it must be noted that although there may be more than one optimal policy, we denote them all by π^*. All optimal policies share the same value function, referred to as the *optimal value function*, denoted V^*.

Definition 1.5 (Optimal Value Function)**.**

$$V^*(s) = \max_\pi V_\pi(s), \, \forall s \in \mathcal{S}. \tag{1.14}$$

Optimal policies also share the same *optimal action-value* function, denoted Q^*.

Definition 1.6 (Optimal Action-Value Function)**.**

$$Q^*(s, a) = \max_\pi Q_\pi(s, a), \, s \in \mathcal{S}, a \in \mathcal{A}. \tag{1.15}$$

Using the Bellman equation for the state-value function in Equation 1.12, we can write the optimal state-value function Q^* in terms of the optimal value function

V^*:

$$Q^*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma V^*(S_{t+1}) | S_t = s, A_t = a\right]. \tag{1.16}$$

1.2 Dynamic Programming

The term DP refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as an MDP. Standard DP algorithms are however of limited utility in solving a reinforcement learning task because of both the assumption of a perfect model and because of the great computational burden they carry. Nonetheless, DP algorithms are important from a theoretical standpoint as they provide a foundation for the understanding of more advanced methods. We start with assuming a finite MDP, as DP algorithms do not ensure convergence to an exact solution in the case of continuous action or state spaces.

It is important to distinguish between algorithms that do not have a model of the MDP environment versus algorithms that do have such a model. The former algorithms are known as *learning* algorithms, as the agent will need to interact with the real-world environment and learn the Value Function from data it receives through interaction. The latter are known as *planning* algorithms as the agent requires no interaction with the real-world environment but rather projects probabilistic scenarios of future states and rewards for various choices of actions, and solves for the Value Function based on such projected outcomes. Another distinction that is worth making is the one between *prediction* tasks and *control* tasks: in fact, in the former case the goal is to predict the Value Function of a specific state, while in the latter the goal is to find the optimal policy to find the optimal solutions.

1.2.1 Fixed-Point Theory

In this section we will present the three classical Dynamic Programming (DP) algorithms, which are founded on the Bellman Equations presented in the previous section: in fact, each of the three algorithms is an iterative one where the computed Value function converges to the true Value Function as the number of iterations approaches infinity; furthermore, each of the three algorithms is based on the concept of *Fixed-Point* , Agarwal et al. (2018), Rao, Jelvis (2022), and on updating the computed Value-Function towards the fixed point.

Definition 1.7 (Fixed-Point)**.** *The Fixed-Point of a function $f : \mathcal{X} \to \mathcal{X}$ is a value $x \in \mathcal{X}$ that satisfies the equation: $x = f(x)$.*

Note that for some functions we have multiple fixed-points and for some other we have none; the algorithms that we will cover consider functions that have a unique fixed point.

Theorem 1.1 (Banach Fixed-Point Theorem)**.** *Let $\mathcal{X}$ be a non-empty set equipped with a complete metric $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. Let $f : \mathcal{X} \to \mathcal{X}$ be such that there exists a $L \in [0, 1)$ such that $d(f(x_1), f(x_2)) \leq L \cdot d(x_1, x_2)$. Then,*

1. *There exists a unique Fixed-Point $x^* \in \mathcal{X}$:*

$$x^* = f(x^*);$$

2. *For any $x_0 \in \mathcal{X}$, and sequence $[x_i | i = 0, 1, \dots]$ defined as $x_{i+1} = f(x_i)$ for all $i = 0, 1, \dots$, we have:*

$$\lim_{i \to \infty} x_i = x^*;$$

3.

$$d(x^*, x_i) \leq \frac{L^i}{1 - L} \cdot d(x_1, x_0),$$

or, equivalently

$$d(x^*, x_{i+1}) \leq L \cdot d(x^*, x_i).$$

To properly explain the theorem above, we are going to briefly define what a *contraction* is.

Definition 1.8 (Contraction)**.** *A function $f : \mathcal{X} \to \mathcal{X}$ is said to be a contraction function if two points in $\mathcal{X}$ get close when they are mapped by f. That is, there is some $L \in [0, 1)$ such that*

$$d(f(x_1), f(x_2)) \leq L \cdot d(x_1, x_2).$$

Banach Fixed-Point theorem is thus extremely important for DP algorithms, as, once we have identified the appropriate set $\mathcal{X}$ and the appropriate metric d, and ensured that f is a contraction function with respect to d, the theorem enables us to solve for the fixed-point of f with an iterative process of applying f repeatedly, starting with any arbitrary value $x_0 \in \mathcal{X}$.

1.2.2 Bellman Policy Operator and Policy Evaluation

The first DP algorithm we are going to deal with is called *Policy Evaluation*; this algorithm solves the problem of calculating the Value Function of a Finite MDP evaluated with a fixed policy π. The specification of this prediction problem is as follows: Let the states of the MDP be $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$. We are given a fixed policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ and we are also given the transition probability function $P : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ in the form of a data structure. The prediction problem is to compute the Value Function of the MDP when evaluated with policy π, which we denote as $V^\pi : \mathcal{S} \to \mathbb{R}$. We could solve this problem using Equation 1.13, however when the number of states is large, the computational burden becomes too high.

Definition 1.9 (Bellman Policy Operator)**.** *The Bellman Policy Operator is a*

function $B^\pi : \mathbb{R}^m \to \mathbb{R}^m$ *defined as*

$$B^\pi(V) = \mathcal{R}^\pi + \gamma P^\pi \cdot V, \tag{1.17}$$

for any vector V *in the vector space* $\mathbb{R}^m$.

Notice that the definition above is equivalent to the following

$$B^\pi(V)(s) = \sum_{a \in \mathcal{A}} \pi(a, s) \left(r(s, a) + \gamma \sum_{s \in \mathcal{S}} P(s'|s, a) \cdot V(s') \right). \tag{1.18}$$

Using Equation 1.17 above we can express the MDP Bellman Equation as:

$$V^\pi = B^\pi(V^\pi).$$

which means that $V^\pi \in \mathbb{R}^m$ is a Fixed-Point for the Bellman Policy Operator. Now choosing as our metric $d : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$ the L^∞ norm, defined as:

$$d(X, Y) = ||X - Y||_\infty = \max_{s \in \mathcal{S}} |(X - Y)(s)|.$$

B^π is a contraction function under the L^∞ norm. We can now invoke Banach Fixed Point Theorem, Theorem 1.1, to come up with the following theorem.

Theorem 1.2 (Policy Evaluation Theorem)**.** *For a Finite MDP with* $|\mathcal{S}| = m$ *and* $\gamma \leq 1$*, if* $V^\pi \in \mathbb{R}^m$ *is the Value Function of the MDP when evaluated with a fixed policy* $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$*, then* V^π *is the unique Fixed-Point of the Bellman Policy Operator* $B^\pi : \mathbb{R}^m \to \mathbb{R}^m$*, and*

$$\lim_{i \to \infty} (B^\pi)^i(V_0) \to V^\pi \text{ for all starting Value Functions } V_0 \in \mathbb{R}^m.$$

From Theorem 1.2 we can derive the following algorithm.

Algorithm 1.1 Policy Evaluation Algorithm, Source: Sutton, Barto (2018)

Require: π, the policy to be evaluated
1: Initialize $V_0 \in \mathbb{R}^m$
2: Initialize $\epsilon =$ a small positive number , $\Delta = 0, i = 0$
3: **while** $\Delta \geq \epsilon$ **do**
4: $\quad V_{i+1}(s) \leftarrow B^\pi(V_i)(s)$, for all $s \in \mathcal{S}$
5: $\quad \Delta \leftarrow d(V_i, V_{i+1}) = \max_{s \in \mathcal{S}} |(V_i - V_{i+1})(s)|$
6: **end while**

It is important to notice that the Banach Fixed-Point Theorem not only assures convergence to the unique solution, it also assures a reasonable speed of convergence to it, Rao, Jelvis (2022).

1.2.3 Policy Improvement

In the previous subsection we presented an algorithm to solve the MDP *prediciton* problem, the next two algorithms however are dedicated to solving the MDP *control* problem. In order to make this step from *prediction* to *control* we must first define a function that is motivated by the idea of improving a policy or a value function with a *greedy* technique.

Definition 1.10 (Greedy Policy)**.** *The Greedy Policy Function is a function* $G : \mathbb{R}^m \to (\mathcal{S} \to \mathcal{A})$*, which is to interpret as a function mapping a Value Function* V *to a deterministic policy* $\pi'_D : \mathcal{S} \to \mathcal{A}$*. It is defined as:*

$$G(V)(s) = \pi'_D(s) = \arg\max_{a \in \mathcal{A}} \left\{ r(s,a) + \gamma \cdot \sum_{s' \in \mathcal{S}} P(s'|a,s) \cdot V(s') \right\}, \forall s \in \mathcal{S}.$$

Now that we have introduced the Greedy Policy Function, we can move to the concept of improvement referred to as either Value Functions or Policies.

Definition 1.11 (Value Function Comparison)**.** *For Value Functions* $X, Y : \mathcal{S} \to \mathbb{R}$ *of an MDP, we say* $X \geq Y$ *if and only if:*

$$X(s) \geq Y(s), \forall s \in \mathcal{S}.$$

Thus, whenever we refer to a 'better Value Function', it should be interpreted as that the Value Function is *no worse for each of the states* compared to another Value Function. Using the two definitions above, we can introduce the following theorem by Richard Bellman, Bellman (1957a).

Theorem 1.3 (Policy Improvement Theorem[1])**.** *Let* $\pi'_D = G(V^\pi)$ *and let* π *be any other policy. Then, for a Finite MDP,*

$$V^{\pi'_D} = V^{G(V^\pi)} \geq V^\pi.$$

The Policy Improvement Theorem allows us to introduce the first DP algorithm to solve the MDP control problem, called *Policy Iteration*, Howard (1960).

1.2.4 Policy Iteration

The Policy Improvement Theorem allows us to start with any Value Function V^π for a policy π, perform a greedy policy improvement to create a policy $\pi'_D = G(V^\pi)$, and then perform a Policy Evaluation with starting Value Function V^π, resulting in an improved Value Function $V^{\pi'_D}$. We clearly can repeat the same process multiple times, creating further improved policies and associated Value Functions until we have no improvement. This idea yields the following algorithm[2].

Algorithm 1.2 Policy Iteration Algorithm, Source: Sutton, Barto (2018)

1: Initialize $V_0 \in \mathbb{R}^m$
2: Initialize $\epsilon =$ a small positive number , $\Delta = 0, j = 0$
3: **while** $\Delta \geq \epsilon$ **do**
4: Deterministic Policy $\pi_{j+1} \leftarrow G(V_j)$
5: Value Function $V_{j+1} \leftarrow \lim_{i\to\infty}(B^{\pi_{j+1}})^i(V_j)$
6: $\Delta \leftarrow d(V_j, V_{j+1}) = \max_{s\in\mathcal{S}} |(V_j - V_{j+1})(s)|$
7: **end while**

The algorithm above terminates when there is no further improvement to the Value

[1]The proof of Theorem 1.10 can be found at Rao, Jelvis (2022)
[2]In step 5 of the algorithm we perform a full policy evaluation

Function, that is:

$$V_j = \left(B^{G(V_j)}\right)^i (V_j) = V_{j+1}.$$

Which, for $i = 1$ yields:

$$V_j(s) = B^{G(V_j)}(V_j)(s) = r\left(s, G(V_j)(s)\right) + \gamma \sum_{s' \in \mathcal{S}} P\left(s' | G(V_j)(s), s\right) \cdot V_j(s'),\ \forall s \in \mathcal{S}.$$

However, from Definition 1.10, we know that for each state, the action that comes from the greedy policy is the action that maximizes

$$\left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|a,s) \cdot V_j(s') \right\}.$$

Therefore, by joining all together we can see that what we have is the following

$$V_j(s) = \max_{a \in \mathcal{A}} \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|a,s) \cdot V_j(s') \right\}\ \forall s \in \mathcal{S}.$$

However, we can notice the above is non-other than the MDP State-Value Function Bellman Optimality Equation as in Equation 1.15, which means that when V_{j+1} is identical to V_j, the Policy Iteration algorithm has converged to the Optimal Value Function; the associated deterministic policy at convergence is an Optimal Policy as $V\pi_j = V_j \approx V^*$. The latter in turn means that the MDP evaluated with the deterministic policy π_j achieves the Optimal Value Function, that is, the Policy Iteration algorithm solves the MDP control problem. The following theorem states what we have just concluded.

Theorem 1.4 (Policy Iteration Convergence Theorem)**.** *For a Finite MDP with $|\mathcal{S}| = m$ and $\gamma < 1$, the Policy Iteration algorithm converges to the Optimal Value Function $V^* \in \mathbb{R}^m$ along with a Deterministic Optimal Policy $\pi_D^* : \mathcal{S} \to \mathcal{A}$, no matter which Value Function $V_0 \in \mathbb{R}^m$ we start the algorithm with.*

1.2.5 Value Iteration

In order to introduce the Value Iteration algorithm, we must first make a small change to the definition of the Greedy Policy Function in Definition 1.10.

Definition 1.12 (Bellman Optimality Operator)**.** *The Bellman Optimality Operator $B^* : \mathbb{R}^m \to \mathbb{R}^m$ is defined as*

$$B^*(V)(s) = \max_{a \in \mathcal{A}} \left\{ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|a,s) \cdot V(s') \right\}, \forall s \in \mathcal{S}.$$

If we now apply the Bellman Policy Operator on any Value Function $V \in \mathbb{R}^m$ using the greedy policy, it should be identical to the Bellman Optimality Operator, therefore

$$B^{G(V)}(V) = B^*(V), \forall\, V \in \mathbb{R}^m.$$

It is interesting to observe that by specializing V to be the Value Function V^π for a policy π, we get:

$$B^{G(V^\pi)}(V^\pi) = B^*(V^\pi),$$

which is a succinct representation of the first stage of Policy Evaluation with an improved policy $G(V^\pi)$. The Bellman Optimality Operator is motivated by the MDP State-Value Function Optimality Equation. We can therefore express the MDP State-Value Function Bellman Optimality Equation succinctly as:

$$V^* = B^*(V^*),$$

which means that $V^* \in \mathbb{R}^m$ is a Fixed-Point of the Bellman Optimality Operator $B^* : \mathbb{R}^m \to \mathbb{R}^m$. Now, since B^* is a contraction function[3], and invoking Banach Fixed-Point Theorem allows us to prove the convergence of Value Iteration.

Theorem 1.5 (Value Iteration Convergence)**.** *For a Finite MDP with $|\mathcal{S}| = m$ and $\gamma < 1$, if $V^* \in \mathbb{R}^m$ is the Optimal Value Function, then V^* is the unique*

[3]The proof can be found at Rao, Jelvis (2022)

Fixed-Point of the Bellman Optimality Operator $B^* : \mathbb{R}^m \to \mathbb{R}^m$, *and*

$$\lim_{i\to\infty}(B^*)^i(V_0) \to V^*, \text{ for all starting Value Functions } V_0 \in \mathbb{R}^m.$$

Theorem 1.5 leads us to the following algorithm by Bellman, Bellman (1957b).

Algorithm 1.3 Value Iteration Algorithm, Source: Sutton, Barto (2018)

1: Initialize $V_0 \in \mathbb{R}^m$
2: Initialize $\epsilon =$ a small positive number , $\Delta = 0, j = 0$
3: **while** $\Delta \geq \epsilon$ **do**
4: $\quad V_{i+1}(s) \leftarrow B^*(V_i)(s)$ for all $s \in \mathbb{S}$
5: $\quad \Delta \leftarrow d(V_i, V_{i+1}) = \max_{s\in\mathcal{S}} |(V_i - V_{i+1})(s)|$
6: **end while**

1.3 Prediction and Control with Reinforcement Learning

In this section, we are going to present some key methods in RL for estimating value functions and discovering optimal policies. The main difference in these methods as compared to DP is that they do not assume perfect knowledge of the environment.

1.3.1 Monte Carlo Methods

Monte Carlo (MC) methods are a family of so-called *model-free* methods: in fact, as opposed to *model-based* methods, MC methods do not need to estimate the exact model but rather aim at *learning* the value function of the policy. In MC methods, we approximate the value function from the average return computed by sampling the past experience with an incremental approach: the current average is updated through the new experience samples collected.

It is important to point out that the algorithms we are going to present are *on-policy* algorithms, which use the same policy to explore and to update, as opposed

to *off-policy* algorithms which use two policies: one to explore and collect information and one for learning a target policy.

Monte Carlo Methods for Prediction

We begin by looking at MC methods for prediction, that is, for learning the value function for a given policy.

In MC methods, the first time a state is visited in an episode is referred to as the *first-visit* to s. The following algorithm estimates the value of a state s under a policy π as the average of returns following first visits to s as:

$$\hat{V}^{\pi} = \sum_{i=0}^{N-1} G_i = \sum_{i=0}^{N-1} \sum_{t=0}^{T-1} \gamma^t R_{i,t+1}. \tag{1.19}$$

The MC estimator thus has mean $\mathbb{E}\left[\hat{V}^{\pi}\right] = V^{\pi}$ and variance $Var(\hat{V}^{\pi}) = \frac{Var(V^{\pi})}{N}$ as the number of fist-visits to s goes to infinity. We can now present the following algorithm, Sutton, Barto (2018).

Algorithm 1.4 First - Visit MC Prediction, Source: Sutton, Barto (2018)

Input: a policy π to be evaluated
Initialize: $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
Initialize: $Returns(s)$, an empty list, for all $s \in \mathcal{S}$
for each episode **do**
    Generate an episode following π
    $G \leftarrow 0$
    for each step of episode $t = T-1, T-2, \ldots, 0$ **do**
        $G \leftarrow \gamma G + R_{t+1}$
        if S_t is **not** in $\{S_0, S_1, \ldots, S_{t-1}\}$ **then**
            Append G to $Returns(S_t)$
            $V(S_t) \leftarrow$ average $(Returns(S_t))$
        end if
    end for
end for

Monte Carlo Methods for Control

The idea in MC Control is to proceed in a Generalized Policy Iteration (GPI) fashion: in GPI the value function is repeatedly updated to approximate the value function for the current policy, and the policy is repeatedly improved with respect to the current value function. The two combined cause both the policy and the value function to approach optimality. The natural idea would then be to use an MC Prediction algorithm such as Algorithm 1.4, followed by a Greedy Policy Improvement; the problem would be that Greedy Policy calculation in Definition 1.10) requires a model for the transition probability function $P(s'|s,a)$ and the reward function $r(s,a)$, two things that are not at our disposal as we are dealing with a *model-free* method. We can however reformulate the problem: in fact, we can see that the equation for the Greedy Policy Improvement is equivalent to

$$\pi'_D(s) = \arg\max_{a \in \mathcal{A}} Q^\pi(s,a) \, \forall s \in \mathcal{S}.$$

This view of Greedy Policy Improvement is extremely useful because it allows us to use an MC Prediction algorithm to calculate Q^π, followed by Greedy Policy Improvement until convergence.

Algorithm 1.5 Monte Carlo Exploring Starts, Source: Sutton, Barto (2018)

Initialize: $\pi(s) \in \mathbb{A}$, arbitrarily, for all $s \in \mathcal{S}$
Initialize: $Q(s,a) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize: $Returns(s,a)$, an empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}$
for each episode **do**
    Choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}$ randomly such that all pairs have probability > 0
    Generate an episode from S_0, A_0 following π
    $G \leftarrow 0$
    for each step of episode $t = T-1, T-2, \ldots, 0$ **do**
        $G \leftarrow \gamma G + R_{t+1}$
        if S_t, A_t is **not** in $\{(S_0, A_0), (S_1, A_1), \ldots, (S_{t-1}, A_{t-1})\}$ **then**
            Append G to $Returns(S_t, A_t)$
            $Q(S_t, A_t) \leftarrow$ average $(Returns(S_t, A_t))$
            $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$
        end if
    end for
end for

1.3.2 Temporal Difference Learning

Temporal-difference (TD) learning, Tesauro (1995), is probably one of the most important ideas that was brought into the reinforcement learning landscape. TD is a combination of Monte Carlo (MC) and Dynamic Programming (DP): in fact, as for MC methods, TD methods learn from experience without a model of the environment dynamics, on the other hand, as for DP methods, TD methods update estimates based partly on other learned estimates without waiting for a final outcome, that is, they use *bootstrapping*.

TD Prediction

As we have seen in the previous section, MC methods wait until the return following the visit to a state is known, and then use that return as an update target for the value function of that state. A simple MC update rule would then be

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t) \right].$$

Now, as we have previously mentioned, this MC method must wait until the end of the episode to calculate the return and determine the increment to $V(S_t)$. The significant improvement we have with TD methods is that we can perform an update on the value function immediately after observing the return R_{t+1} and the estimate $V(S_{t+1})$. We can thus define a simple update rule for a TD algorithm to be:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right]. \tag{1.20}$$

This rule brings us to the Algorithm 1.6, which is a very simple TD prediction algorithm.

Algorithm 1.6 TD(0) Prediction, Source; Sutton, Barto (2018)

Input: a policy π to be evaluated
Parameter: the step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}$, if s is terminal, then $V(terminal) = 0$
for each episode **do**
    Initialize S
    for each step in *episode* **do**
        $A \leftarrow \pi(S)$
        Take action A, observe R, S'
        $V(S) \leftarrow V(S) + \alpha \left[R + \gamma V(S') - V(S)\right]$
        $S \leftarrow S'$
        if S is terminal **then**
            break
        end if
    end for
end for

SARSA

SARSA is a *on-policy* TD algorithm for control tasks, as we saw in the case of MC Control algorithms, we follow the pattern of Generalized Policy Iteration (GPI) using TD methods for the prediction part. Again, as we saw for MC Control algorithms, we start by learning an action-value function rather than a state-value function; we thus need to estimate $Q^{\pi}(s, a)$ for the current policy and for all states s and actions a.

Since we are not dealing with transitions from state to state as in the previous section, but with transitions from state-action pair to state-action pair we have to specify that formally they are both Markov chains with a reward process, thus the same theorems that assure the convergence of state values in the TD(0) algorithm, Algorithm 1.6, apply to action-values. The update rule is thus

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]. \tag{1.21}$$

The update is done after every transaction from a non-terminal state S_t. If S_{t+1}

is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined to be zero. The rule thus uses for each iteration the tuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, the same tuple gives the name to the algorithm, which is $SARSA$[4].

Algorithm 1.7 SARSA, Source: Sutton, Barto (2018)

Parameter: the step size $\alpha \in (0, 1]$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}$, if s is terminal, then $Q(terminal, \cdot) = 0$
for each episode **do**
    Initialize S
    $A \leftarrow \pi_Q(S)$
    for each step in *episode* **do**
        Take action A, observe R, S'
        $A' \leftarrow \pi_Q(S')$
        $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma Q(S', A') - Q(S, A)\right]$
        $S \leftarrow S'$; $A \leftarrow A'$
        if S is terminal **then**
            break
        end if
    end for
end for

Q-Learning

Q-learning is one of the main breakthroughs in reinforcement learning and is an *off-policy* Temporal Difference learning algorithm first presented in Watkins, Dayan (1992). The update rule for Q-learning is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{1.22}$$

The update rule is the starting point for the following algorithm.

[4] In the algorithm, π_Q refers to the policy derived from Q

Algorithm 1.8 Q-learning, Source: Sutton, Barto (2018)

Parameter: the step size $\alpha \in (0, 1]$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}$, if s is terminal, then $Q(terminal, \cdot) = 0$
for each episode **do**
    Initialize S
    for each step in *episode* **do**
        $A \leftarrow \pi_Q(S)$
        Take action A, observe R, S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma Q(S', A') - Q(S, A)\right]$
        if S is terminal **then**
            break
        end if
    end for
end for

Chapter 2

Approximate Solution Methods

In the previous chapter, we presented Dynamic Programming algorithms for Markov Decision Processes (MDPs). These algorithms operate efficiently when MDPs are represented using finite data structures, and we can express the Value Function in a tabular format, consisting of states and their corresponding values. The core of these Dynamic Programming methods involves a sweeping process, where all states are visited during each iteration to iteratively update the value function. However, practical challenges arise when dealing with MDPs featuring a very large state space:

1. Conventional "tabular" representations of MDPs or their associated Value Functions become unfeasible due to storage limitations.

2. Sweeping through all states and their associated transition probabilities proves either excessively time-consuming or, in the case of infinite state spaces, an infeasible task.

To address these challenges, we need to move to methods that involve the approximation of the Value Function. The adaptation of Dynamic Programming algorithms to their Approximate Dynamic Programming (ADP) counterparts offers a

viable solution. This adaptation involves a fundamental shift in the methodology: instead of sweeping all states in each iteration, we opt for a more practical approach. Specifically, we selectively sample a relevant subset of states and compute their values employing the same calculations used in the tabular setting. Subsequently, we construct or update a function approximation for the Value Function based on the computed values of the sampled states. Moreover, in cases where the set of transitions emanating from a given state is large or infinite, we can not use explicit transition probabilities. Instead, we leverage probabilistic sampling techniques from the transition probability distribution. Importantly, despite these modifications to suit larger state spaces, the fundamental structure of the algorithms and the core principles governing them, including the Fixed-Point concept and the Bellman Operators we presented in the previous chapter, remain unaltered.

2.1 Function Approximation

In this section, we present function approximation within a generalized context, not limited to the approximation of Value Functions or Policies. We denote the predictor variable as x, belonging to an arbitrary domain denoted as X, and the response variable as $y \in \mathbb{R}$. We treat x and y as unknown random variables, and our objective is to estimate the probability distribution function f of the conditional random variable $y|x$ based on data provided in the form of a sequence of (x, y) pairs. We consider parameterized functions f with parameters denoted as w. The specific data type of w depends on the particular form of function approximation. We represent the estimated probability of y given x as $f(x; w)(y)$. Assuming we are furnished with data in the form of a sequence of n (x, y) pairs, $[(x_i, y_i)|1 \leq i \leq n]$. The concept of estimating the conditional probability $P[y|x]$

is formalized by seeking $w = w^*$ such that:

$$w^* = \arg\max_w \left\{ \prod_{i=1}^{n} f(x_i; w)(y_i) \right\} = \arg\max_w \left\{ \sum_{i=1}^{n} \log f(x_i; w)(y_i) \right\}.$$

We are thus operating in the Maximum Likelihood Estimation (MLE) framework, Rossi (2018), Rao, Jelvis (2022): in such a framework we say the data $[(x_i, y_i)|1 \leq i \leq n]$ specifies the *empirical probability distribution* D of $y|x$ and the function f specifies the *model probability distribution*. In MLE, we essentially minimize the *cross-entropy* [1] between the probability distribution D and M. In this work, we will use incremental estimation wherein at each iteration of the incremental estimation we use data to update the parameters of the function. The estimate f also allows us to calculate the model-expected value of y conditional on x, that is:

$$\mathbb{E}_M [y|x] = \mathbb{E}_{f(x;w)}[y] = \int_{-\infty}^{+\infty} y \cdot f(x; w)(y) dy.$$

In the context of Dynamic Programming and Reinforcement Learning, the function approximation's prediction provides an estimate of the Value Function for any state.

2.1.1 Value Function Approximation

The prediction methods we have described so far all involve some sort of update to an estimated value function towards an update target. The updates that characterize the methods we have presented so far are all quite trivial: in fact, we were able to simply change the value function estimate for an individual state s, while leaving all the estimated values for the other states unchanged. In the following sections, we will present methods that implement more complex and sophisticated updates, where updating at an individual state s generalizes so that the estimated values of other states will be changed as well.

[1] Given two distributions D, M, the cross-entropy is defined as $\mathcal{H}(D, M) = -\mathbb{E}_D [\log M]$

Nowadays, many *supervised learning* methods are used to carry out the approximation task. It must be noted that not all function approximation methods are well suited for a reinforcement learning task. In fact, it is important that learning can occur online, while the agent interacts with the environment. In order to be able to do so, a supervised learning method needs to learn efficiently from incrementally acquired data. Furthermore, many reinforcement learning tasks involve a non-stationary target function [2], thus making it necessary for the learning method to be able to handle nonstationary data.

The Prediction Objective

In the prediction methods we have presented so far, we never specified an explicit objective for prediction: in fact, in the tabular case, the learned value function can converge to the true value function exactly, thus making it pointless to have a measure of prediction quality. However, since we are now dealing with infinite or very large state spaces, and we have by assumption far more states than weights, it becomes infeasible to correctly approximate the value for all states. We thus must specify a state distribution $\mu(s) \geq 0$, with $\sum_s \mu(s) = 1$, representing how much we weigh the error in each state s; such an error is measured as the square of the difference between the approximate value $\hat{V}(s, w)$ and the true value $V^\pi(s)$. Weighting this over the state space by μ, we obtain a natural objective function, the *Mean Squared Value Error*, Sutton, Barto (2018), denoted $\overline{\text{VE}}$, defined as

$$\overline{\text{VE}}(w) := \sum_{s \in \mathcal{S}} \mu(s) \left(V^\pi(s) - \hat{V}(s, w) \right)^2 . \tag{2.1}$$

[2] A process Y_t is said to be *covariance-stationary*, Hamilton (1994), if neither the mean μ_t nor the autocovariances γ_{jt} depend on the time t, that is

$$\mathbb{E}[Y_t] = \mu \text{ for all } t,$$
$$\mathbb{E}[(Y_t - \mu)(Y_{t-j} - \mu)] = \gamma_j \text{ for all } t \text{ and any } j$$

The square root of $\overline{\text{VE}}$ gives a rough measure of how much the approximate values differ from true values.

2.1.2 Stochastic-Gradient Methods

In this section we will present one class of learning methods for function approximation, that is, those based on *Stochastic Gradient Descent* (SGD). In gradient descent methods, the weight vector is a column vector with a fixed number of real-valued components $w := (w_1, w_2, \ldots, w_m) \in \mathbb{R}^m$, and the approximate value function $\hat{V}(s, w)$ is a differentiable function of w for all $s \in \mathcal{S}$. In these methods, w is updated at each of a series of discrete time steps $t = 0, 1, \ldots$, so we will denote w_t the weight vector at time step t. Assume now we are trying to minimize the Mean Squared Value Error (2.1) by minimizing the error on the observed samples. SGD methods will do so by adjusting the weight vector after each example by a small amount in the direction that would most reduce the error on that sample. The update rule for the weights would then be

$$\begin{aligned} w_{t+1} &= w_t - \frac{1}{2}\alpha\nabla\left[V^\pi(S_t) - \hat{V}(S_t, w_t)\right]^2, \\ w_{t+1} &= w_t + \alpha\left[V^\pi(S_t) - \hat{V}(S_t, w_t)\right]\nabla\hat{V}(S_t, w_t). \end{aligned} \tag{2.2}$$

where $\nabla f(w)$ denotes the column vector of partial derivatives of the expression with respect to the components of the vector:

$$\nabla f(w) = \left(\frac{\partial f(w)}{\partial w_1}, \frac{\partial f(w)}{\partial w_2}, \ldots, \frac{\partial f(w)}{\partial w_m}\right).$$

The derivative vector is the *gradient* of f with respect to w, which gives the name to the family of methods: in fact, SGD methods take a step in w_t which is proportional to the negative gradient of the sample's error, which is the direction in which the error falls most rapidly.

2.1.3 Linear Function Approximation

Let us define a sequence of feature functions $\phi_j : \mathcal{X} \to \mathbb{R}$ for each $j = 1, 2, \ldots, m$ and we define $\phi : \mathcal{X} \to \mathbb{R}^m$ as:

$$\phi(x) : (\phi_1(x), \phi_2(x), \ldots, \phi_m(x)) .$$

We will treat $\phi(x)$ as a column vector for all $x \in \mathcal{X}$. For Linear function approximation, the parameters are represented as a column vector $w = (w_1, w_2, \ldots, w_m) \in \mathbb{R}^m$. It must be noted that Linear function approximation is based on the *Gaussian assumption*[3], so the cross-entropy loss function for a given set of data $[x_i, y_i | 1 \leq i \leq n]$ is defined as:

$$\mathcal{L}(w) = \frac{1}{2n} \cdot \sum_{i=1}^{n} \Big(\phi(x_i)^T \cdot w - y_i\Big)^2 .$$

We can easily notice that this loss function is identical to the Mean Squared Error (MSE) [4] of the linear predictions $\phi(x_i)^T \cdot w$ relative to the response values y_i associated with the predictor values x_i. Now, if we were to include L^2 regularization with λ as regularization coefficient we would get a Ridge regression, Hastie et al. (2001),

$$\mathcal{L}(w) = \frac{1}{2n} \left(\sum_{i=1}^{n} (\phi(x_i)^T \cdot w - y_i)^2 \right) + \frac{1}{2} \cdot \lambda \cdot |w|^2.$$

And the gradient of $\mathcal{L}(w)$ with respect to w is

$$\nabla_w \mathcal{L}(w) = \frac{1}{n} \cdot \left(\sum_{i=1}^{n} \phi(x_i) \cdot (\phi(x_i)^T \cdot w - y_i) \right) + \lambda \cdot w.$$

[3] It is the assumption of $y|x$ following a Gaussian Distribution, with mean

$$\mathbb{E}_M[y|x] = \sum_{j=1}^{m} \phi_j(x) \cdot w_j = \phi(x)^T \cdot w.$$

and constant variance σ^2.

[4] The MSE is given by $\frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$, where $\hat{Y}_i$ is the prediction and Y_i is the true value.

With the above formula for the gradient, we can find the optimal weights via Stochastic Gradient Descent using the update rule as in (2.2).

Note that in linear function approximation, we can directly solve for the optimal weights w^* if m is not too large. If we denote Φ the $n \times m$ matrix defined as $\Phi_{i,j} = \phi_j(x_i)$ and $Y \in \mathbb{R}^n$ the columns vector defined as $Y_i = y_i$, then we can find the optimal weight as the solution of the following linear system of size m:

$$\begin{aligned} \frac{1}{n} \cdot \Phi^T(\Phi w^* - Y) + \lambda w^* &= 0, \\ \left(\Phi^T \cdot \Phi + n\lambda \cdot I_m\right) \cdot w^* &= \Phi^T \cdot Y, \\ w^* &= \left(\Phi^T \cdot \Phi + n\lambda \cdot I_m\right)^{-1} \cdot \Phi^T \cdot Y. \end{aligned}$$

2.1.4 Nonlinear Function Approximation with Artificial Neural Networks

In the previous section, we presented linear function approximation; now we will generalize it in order to allow us to perform nonlinear function approximation using a fully connected (FC) Artificial Neural Network (ANN), as in Rao, Jelvis (2022). Throughout this section, we will try to use as much as possible the notation used for the previous section.

Let us start by denoting as L the number of layers of an ANN, then we will denote as $l = 0, 1, \ldots, L-1$ the hidden layers of the ANN, and the layer $l = L$ will be the output layer. Before we move on, it is also worth specifying that all the vectors that we will deal with will be treated as column vectors. We will further denote as i_l the input vector to layer l, and as o_l the output of layer l. As for standard notation in statistics, we will denote as $x \in \mathcal{X}$ the predictor variable and as $y \in \mathbb{R}$ the response variable. We then have that the input of the first layer will be $i_0 = \phi(x) \in \mathbb{R}^m$, and the output of the last layer will be $o_L = \mathbb{E}_M[y|x]$. Clearly, the input for every layer $l+1$ will be the output of the layer l, that is,

$i_{l+1} = o_l, \forall l = 0, 1, \ldots, L-1$.

We will denote the parameters for layer l as a matrix w_l of dimensions $dim(o_l) \times dim(i_l)$[5]. Note that the number of neurons in layer l is always equal to the dimension of its output, that is, it is equal to $dim(o_l)$. The neurons in each layer l define a linear transformation from layer input i_l to a variable we will denote as z_l, defined as:

$$z_l = w_l \cdot i_l. \tag{2.3}$$

Let us now denote the activation function of a layer l as a function $g_l : \mathbb{R} \to \mathbb{R}$. In ANNs, the activation function is applied pointwise on each dimension of the vector z_l, yielding the output of the layer l; we will thus denote such an operation as:

$$o_l = g_l(z_l). \tag{2.4}$$

If we now combine Equations 2.3 and 2.4 together with the input of the first layer being $i_0 = \phi(x)$, and the output of the final layer being $o_L = \mathbb{E}_M[y|x]$, we obtain the calculation that is performed by the ANN, which is known as *forward-propagation*. The following step is to derive an expression for the gradient of the cross-entropy loss, namely $\nabla_{w_l}\mathcal{L}$; the problem can be reduced to calculating the cross-entropy loss gradient of each layer, that is $\nabla_{z_l}\mathcal{L}$, which is possible through the chain rule: in fact we can see that:

$$\frac{\partial \mathcal{L}}{\partial w_l} = \frac{\partial \mathcal{L}}{\partial z_l} \cdot \frac{\partial z_l}{\partial w_l}.$$

We thus have the following expression for the gradient

$$\nabla_{w_l}\mathcal{L} = \nabla_{z_l}\mathcal{L} \cdot i_l^T, \tag{2.5}$$

where with $\nabla_{z_l}\mathcal{L} \cdot i_l^T$ we represent the outer product, which yields a matrix of dimensions $dim(o_l) \times dim(i_l)$.

[5]When we use the notation $dim(v)$, we refer to the dimension of a vector v.

If we now denote as P_l the gradient of the loss function with respect to z_l, that is $P_l = \nabla_{s_l}\mathcal{L}$, we can see that the gradient calculation has been reduced to the calculation of P_l for each layer. As in, Rao, Jelvis (2022), we can introduce the following theorem that is at the core of *back-propagation*, and provides us with a recursive formulation of P_l.

Theorem 2.1. [6] *For all l = 0, 1, . . . , L-1,*

$$P_l = (w_{l+1}^T \cdot P_{l+1}) \circ g_l'(z_l).$$

Where ∘ represents the point-wise multiplication of two vectors of the same dimension.

In order to calculate P_L, we can run the above recursive formulation for P_l, estimate the loss gradient, and perform gradient descent to obtain w_l^*. To calculate the gradient of the last layer with respect to z_L, that is, $\frac{\partial \mathcal{L}}{\partial z_L}$; with the intention of calculating such a quantity, we need to assume a functional form for $\mathbb{P}[y|s_L]$, Rao, Jelvis (2022). The functional form of choice will be a generic exponential functional form for the probability distribution function:

$$p(y|\theta, \tau) = h(y, \tau)e^{\frac{\theta \cdot y - A(\theta)}{d(\tau)}}. \tag{2.6}$$

In Equation 2.6, θ is the parameter related to the mean of the distribution, and τ is the parameter related to the variance of the distribution. Further notice that $h(\cdot,\cdot), A(\cdot), d(\cdot)$ are general functions whose specializations define the family of distributions that can be modeled with this functional form: in the case of our ANN, we can assume that τ is constant, and we can set θ to be z_L, as follows:

$$\mathbb{P}[y|z_L] = p(y|z_L, \tau) = h(y, \tau)e^{\frac{z_L \cdot y - A(z_L)}{d(\tau)}}. \tag{2.7}$$

Now, before moving on, let us recall that z_L, o_L, P_L are all scalars in the case that

[6] The proof of the Theorem 2.1 can be found at Rao, Jelvis (2022).

we are considering[7]. Before moving on, let us denote the expectation taken under the functional form in Equation 2.7 as $\mathbb{E}_p[\cdot]$, and let us establish the following Lemma:

Lemma 2.1. [8]

$$\mathbb{E}_p[y|z_L] = A'(z_L).$$

We can now require the prediction of the ANN to be equal to $\mathbb{E}_p[y|z_L]$:

$$o_L = g_L(z_L) = \mathbb{E}_p[y|z_L] = A'(z_L).$$

The equation above tells us that the activation value of the output layer L must be equal to the derivative of the $A(\cdot)$ function. Now, combining all the above, we can derive a single expression for P_L.

Theorem 2.2. [9]

$$P_L = \frac{\partial \mathcal{L}}{\partial z_L} = \frac{o_L - y}{d(\tau)}.$$

Notice that at each iteration of gradient descent, we will require an estimate of the loss gradient up to a constant factor, thus being able to ignore the constant $d(\tau)$ and allowing us to calculate only $P_L = o_L - y$. Once we have calculated P_L, we can then recursively calculate P_l for each layer l from $l = L - 1$ to $l = 0$.

2.2 Policy Gradient Methods

In this section, we present a different class of reinforcement learning techniques known as *Policy Gradient methods*. These methods offer an alternative approach to solving reinforcement learning problems compared to the familiar *action-value* methods we have explored thus far. Action-value methods focus on learning the

[7] This case is the univariate regression, where the response variable is $y \in \mathbb{R}$

[8] The proof of Lemma 2.1 can be found at Rao, Jelvis (2022).

[9] The proof of Theorem 2.2 be found at Rao, Jelvis (2022).

value of actions in a given state and then making decisions based on these estimates; on the other hand, Policy Gradient methods directly learn a *parameterized policy* to determine actions, bypassing the need for approximating the Value Function.

To formalize Policy Gradient methods, we will now introduce some essential notation. We will represent the policy's parameter vector as $\theta \in \mathbb{R}^d$, and define $\pi(a|s,\theta)$ as the probability of selecting action a at time t given the current state s and the policy parameter θ. In mathematical terms:

$$\pi(a|s,\theta) = \mathbb{P}\left\{A_t = a|S_t = s, \theta_t = \theta\right\}.$$

It is important to note that if a particular Policy Gradient method also incorporates a learned value function, we denote the weight vector of this value function as $w \in \mathbb{R}^m$.

All Policy Gradient methods involve optimizing a scalar performance measure denoted as $J(\theta)$. These methods aim to *maximize* this performance measure, which leads to updates approximating *gradient ascent* in J. The update rule for Policy Gradient methods typically takes the following form:

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)}. \tag{2.8}$$

In this equation, $\alpha\widehat{\nabla J(\theta_t)} \in \mathbb{R}^m$ represents a stochastic estimate, and its expected value approximates the gradient of the performance measure with respect to the policy parameter θ_t, Sutton, Barto (2018). Before moving on to the following section, it is important to make a distinction between *policy gradient* methods and *actor-critic methods*: in fact, in the former, the update is usually as in (2.8), in the latter, on the other hand, we have an *actor* that learns the policy and a *critic* that learn a value function, usually a state-value one.

In the subsequent sections, we will explore various Policy Gradient and Actor-

Critic methods that operate based on this framework. These methods provide a powerful toolset for solving reinforcement learning tasks, particularly when dealing with problems where a parameterized policy is advantageous over traditional action-value methods.

2.2.1 Policy Approximation

In policy gradient methods, the policy can be parameterized in any way, as long as $\pi(a|s,\theta)$ is differentiable with respect to its parameters, that is, as long as $\nabla\pi(a|s,\theta)$ exists and is finite for all $s \in \mathcal{S}, a \in \mathcal{A}$, and $\theta \in \mathbb{R}^m$. Moreover, we generally require the policy to be stochastic, and never deterministic, to ensure proper exploration of the environment.

Policy Gradient methods have proved to perform better than action-value methods in tasks with continuous action spaces; however, as in our case, if the action space is not too large, then it is usually preferred to form parametrized numerical preferences $h(s,a,\theta) \in \mathbb{R}$ for each state-action pair: the actions with the highest preferences in each state are given the highest probabilities of being selected using an exponential soft-max distribution:

$$\pi(a|s,\theta) := \frac{e^{h(s,a,\theta}}{\sum_b e^{s,b,\theta}}. \tag{2.9}$$

We call this kind of policy parametrization *soft-max in action preferences*.

2.2.2 Policy Gradient Theorem

In this section, we will present what is the base for all Policy Gradient (PG) methods, which is the *Policy Gradient Theorem* (PGT). As we have mentioned previously, in PG methods, we perform gradient ascent in J, which is a performance measure for the parametrized policy $\pi(a|s,\theta)$. We now define such a performance measure as

$$J(\theta) := \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t \cdot R_{t+1} \right]. \tag{2.10}$$

For the state-value and action-value function we use the definitions in (1.9) and (1.10), respectively.

Note that $J(\theta), V^\pi,$ and Q^π are all measures of expected returns. However, they differ in that $J(\theta)$ is the expected return when following policy π averaged over all states $s \in \mathcal{S}$ and all actions $a \in \mathcal{A}$, while $V^\pi(s)$ is the expected return for a specific state $s \in \mathcal{S}$ when following policy π, and $Q^\pi(s,a)$ is the expected return for a specific state $s \in \mathcal{S}$ and a specific action $a \in \mathcal{A}$ when following a specific policy π. We now introduce another function: the *advantage function*, which captures how much more value one particular action provides relative to the average value across actions for a given state. Such a function plays a crucial role in reducing the variance of PG algorithms and is defined as

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s). \tag{2.11}$$

Furthermore, we will introduce a function that denotes the probability of going from state s to s' in t steps by following policy π, namely $p(s \to s', t, \pi)$. Using such a function we can reformulate the expression for J in (2.10) as

$$\begin{aligned} J(\theta) &= \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t \cdot R_{t+1}\right] = \sum_{t=0}^{\infty} \gamma^t \mathbb{E}_\pi[R_{t+1}] \\ &= \sum_{t=0}^{\infty} \gamma^t \cdot \sum_{s\in\mathcal{S}} \left(\sum_{S_0\in\mathcal{S}} p_0(S_0)\cdot p(S_0 \to s, t, \pi)\right) \cdot \sum_{a\in\mathcal{A}} \pi(a|s,\theta)\cdot r(s,a) \\ &= \sum_{s\in\mathcal{S}} \left(\sum_{S_0\in\mathcal{S}} \sum_{t=0}^{\infty} \gamma^t \cdot p_0(S_0)\cdot p(S_0 \to s, t, \pi)\right) \cdot \sum_{a\in\mathcal{A}} \pi(a|s,\theta)\cdot r(s,a). \end{aligned} \tag{2.12}$$

We can now define a reformulated performance measure as follows.

Definition 2.1. *Let*

$$\rho^\pi(s) = \sum_{S_0\in\mathcal{S}} \sum_{t=0}^{\infty} \gamma^t \cdot p_0(S_0)\cdot p(S_0 \to s, t, \pi),$$

then we can reformulate the performance measure J as

$$J(\theta) = \sum_{s \in \mathcal{S}} \rho^{\pi}(s) \cdot \sum_{a \in \mathcal{A}} \pi(a|s, \theta) \cdot r(s, a).$$

In the above definition, we refer to $\rho^{\pi}(s)$ as the *Discounted-Aggregate State-Visitation* measure; it is important to note that the distribution of states under the ρ^{π} measure is an *improper*[10] distribution reflecting the relative likelihood of occurrence of states on a trace experience. We introduced the notion of improper distribution of states under the measure ρ^{π} in order to use the notation of expected value under this distribution when needed.

We are now ready to present the PGT, which provides a powerful formula for the gradient of $J(\theta)$ with respect to θ, thus allowing us to formulate update rules to perform Gradient Ascent.

Theorem 2.3 (Policy Gradient Theorem[11])**.**

$$\nabla_{\theta} J(\theta) = \sum_{s \in \mathcal{S}} \rho^{\pi}(s) \cdot \sum_{a \in \mathcal{A}} \nabla_{\theta} \pi(a|s, \theta) \cdot Q^{\pi}(s, a).$$

2.2.3 REINFORCE

The first PG algorithm we are going to present is the REINFORCE algorithm, Williams (1992). It is important to point out that in Stochastic Gradient Ascent, we need to obtain samples such that the expectation of the sample gradient is proportional to the actual gradient of the performance measure as a function of the parameter. In accomplishing this task, the PGT is crucial as it gives an exact expression proportional to the gradient; all that is now needed is a way of sampling whose expectation equals or approximates this expression. We can notice that the right-hand side of the PGT is a sum over states weighted by how often the states

[10] We say that it is an improper distribution to convey the fact that $\sum_{s \in \mathcal{S}} \rho^{\pi}(s) \neq 1$, that is, the distribution is not normalized.

[11] The proof of Theorem 2.3 can be found at Sutton, Barto (2018).

occur under the target policy π, thus we can rewrite the equation in theorem 2.3 as:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[\sum_a Q^\pi(S_t, a) \nabla_\theta \pi(a|S_t, \theta) \right]. \tag{2.13}$$

We continue by introducing A_t in the same way as we introduced S_t in equation 2.13, that is, by replacing a sum over the random variable's possible values with an expectation under π, and then sampling the expectation. Continuing from equation 2.13 we have

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_\pi \left[\sum_a \pi(a|S_t, \theta) Q^\pi(S_t, a) \frac{\nabla_\theta \pi(a|S_t, \theta)}{\pi(a|S_t, \theta)} \right] \\ &= \mathbb{E}_\pi \left[Q^\pi(S_t, A_t) \frac{\nabla_\theta \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \right] \\ &= \mathbb{E}_\pi \left[G_t \frac{\nabla_\theta \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \right]. \end{aligned} \tag{2.14}$$

Where the last expression is due to the fact that $\mathbb{E}_\pi[G_t|S_t, A_t] = Q^\pi(S_t, A_t)$. The final expression is also what we needed to perform gradient ascent, a quantity that can be sampled on each time step whose expectation is equal to the gradient; the REINFORCE update is then given by

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla_\theta \log \pi(A_t|S_t, \theta_t)^{12}. \tag{2.15}$$

[12]Notice that writing $\nabla \ln x$ is equivalent to $\frac{\nabla x}{x}$

Algorithm 2.1 REINFORCE, Source: Sutton, Barto (2018)

Input: a differentiable policy parametrization $\pi(a|s,\theta)$
Parameter: the step size $\alpha > 0$
Initialize: policy parameter $\theta \in \mathbb{R}^d$
for each episode **do**
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$**, following** $\pi(\cdot|\cdot,\theta)$
  for each s dotep of the episode
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\theta \leftarrow \theta + \alpha\gamma^t G \nabla_\theta \ln \pi(A_t|S_t,\theta)$
  end for
end for

REINFORCE with Baseline

The PGT (2.3) can be generalized to include a comparison of the action value to a *baseline* denoted $b(s)$, as presented in the following theorem.

Theorem 2.4 (Policy Gradient Theorem with Baseline)**.**

$$\nabla_\theta J(\theta) = \sum_{s\in\mathcal{S}} \rho^\pi(s) \cdot \sum_{a\in\mathcal{A}} \left(Q^\pi(s,a) - b(s)\right) \nabla_\theta \pi(a|s,\theta). \tag{2.16}$$

It is important to notice that the baseline can be any function, including a random variable, as long as it does not vary with the action a. In fact, if such a condition holds, the equation remains valid because the subtracted quantity is zero:

$$\sum_{a\in\mathcal{A}} b(s)\nabla_\theta \pi(a|s,\theta) = b(s)\nabla_\theta \sum_{a\in\mathcal{A}} \pi(a|s,\theta) = b(s)\nabla_\theta 1 = 0.$$

The PGT with baseline (2.4) can be used to derive the following update rule

$$\theta_{t+1} := \theta_t + \alpha \left(G_t - b(S_t)\right) \nabla_\theta \log \pi(A_t|S_t,\theta_t). \tag{2.17}$$

One trivial choice for the baseline can be an estimate of the state value, $\widehat{V}^\pi(S_t, w), w \in \mathbb{R}^m$.

Algorithm 2.2 REINFORCE with Baseline, Source: Sutton, Barto (2018)

Input: a differentiable policy parametrization $\pi(a|s,\theta)$
Input: a differentiable state-value function parametrization $\widehat{V}^{\pi}(s,w)$
Parameter: the step size $\alpha^{\theta} > 0, \alpha^{w} > 0$
Initialize: policy parameter $\theta \in \mathbb{R}^d$ and state-value weights $w \in \mathbb{R}^m$
for each episode **do**
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$**, following** $\pi(\cdot|\cdot,\theta)$
    for each s dotep of the episode
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
        $\delta \leftarrow G - \widehat{V}(S_t, w)$
        $\theta \leftarrow \theta + \alpha\gamma^t \delta \nabla_{\theta} \ln \pi(A_t|S_t,\theta)$
    end for
end for

2.3 Actor-Critic Methods

It is important to begin this section by explaining why REINFORCE with Baseline cannot be considered an actor-critic method: in fact, REINFORCE uses the state-value function only as a baseline and not as a critic. The key difference is that the value estimate for a state is not updated from the estimated values of the subsequent states (*bootstrapping*), but rather is used as a baseline for the state whose estimate is being updated. Notice that using bootstrapping techniques introduces bias and dependence on the quality of the approximation for the critic while reducing variance and accelerating learning. One of the most common approaches is to use the critic's value function as the baseline, which results in the TD error of the estimated value function.

2.3.1 Generalized Advantage Estimation

In this section, we will present a method to estimate the advantage function as in Schulman et al. (2018). The objective is to produce an accurate estimate $\widehat{A}_t$ if the discounted advantage function $A^{\pi}(s_t, a_t)$. Such an estimate will then be used to construct a PG estimator. Now, if we let V be an approximate value function, we

can then define the TD error of V with discount γ as

$$\delta_t^V := r_t + \gamma V(S_{t+1}) - V(S_t).$$

We can consider δ_t^V as an estimate of the advantage of action A_t, as if we have a value function estimate such that $V = V^\pi$, then it is a γ-just[13] advantage estimator and an unbiased estimator of A^π[14]. Considering now the sum of k of these δ terms, denoted $\widehat{A}_t^{(k)}$, we will see that $\widehat{A}_t^{(k)}$ involves a $k-step$ estimate of the returns, minus a baseline term $-V(S_t)$ [15]. It can be noticed that the bias becomes smaller as $k \to \infty$, in fact, taking $k \to \infty$, we get

$$\widehat{A}_t^{(\infty)} = \sum_{l=0}^{\infty} \gamma^l \delta_{t+l}^V = -V(S_t) + \sum_{l=0}^{\infty} \gamma^l r_{t+l}. \tag{2.18}$$

The Generalized Advantage Estimator GAE(γ, λ) is then defined as the exponentially-weighted average of these k-step estimators:

$$\begin{aligned}
\widehat{A}_t^{GAE(\gamma,\lambda)} &:= (1-\lambda)\left(\widehat{A}_t^{(1)} + \lambda \widehat{A}_t^{(2)} + \lambda^2 \widehat{A}_t^{(3)} + \dots\right) \\
&= (1-\lambda)(\delta_t^V + \lambda(\delta_t^V + \gamma\delta_{t+1}^V + \lambda^2(\delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V + \dots) \\
&= (1-\lambda)(\delta_t^V(1 + \lambda + \lambda^2 + \dots) + \gamma\delta_{t+1}^V(\lambda + \lambda^2 + \lambda^3 + \dots) \\
&\quad + \gamma^2\delta_{t+2}^V(\lambda^2 + \lambda^3 + \lambda^4 + \dots) + \dots) \\
&= (1-\lambda)\left(\delta_t^V\left(\frac{\lambda}{1-\lambda}\right) + \gamma\delta_{t+1}^V\left(\frac{\lambda}{1-\lambda}\right) + \gamma^2\delta_{t+2}^V\left(\frac{\lambda}{1-\lambda}\right)\right) \\
&= \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+1}^V.
\end{aligned} \tag{2.19}$$

[13] A definition of what is a γ-just advantage estimator is out of scope for this work. For a definition of γ-just, see Schulman et al. (2018).

[14] The estimator is unbiased as

$$\begin{aligned}
\mathbb{E}[\delta_t^{V^\pi} | S_{t+1}] &= \mathbb{E}[r_t + \gamma V^\pi(S_{t+1}) - V^\pi(S_t) | S_{t+1}] \\
&= \mathbb{E}[Q^\pi(S_t, A_t) - V^\pi(S_t) | S_{t+1}] = A^\pi(S_t, A_t).
\end{aligned}$$

[15] For the detailed proof, see Schulman et al. (2018)

2.3.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO), introduced by Schulman et al. (2017b), is a method of the PG family, more specifically of the Trust Region one. PPO, in fact, addresses those that are the the main problems of 'vanilla' PG methods, such as large policy updates and therefore training instability. In Trust Region methods, the objective function is maximized by solving a constrained optimization problem using the Kullback-Leibler (KL) divergence.

Definition 2.2 (Kullback-Leibler Divergence)**.** *The Kullback-Leibler divergence is a measure of the distance between a probability distribution P and a reference probability distribution Q. It is defined as*

$$D_{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x) \log \left(\frac{P(x)}{Q(x)} \right).$$

Thus, in Trust Region Policy Optimization (TRPO), Schulman et al. (2017a), we solve one of the following constrained optimization problem

$$\begin{gathered} \max_{\theta} \left\{ \mathbb{E}_t \left[\frac{\pi_{\theta(A_t|S_t)}}{\pi_{\theta_{old}(A_t|S_t)}} \widehat{A}_t \right] \right\} \\ \text{subject to } \mathbb{E}_t \left[D_{KL}[\pi_{\theta_old}(\cdot|S_t), \pi_\theta(\cdot|S_t)] \right] \leq \delta, \end{gathered} \tag{2.20}$$

or

$$\max_{\theta} \left\{ \mathbb{E}_t \left[\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)} \widehat{A}_t - \beta D_{KL}[\pi_{\theta_{old}}(\cdot|S_t), \pi_\theta(\cdot|S_t)] \right] \right\}. \tag{2.21}$$

In practice, the constrained problem described in Equation 2.20 is usually more popular, as it does not involve the calibration of the penalization parameter β that is present in Equation 2.21.

In PPO, a clipped surrogate objective function is used. Such an objective function is derived from the Conservative Policy Iteration:

$$\mathbb{E}_t \left[\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)} \widehat{A}_t \right]. \tag{2.22}$$

Now, letting $r_t(\theta) = \frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)}$, we can define the clipped surrogate objective as

$$L_t^{CLIP}(\theta) = \mathbb{E}_t \left[min \left(r_t(\theta)\widehat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\widehat{A}_t \right) \right]. \tag{2.23}$$

where $clip(\cdot, 1-\epsilon, 1+\epsilon)$ modifies the argument by *clipping* in the interval $[1-\epsilon, 1+\epsilon]$. It is important to notice that this new objective function is equal to the one of Conservative Policy Iteration to first order around θ_{old}, Schulman et al. (2017b), but they strand away as θ moves away from θ_{old}. Now that we have defined this new objective function, we can present the final objective function, which makes use of a learned value function in order to reduce variance, and an entropy[16] bonus, as follows.

$$L_t^{PPO}(\theta) = \mathbb{E}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 H(\pi_\theta(S_t)) \right]. \tag{2.24}$$

Furthermore, in PPO we run the policy for T timesteps, where T is usually much less than the episode length, and then use the collected samples to perform an update to the policy using minibatch SGD for K epochs.

[16]The entropy of a Random Variable is defined as

$$H(X) := -\sum_{x \in \mathcal{X}} p(x) \log p(x) = \mathbb{E}[-\log p(X)]$$

Chapter 3

Methodology

3.1 The Model

The proposed Reinforcement Learning model is based on Proximal Policy Optimization (PPO), which we presented in Section 2.3.2, and whose pseudo code we will now present.

The algorithm will be implemented in Python, and is composed of three objects:

- the Agent;
- the Environment;
- the Data Loader;

3.1.1 The Agent

The Agent is the backbone of our Proximal Policy Optimization (PPO) framework. Within the model, the Agent takes on the crucial role of interfacing with the Environment, handling the collection of rewards, and, subsequently, carrying out the optimization processes for both the Actor and the Critic components of the model. Being the central decision-maker, the Agent is the one that determines

when to buy, sell, or hold the asset based on the Actor's policy parametrization gained from its interactions with the Environment.

We constructed the Actor and the Critic as Fully Connected Artificial Neural Networks (FC-ANN). The Actor, which is represented by its policy parametrization, is the one that picks the optimal actions to take in the Environment, effectively defining the trading strategy. In contrast, the Critic, which is represented by its value network, provides feedback by estimating the value of state-action pairs, aiding in assessing the effectiveness of chosen actions. It is important to stress that the Critic is only used in the training phase, while the Actor is used in the test phase by the Agent. The Actor and the Critic are continuously refined and adapted through interactions with the financial Environment to maximize returns while managing risk. The collaborative efforts of these components, under the Agent's guidance, form the core of our PPO-based trading framework, enabling adaptive and informed decision-making in the complex landscape of financial markets.

3.1.2 The Environment

In our PPO-based trading model, the Environment assumes a central role, being the link between the Agent and the financial markets. One of the most crucial functions that the Environment carries out is providing real-time observations from the state to the Agent: these observations encapsulate information, including market conditions, asset prices, technical indicators, and relevant data points. These observations allow the Agent to make informed decisions that adapt to the dynamic nature of financial markets.

Furthermore, the Environment receives the Agent's trading actions; these actions dictate when to buy, sell, or hold the asset within the simulated trading environment. This exchange between the Agent and the Environment mirrors the real-world trading experiences.

However, the Environment's role is far more important than being a mere link; in

fact, it assumes the essential task of computing the rewards, which the Agent then uses to optimize the parameters of the networks. The rewards play a crucial role, as they are the means through which the Agent receives feedback on its actions, thus making it possible to reinforce behaviors that lead to profitable trades and discourage trades that result in financial losses.

The Environment operates as an incubator for the Agent's trading strategies, refining them through a continuous feedback loop; this iterative process is the core of optimizing the Agent's decision-making capabilities.

In order to avoid *overfitting*, the Environment switches data set every n iterations: the dataset that will be used for the next n iterations is chosen as follows. Let $X_{i,t}$ be a counting process that increases by 1 each time a dataset is used, where $i = 1, 2, \ldots, K$, with K being the number of data sets, and where $t = 0, 1, \ldots, T$ is the training iteration. The Environment then picks the next dataset as the one with the lowest value in its respective counting process; in the case of a tie, we split it randomly.

3.1.3 The Data Loader

The Data Loader in our reinforcement learning model is a component that plays a pivotal role in the entire system's functionality. It is an interface responsible for acquiring, transforming, and preparing financial data from external sources. This object not only fetches the data but also undertakes preprocessing steps, as mentioned in Section 3.2, to ensure the data is well-suited for the subsequent stages of the model.

Its first responsibility is retrieving a wide array of financial information, such as stock prices and trading volumes, depending on the use case. After obtaining this raw data, the Data Loader performs a comprehensive data-wrangling process, including handling missing values, dealing with outliers, performing feature engineering, and normalizing data to create relevant input features for the Agent.

The Data Loader is the cornerstone of the model's data pipeline, bridging the gap between external data providers and the Agent: in fact, its handling of financial data facilitates more effective learning and enhances the model's capacity to deal with the complexities and uncertainties of the financial markets. A well-designed and versatile Data Loader is thus crucial to ensure the model's success in adapting to changing market conditions and making informed investment choices.

3.1.4 Network Architecture

Both the Actor and the Critic have been implemented as an FC-ANN, specifically, we opted for a 3-layer, 128 neurons per layer architecture. The only difference between the Actor's and the Critic's network stands in the final layer: in fact, the Actor presents 3 neurons with SoftMax activation, one for each action, while the Critic presents a single linear layer. As for the activation function of the hidden layer, we choose to experiment with a variety of activation functions, such as the ReLU and the Tanh, depicted below.

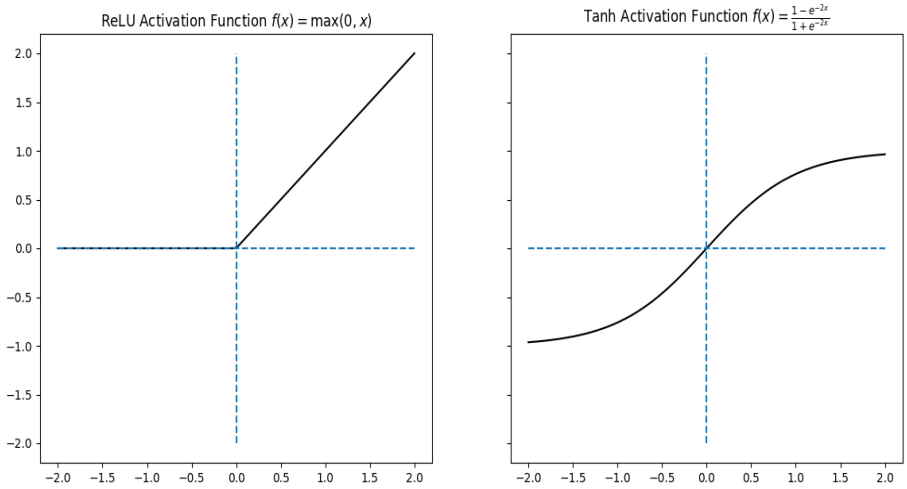


Fig. 3.1: ReLU and Tanh Activation Functions

3.2 Data Processing

In this section, we present the process that the raw data goes through before being used to train the model. The data we use is going to be used in the Proximal Policy Optimization algorithm that we implemented. The objective is to model the financial markets as an MDP so that we can apply the algorithms and methods we presented in the previous sections to stock trading. Specifically, we have that the state space $\mathcal{S}$ is composed of the close, high, open, and low price, the technical features, past returns, and the gross position of the agent. The action space $\mathcal{A}$ is just a three-dimensional vector, with each position representing one action in $\{short, flat, long\}$, we are thus dealing with a discrete action space. For the reward R_t, we decided to opt for two formulations: the first one is just the percentage change in portfolio value, and the second one is the Sharpe Ratio of the previous trading month.

3.2.1 Tickers

The choice of the tickers is related to the way the Environment interacts with the agent: in fact, since the Environment switches data set every n iterations, we can use this property to our advantage to both reduce overfitting and to bring diversity in the Agent's training. Thus, in order to bring diversity, we choose to pick several stocks, some broad market indices, and a commodity index. It is however very important to avoid as much as possible having redundant information, which in our setting translates to having highly correlated time series.

3.2.2 Features

In this section, we will go over the Feature Engineering (FE) process, which allows us to create many features from a single financial time series. These features in finance are often referred to as *technical indicators*; they are mainly used to remove

noise from and highlight some properties of the time series itself. Before presenting the features, we will introduce some notation that is used throughout the section. The opening price at each time step is denoted as O_t, the closing price C_t, the highest price H_t, and the lowest price L_t, with the Trading Volume each day being denoted as V_t.

Simple Moving Average

The *Simple Moving Average* (SMA) is a technical indicator that is computed as

$$SMA_t(w) = \frac{\sum_{t=T-w}^{T} x_t}{w}, \tag{3.1}$$

where w is the so-called *window size*, which identifies the number of periods that are included in the calculation of the SMA, and x_t is the observation at time t of our process. In using an SMA, the choice of w is of crucial importance: in fact, choosing a small value for w, such as 9 or 10 will lead our SMA to be extremely reactive, but also very noisy; on the other hand, choosing a larger value for w, such as 21, or even 240, will lead to a less reactive but also less noisy indicator. The criterion that should be used when using an SMA as a technical indicator is actually very intuitive: if at time t the process we are observing has a value x_t higher than the SMA, then we can consider the process to be in an ascending trend. Clearly, the larger the value for the window size, the larger the power of the signal. In the figure below it is possible to how two different SMAs react to price changes.

Fig. 3.2: SMA 10, SMA 242 and the SP500 Index

Exponential Moving Average

The *Exponential Moving Average* (EMA) is a technical indicator of the same family of the SMA, that is, the moving average one. The difference with SMA is the way in which the rolling average is computed, in fact, in EMA, the weights of the past observation decay exponentially, whereas in the SMA there is no weight decay. EMA can be calculated with the following recursive formula:

$$\mathrm{EMA}_t = \lambda \cdot x_t + (1 - \lambda) \cdot \mathrm{EMA}_{t-1}, \tag{3.2}$$

The parameter $\lambda \in [0, 1]$ is the decay factor, which represents the rate at which the weights of past observations decay as you move further back in time; a higher λ gives more weight to recent data, while a lower λ gives more equal weight to historical data.

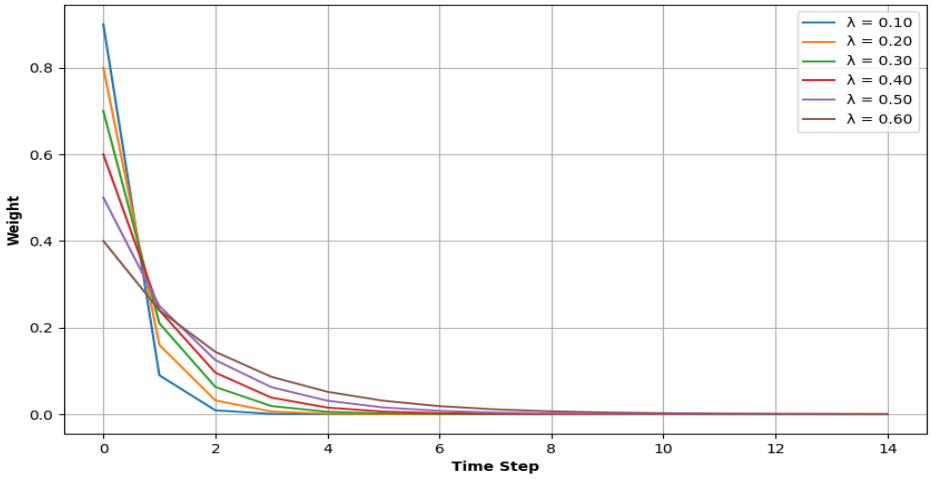


Fig. 3.3: Weight decay in EMA for different values of λ

Moving Average Convergence Divergence

The *Moving Average Convergence Divergence* (MACD) is a crucial technical indicator in technical analysis. It provides valuable insights into the momentum and direction of asset prices. We can calculate the MACD from two exponential moving averages (EMAs), typically a 12-period EMA and a 26-period EMA. MACD is then calculated as the difference between these EMAs and is used to identify potential trend reversals or continuations: in fact, when the MACD crosses above the signal line, it generates a bullish signal, suggesting upward momentum; on the other hand, we can consider a crossover below the signal line to be bearish, indicating a possible downtrend. Additionally, the MACD histogram, which represents the difference between the MACD and signal line, visually represents momentum strength. Traders often use these MACD components to make informed decisions regarding market entry and exit points. The equations for MACD components are

as follows:

$$\begin{aligned} \text{MACD} &= \text{EMA}_{12} - \text{EMA}_{26}, \\ \text{Signal Line} &= \text{EMA}_{9}(\text{MACD}), \\ \text{Histogram} &= \text{MACD} - \text{Signal Line}. \end{aligned} \tag{3.3}$$

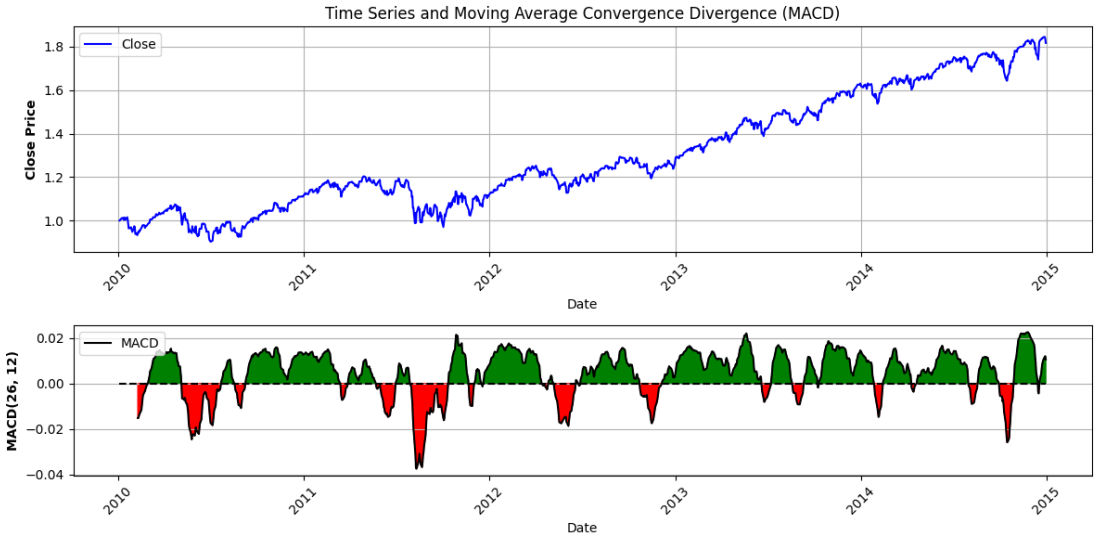


Fig. 3.4: Smoothed Histogram of the MACD(26, 12) compared to the SP500 index

On Balance Volume

The *On Balance Volume* (OBV) is the first indicator of the Volume family. The OBV technical indicator is a popular tool in technical analysis. It helps traders and analysts assess the strength of a price trend by measuring the cumulative volume flow. We can calculate OBV by adding the volume on days when the price increases and subtracting the volume when the price decreases. This indicator effectively shows buying or selling pressure behind a price movement. When OBV confirms a price trend, it can provide valuable insights, indicating potential trend reversals or continuations. Analysts often use OBV with other technical indicators to make more informed trading decisions. In mathematical terms, we can express

the OBV as:

$$OBV_t = OBV_{t-1} + \begin{cases} V_t, & \text{if } C_t > C_{t-1} \\ -V_t, & \text{if } C_t < C_{t-1} \\ 0, & \text{if } C_t = C_{t-1} \end{cases}. \tag{3.4}$$

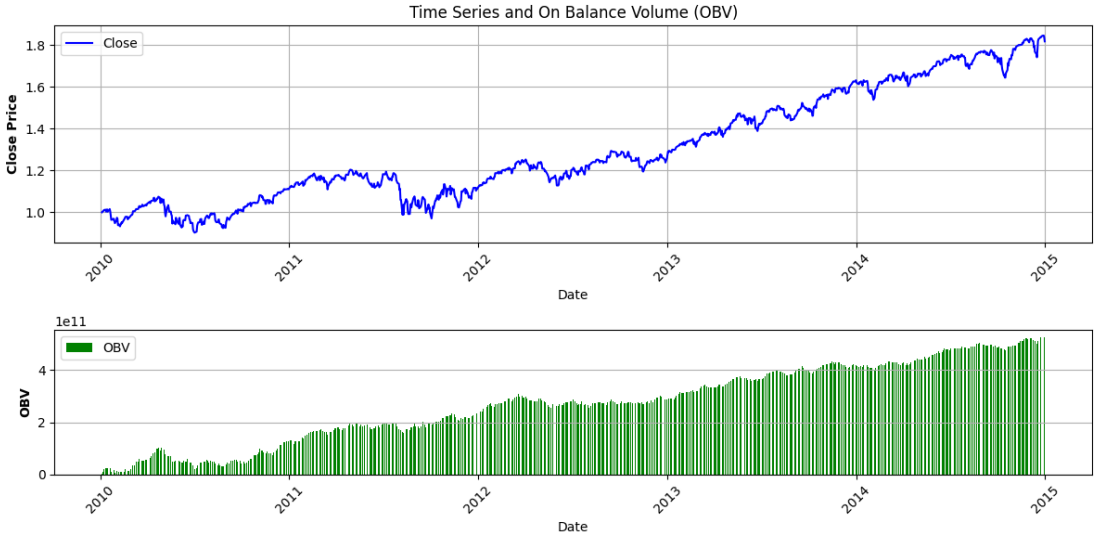


Fig. 3.5: OBV compared to the SP500 index

Money Flow Index

The *Money Flow Index* (MFI) is another popular tool in technical analysis, offering insights into the strength of a price trend and potential reversal points. MFI combines price and volume data to determine overbought and oversold conditions. Its calculation involves the average price of an asset over a specified period, the typical price, and the money flow. The money flow (MF) is the product of the typical price (TP) and the trading volume, and the MFI is then calculated based on the Money Flow Ratio (MFR), which is the ratio of positive money flow (MF^+) to negative money flow (MF^-). MFI values range from 0 to 100, with overbought conditions typically indicated when the MFI surpasses 70 and oversold conditions when it falls below 30. Traders use the MFI to identify potential trend reversals or

continuations, the divergence between price and MFI signals, and to make more informed decisions about entry and exit points in the market. The equations for calculating the MFI components are as follows:

$$\begin{aligned}
\mathrm{TP}_t &= \frac{H_t + L_t + C_t}{3}, \\
\mathrm{MF}_t &= \mathrm{TP}_t \times V_t, \\
\mathrm{MF}_t^+ &= \sum_{i=t-n}^{t} \mathbb{I}_{\{\mathrm{TP}_i > \mathrm{TP}_{i-1}\}} \times \mathrm{MF}_i, \\
\mathrm{MF}_t^- &= \sum_{i=t-n}^{t} \mathbb{I}_{\{\mathrm{TP}_i < \mathrm{TP}_{i-1}\}} \times \mathrm{MF}_i, \\
\mathrm{MFR}_t &= \frac{\mathrm{MF}_t^+}{\mathrm{MF}_t^-}, \\
\mathrm{MFI}_t &= 100 - \frac{100}{1 + \mathrm{MFR}_t}.
\end{aligned} \tag{3.5}$$

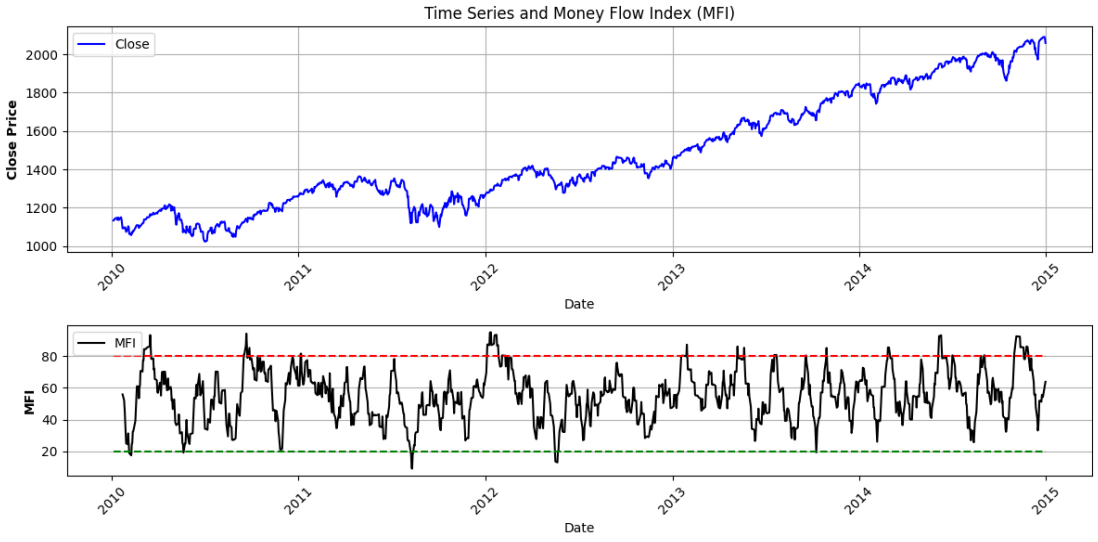


Fig. 3.6: MFI compared to the SP500 index

Chaikin Money Flow

The *Chaikin Money Flow* (CMF) is a popular technical indicator used by traders to assess the strength and direction of money flow in a financial instrument, typically a

stock. It helps traders make informed decisions by providing insights into whether a security is under buying or selling pressure. The CMF is calculated using two key components: the Money Flow Multiplier (MFM) and the Money Flow Volume (MFV).

$$\begin{aligned} \mathrm{MFM}_t &= \frac{(C_t - L_t) - (H_t - C_t)}{H_t - L_t)}, \\ \mathrm{MFV}_t &= \mathrm{MFM}_t \times V_t, \\ \mathrm{CMF}_t &= \frac{\sum_{i=t-n}^{t} \mathrm{MFV}_i}{\sum_{i=t-n}^{t} V_i}. \end{aligned} \tag{3.6}$$

Note that the result is often smoothed with a moving average to generate a more easily interpretable indicator. Traders use CMF to identify potential trend reversals, the divergence between price and money flow, and overbought or oversold conditions. A positive CMF suggests accumulation and potential bullish momentum, while a negative CMF indicates distribution and potential bearish pressure.

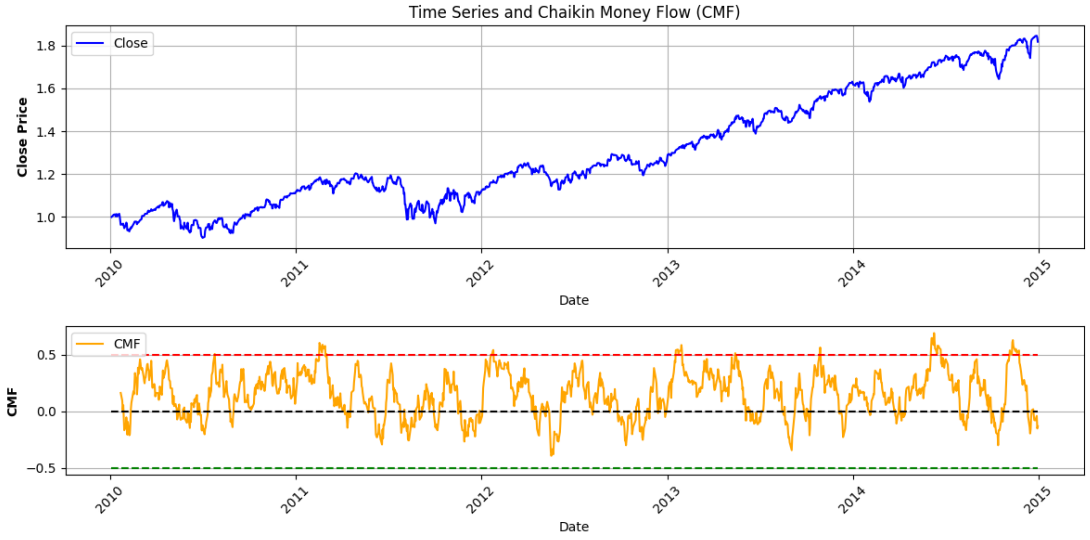


Fig. 3.7: CMF compared to the SP500 index

Commodity Channel Index

The *Commodity Channel Index* (CCI) is a technical indicator developed by Donald Lambert in the 1980s; CCI primarily identifies overbought or oversold conditions in an asset and potential trend reversals. We can calculate CCI using the following formula:

$$\begin{aligned} \mathrm{CCI}_t &= \frac{(\mathrm{TP}_t - \mathrm{SMA}(n))}{0.015 \times \text{Mean Deviation}}, \\ \text{where Mean Deviation} &= \sum_{i=t-n}^{t} \frac{(\mathrm{TP}_i - \mathrm{SMA}(n)_i)}{n}. \end{aligned} \tag{3.7}$$

Traders typically use CCI values above +100 to indicate overbought conditions, suggesting a potential bearish reversal, and values below -100 to signify oversold conditions, hinting at a potential bullish reversal. CCI is a valuable tool to complement other technical and fundamental analysis methods in making well-informed trading decisions.

Detrended Price Oscillator

The *Detrended Price Oscillator* (DPO) is a technical indicator that helps traders identify short-term price trends by removing the long-term trend component from a price series. To calculate the DPO, one has to select a specific period to calculate the DPO, and then calculate the closing prices' simple moving average (SMA) over the chosen period. Shift the SMA backward by half the selected period, creating a displaced moving average, then subtract the displaced moving average from the current closing price to obtain the DPO value. The resulting DPO values oscillate around zero, and traders use them to identify potential overbought or oversold conditions in the short term. When the DPO crosses above zero, it suggests that short-term prices are above the long-term average, indicating a potential bullish trend. Conversely, short-term prices are below the long-term average when the DPO crosses zero, indicating a potential bearish trend. Traders use these signals

with other technical and fundamental analyses to make informed trading decisions.

Relative Strength Index

The *Relative Strength Index* (RSI) is a technical indicator traders use to assess the strength and potential direction of a financial instrument's price movement. It oscillates between 0 and 100 and is typically employed to identify overbought or oversold conditions in the market. We consider an RSI above 70 as signaling an overbought asset, suggesting a potential sell signal, while an RSI below 30 as an oversold asset, signaling a potential buy opportunity. We can calculate the RSI as follows:

$$\begin{aligned} \mathrm{RSI}_t &= 100 - \frac{100}{(1 + \mathrm{RS}_t)}, \\ \text{where } \mathrm{RS}_t &= \frac{\sum_{i=t-14}^{t} |r_t| \times \mathbb{I}_{\{r_t > 0\}}}{\sum_{i=t-14}^{t} |r_t| \times \mathbb{I}_{\{r_t < 0\}}}. \end{aligned} \tag{3.8}$$

In the above calculation, r_t indicates the percentage return between C_{t-1} and C_t.

Average True Range

The *Average True Range* (ATR) is a technical indicator that traders use to assess market volatility. To calculate the ATR, first determine the true range (TR) for each period and then calculate the average of these values; this average is the ATR, representing the average volatility over that specified period.

$$\begin{aligned} \mathrm{TR}_t &= \max \left\{ \begin{array}{c} H_t - L_t, \\ |H_t - C_{t-1}|, \\ |L_t - C_{t-1}| \end{array} \right\}, \\ \mathrm{ATR}_t &= \frac{1}{n} \sum_{i=t-n}^{t} \mathrm{TR}_i. \end{aligned} \tag{3.9}$$

A higher ATR indicates greater market volatility, while a lower ATR suggests lower volatility. Traders utilize the ATR to set stop-loss and take-profit levels, determine

position sizes, and assess overall risk, allowing them to make more strategic and informed trading decisions in various market conditions.

3.2.3 Data Normalization

The last step in the Data Preparation process is to normalize our data; this is a widespread practice, as it makes training faster and helps to avoid getting stuck in a local minimum, Bishop (1995). In practice, the two most popular transformations that are applied to the data in order to scale it are the *Min-Max Normalization* and the *Z-Score Normalization*, Patro, Sahu (2015), Pires et al. (2020).

Min-Max Normalization

The Min-Max normalization is one of the most popular transformations applied to data, and, given a vector of data $\mathcal{D}$, and the interval $[a, b]$, that we want our transformed data to be in, we can define the transformation as

$$\mathcal{D}' = \frac{\mathcal{D} - \min(\mathcal{D})}{\max(\mathcal{D}) - \min(\mathcal{D})} \times (b - a) + a, \tag{3.10}$$

where each calculation is performed element-wise.

Z-Score Normalization

The Z-score normalization is the second data transformation we present; again, given a vector of data $\mathcal{D}$, we define the transformation as

$$\mathcal{D}' = \frac{\mathcal{D} - mean(\mathcal{D})}{std(\mathcal{D})}, \tag{3.11}$$

where $mean(\mathcal{D})$ and $std(\mathcal{D})$ represent the sample mean and sample standard deviation, respectively.

The two transformations we have just presented are both valid ones, however, it is important to make some remarks: in fact, if we use the one in Equation 3.2.3, we must be certain that the minimum and the maximum values of the dataset

will not change in the test set; if this was the case, we would find ourselves with some values outside the interval $[a, b]$, which would probably be considered as influential observations by the algorithm. On the other hand, if we dispose of only a small amount of data, the choice of the transformation in Equation 3.11 might not be optimal, as we might not be able to rely on a good sample estimate for the mean and the variance; furthermore, when using the z-score transformation we also assume that the distribution from which our data is sampled does not change between the training and the test set.

3.3 Performance Measures

In this section, we will present the metrics we will use to evaluate the performance of the PPO-Agent strategy over time. In fact, in order to gauge if the model's Agent is actually achieving good performance, we will compare its metrics with those of a simple buy and hold (B&H) strategy. We will now rapidly go through the metrics that we will adopt.

Sharpe Ratio

The *Sharpe Ratio* (SR) is a risk-adjusted performance measure that relates the expected return and the standard deviation of returns of a given portfolio. Informally, it shows how much compensation the investors get for an extra unit of risk, Sharpe (1966). For the return on a portfolio R_P and the return on a benchmark R_B, it is calculated as

$$\text{SR} = \frac{\overline{D}}{\sigma_D}, \tag{3.12}$$

where

$$D_t = R_{P,t} - R_{B,t},$$
$$\overline{D} = \frac{1}{T}\sum_{t=1}^{T} D_t,$$
$$\sigma_D = \sqrt{\frac{\sum_{t=1}^{T}\left(D_t - \overline{D}\right)^2}{T-1}}.$$

Clearly, a risk-averse investor will prefer to invest in a portfolio with a higher SR, since it rewards more the risk they take on by investing.

Maximum Drawdown

The *maximum drawdown* (MDD) is the worst loss among successive declines from peaks to troughs during a given period. MDD is the worst-case scenario for an investor who starts his/her investment in the period. Investors prefer smaller MDDs to more significant drawdowns in portfolio performance, Choi (2021). Let $R(t,\tau)$ be the log-return between time t and time τ, then we define the MD as:

$$MDD = -\min_{\tau\in[0,T]}\left\{\min_{t\in[0,T]} R(t,\tau)\right\}. \tag{3.13}$$

Calmar Ratio

The *Calmar Ratio* (CR) was introduced in Young (1991). It is yet another risk-adjusted performance measure that can be used to judge a portfolio's returns. It is constructed similarly to the SR in Equation 3.12 but uses the MDD in Equation **??**, rather than the standard deviation. It is generally computed using the average annual return over a period $[0,T]$, and the maximum drawdown in the same period. Letting $R_{t,t+12}$ denoting the annual return of a security, we can calculate the CR as:

$$\text{CR} = \frac{\sum_{t=1}^{T-12} R_{t,t+12}}{\text{MDD}}. \tag{3.14}$$

Sortino Ratio

The *Sortino Ratio* (STR), Rollinger, Hoffman (2013), is a risk-adjusted performance measure that is extremely similar to the SR in Equation 3.12: in fact, in the STR, we use the semi-standard deviation of the returns distribution, rather than the standard deviation. The semi-standard deviation is a Lower Partial Moment (LPM) [1], which simply calculates the standard deviation of the returns below a threshold τ. A sample estimate of semi-standard deviation for a threshold τ can be calculated as:

$$\text{SemiSD}(\tau) = \sqrt{\frac{\sum_{t=1}^{T} \max(\tau - r, 0)^2}{T}}. \tag{3.15}$$

The STR is then defined, in the same way as the SR, therefore we will use the same notations and definitions:

$$\text{STR} = \frac{\overline{D}}{\text{SemiSD}(\tau)}. \tag{3.16}$$

Note that the choice of τ clearly influences greatly the STR; therefore one should always specify the threshold that has been used for the calculation, in order to avoid any possible misunderstanding. A sensible choice for τ can be 0, therefore only accounting for negative returns, or the minimum acceptable return for the investor.

Value at Risk

Value at Risk (VaR) is a risk measure that tells us, given a specified confidence level, what is the maximum amount we can lose over one day. Let the confidence level be denoted as α, then the α-VaR can be calculated as follows: select a series of returns over a time period $[0, T]$; then we can calculate the quantile of the returns distribution given this confidence level, that is, find a return r^* such that:

$$\mathbb{P}\{r \leq r^*\} = 1 - \alpha,$$

[1] For more information and detail about LPMs, one can see LPM (1975)

then the Value at Risk is simply calculated as:

$$\text{VaR}(\alpha) = -r^*.$$

VaR has the property of being a *coherent*[2] risk measure that is, it has *monotonicity*, *translation invariance*, *homogeneity*, and *sub-addittivity*.

Conditional Value at Risk

Conditional Value at Risk(CVaR) or *Expected Shortfall* (ES), McNeil et al. (2015), is the last risk measure that we will present. CVaR helps us to gauge what is the average loss if the VaR is exceeded. In fact, since VaR is actually just a quantile of the returns distribution, it gives us no knowledge of what happens if we happen to go past the VaR: it can be said the VaR is the best of worse-case scenarios. For these reasons, we now present the CVaR. CVaR can be calculated as the average of the VaR that exceed our confidence level, that is in integral form:

$$\text{CVaR}(\alpha) = \frac{1}{\alpha}\int_{-\infty}^{\alpha} \text{VaR}_q dq. \tag{3.17}$$

Note that CVaR assumes neutrality of investors beyond the VaR level, in fact, it weighs all the returns past the VaR equally.

Robustness to Transaction and Borrowing Costs

The last part of our analysis of the Agent's strategy will involve a series of further tests to gauge how much the strategy is sensitive to transaction costs and borrowing costs, that is incurred when the Agent takes a short position on the asset, and it does so by borrowing at the current rate.

[2]For more details about the definition of coherent risk measure, one can see Artzner et al. (1999)

Chapter 4

Results

4.1 Preliminary Remarks

In this Chapter, we are going to present the results of applying the model to empirical data. It is important to note that these results were obtained with technology that is available to everyone: the algorithms were implemented in Python, the data was downloaded online from Yahoo! Finance, and the training and testing of the model was carried out on a device with the following specifics.

Component	Specifics
CPU	8 Cores, 16 threads @ 2.20GHz
GPU	NVIDIA GeForce GTX 1650 Max-Q, 4GB
RAM	16GB DDR4, 2933MHz, SODIMM

Table 4.1: Hardware Specifics

4.2 Data Summary

In this section, we are going to briefly present the data we have selected to train and test the model. The tickers we selected for the training are the following. We selected a training period of 8 years that goes from 2010-01-01 to 2018-01-01 and a test period of 3 years that goes from 2018-01-01 to 2021-01-01. The test period also

goes over 2020, the year of the COVID-19 pandemic, which caused great distress to the financial markets, causing a period of high volatility and negative returns; however, since a *black swan* event is always unpredictable and unexpected, it was highly informative to include 2020 in the test for our model.

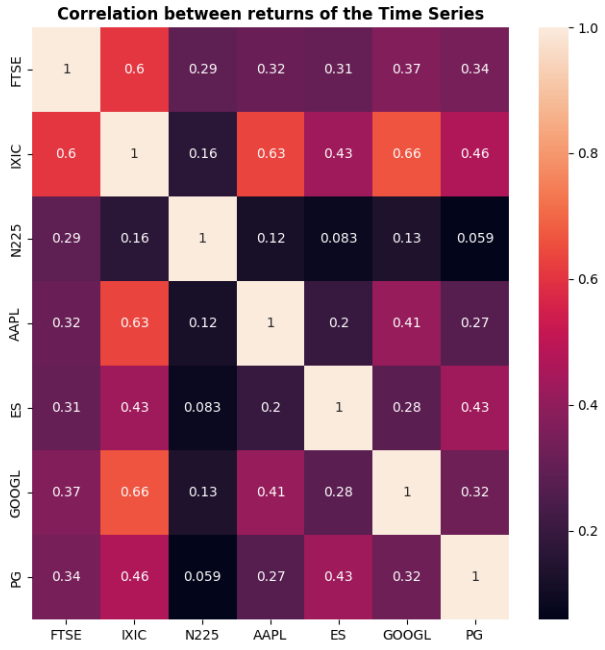


Fig. 4.1: Correlation between the returns of the selected time series

In Figure 4.1, we can see the correlations between the returns of the selected tickers: the correlations we tested have been found to be within an acceptable range. Using highly correlated time series might have led to redundant information in the model, making the dataset switch procedure we adopted while training the model less effective.. For the training and testing of our model, we selected the following tickers:

- Apple Inc., ticker: $AAPL;

- Eversource Energy, ticker: $ES;
- FTSE 100 Index, ticker: $FTSE;
- Google LLC, ticker: $GOOGL;
- Nasdaq Composite Index, ticker: $IXIC;
- Nikkei 225 Index, ticker: $N225;
- Procter & Gamble, ticker: $PG.

After downloading the data for the training and test period, we performed the feature engineering and data normalization process described in Section 3.2.2, and Section 3.2.3 respectively. It is important to point out that the Data Normalization was performed by calculating the mean and standard deviation of the series on the training set, and not on the whole data set, according to Equation 3.11, in order to avoid having a *look-ahead bias* in the model.

4.3 Model Hyperparameters

A hyperparameter can be described as a model parameter that cannot be estimated directly from the data, and for which we have no analytical formula available, Kuhn, Johnson (2018). There are some frameworks that are used to optimize the hyperparameters of a Machine Learning model, such as Random Search, Bergstra, Bengio (2012), Bayesian Optimization, Rasmussen, Williams (2006), and Tree-structured Parzen Estimators (TPEs), Franceschi et al. (2017). However, due to the computational burden of the optimization of hyperparameters, and the computational constraint imposed by the technology used, we were only able to optimize the following hyperparameters manually, and not in an extensive way. We will now describe the main Hyperparameters of the model, which can also be

found in Section 2.3.2. A summary can be found in Table **??** at the end of the section.

Activation Function

While Activation Functions are not hyperparameters, we decided to include them in this section as the thought process behind the decision of which one to use is similar to the one used for hyperparameters: in fact, there is no specific analytical formula that one can use to find which activation is most suited for a specific task, and there is no way to estimate it directly. We can however make some considerations: we considered that our data, after the normalization process, is distributed as a standard normal random variable so it can take both positive and negative values with equal probability, this leads us to tend more toward activation functions that presented an output range which is not limited to $\mathbb{R}^+$, such as the ReLU activation function. Furthermore, the distribution of our data presents also the property of being symmetric, we thus also would like this property in the activation function we use; symmetric activation functions are functions like the sigmoid and the hyperbolic tangent (tanh). After some initial experiments, we noticed that the Agents that made use of the tanh activation function overperformed the others in terms of cumulative reward during the test. It is worth pointing out, as we did in Section 4.1, that this research on optimal activation function was not carried out with any precise method other than a *trial and error* approach, thus there might be a better choice of the activation function.

Number of Games

The *number of games* we train the algorithm is a crucial hyperparameter of the model. In fact, since we do not dispose of any validation set, we risk overfitting the data if we train the model for too many games. On the other hand, if we train the model for a few games, we risk interrupting the learning process before

achieving a suitable policy parametrization. For these reasons, we tried different settings for the number of games, ranging from 300 to 1000.

Horizon

The *horizon* is what in Section 2.3.2 we denoted to as T, that is, the number of timesteps after which we perform an update to the policy. Clearly, the horizon should not be bigger than the Maximum Episode Length, and the common practice is to have a horizon that is a lot smaller than the Maximum Episode Length. It is important to bear in mind, however, that more frequent policy updates, that is a shorter horizon, should be treated with caution, as they can easily lead to overfitting of the model; for this reason, we usually adjust the horizon and the *learning rate* of the optimizer together. We should also consider the balance of the horizon with the discount factor γ, in order to properly define how far should future rewards influence the policy. We decided to test different settings, with the horizon ranging from 84 to 1280 timesteps.

Minibatch

The *minibatch* simply indicates the dimension of the batch on which we are going to perform SGD. In continuous action spaces, it is common to have a very large minibatch dimension, however, since the action space of our task is discrete, we can use way smaller minibatches. We decided to set the minibatch as a fraction of the Horizon ranging from 21 to 128 timesteps.

Epochs

The *epochs* parameter is what we denoted as K in Section 2.3.2. It is what regulates how many updates we perform once the horizon is reached. Clearly, a higher number should also be related to a greater minibatch dimension, and should be treated with caution as it might lead to more unstable learning and overfitting. In

our setting, given the discrete action space and the horizon choice, we decided to have the epochs ranging from 2 to 4 per update.

Clipping Range

The *clipping range*, denoted as ϵ, is the parameter that controls the maximum and minimum size of the policy updates. A large ϵ can lead to faster learning but also to more unstable learning. In fact, if we set the parameter too large, we might have a destructive update to the policy that leads the performance to collapse. For this parameter, we followed the best practice to set the value of ϵ close to the ones of Schulman et al. (2017b), that is, $\epsilon = 0.2$.

Discount Factor

The *discount factor*, denoted as γ, is the parameter that regulates how far future rewards influence the policy. In some settings, it might be useful to set the gamma to a predefined value if, for example, we know it already from the problem statement; in our case, however, we did not have such a situation, thus we decided to set the value of the parameter to the standard one, that is 0.99.

GAE Lambda

The *GAE lambda* parameter, denoted simply λ, is the parameter that governs the exponential weighted average used in Equation 2.19. Practically speaking, this parameter governs how much the Agent is making use of the current value estimate when updating its value estimate: lower values lead to higher bias, weighting more the current value estimate; on the other hand high values of λ lead to a higher variance of the updates, weighting more the rewards obtained in the environment. We decided to experiment with a range that goes from 0.9 to 0.97.

Value Function Coefficient

The Value Function coefficient is what we denoted as c_1 in Equation 2.24, and regulates how much the Value Loss contributes to the total loss. We set this

parameter to 0.5 as in the original paper.

Entropy Coefficient

The *Entropy Coefficient*, denoted as c_2 in Equation 2.24, is what regulates the entropy bonus, which encourages exploration. An extreme value of this parameter again might lead to excessive exploration and slower learning speed. Note that in cases in which exploration of the Environment is not necessary if not detrimental, we set this parameter to 0. In our case, this ranges from 0.01 to 0.001.

Parameter	Value
Number of Games	300-1000
Activation Function	Tanh
Horizon	84-1280
Minibatch	21-128
Epochs	2-4
Clipping Range	0.2
Discount Factor	0.99
GAE Lambda	0.9-0.97
Value Function Coefficient	0.5
Entropy Coefficient	0.001-0.1

Table 4.2: Main Hyperparameters of the model and their values

4.4 Model Training

The training of the model was carried out using the Environment described in section 3.1.2, switching between data sets every 5 iterations. The number of iterations between each switch was chosen in order to avoid overfitting and excessive specialization of the policy to a single asset's dynamics. During training we kept track of the rewards per episode, the entropy of the distribution, and the total loss. After a trial and error process, we managed to achieve a configuration of hyperparameters that allowed us to obtain satisfactory results on our measures. The reward formulation we opted for is the monthly Sharpe ratio, so at each time

step the agent receives as reward the Sharpe ratio calculated on the previous 21 time steps.

The main challenges that we faced during training were due to the wide search space of the hyperparameters: in fact, as we mentioned in Section 4.3, one would need a proper optimization tool in order to find the best configuration. After a process of trial and error, we managed to obtain satisfactory results with the following configuration:

Parameter	**Value**
Number of Games	500
Activation Function	Tanh
Horizon	1280
Minibatch	64
Epochs	4
Clipping Range	0.1
Discount Factor	0.99
GAE Lambda	0.97
Value Function Coefficient	0.5
Entropy Coefficient	0.001

Table 4.3: Hyperparameters configuration of the model

As we can see in Figure 4.2, the reward formulation we adopted allowed the Agent to rapidly learn the dynamics of the environment, obtaining satisfactory results already around the game 200.

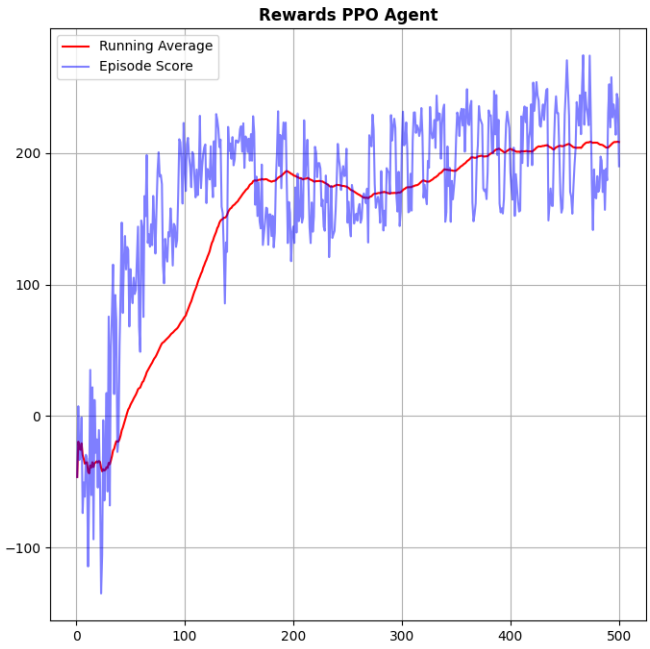


Fig. 4.2: rewards obtained on the training set

4.5 Results on the Test

In this section, we will present the most significant results obtained in the testing phase of the models, using the hyperparameters configuration described in Table 4.3. As we anticipated in Section 4.2, we will test the Agent in three consecutive periods, that is, the year 2018, the year 2019, and the year 2020. In order to evaluate the models, we used the metrics described in Section 3.3. Note that we are going to include in this section only the most significant results that help us showcase the policy of the Agent; one can find all the results in Appendix A.

	$GOOGL, 2018	
Statistic	Agent	Benchmark
Volatility (%)	24.3	29.11
Return (%)	8.73	-11.61
Sharpe Ratio	0.5	-0.32
Maximum Drawdown (%)	18.73	23.4
Calmar Ratio	2.17	-2.08
Semi Volatility (%)	21.26	21.53
Sortino Ratio	0.57	-0.43
Value at Risk (%)	-2.6	-3.1
Conditional Value at Risk (%)	-3.79	-4.5

Table 4.4: Summary statistics for the year 2018 on $GOOGL

As we can see in Table 4.4, the agent managed to overperform the Benchmark in all performance measures, in fact, in this scenario, the Agent managed to generate a positive return even if the benchmark generated a negative one; the impressive fact is that the agent managed to generate such a higher return while attaining also a lower volatility, and a lower value in all risk metrics.

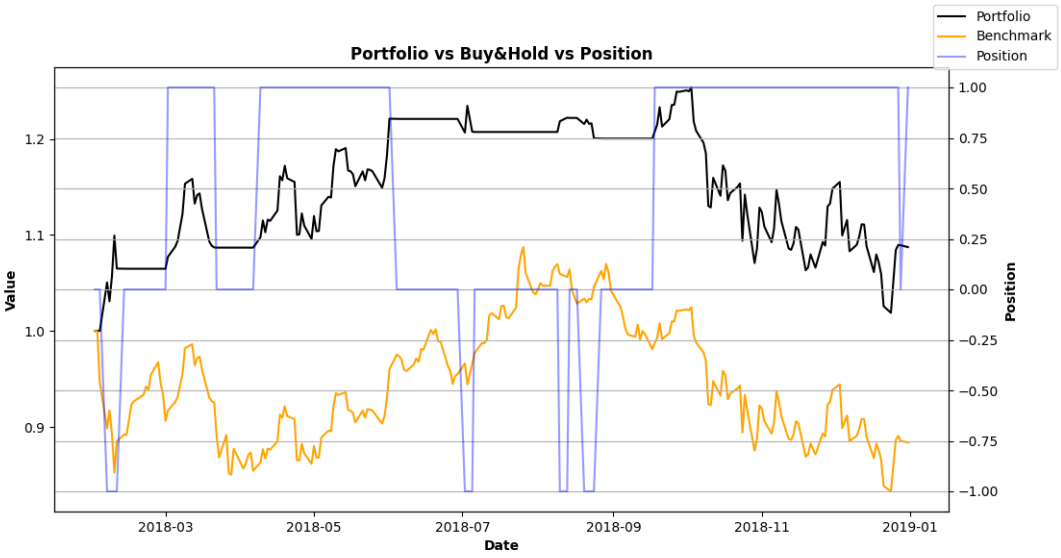
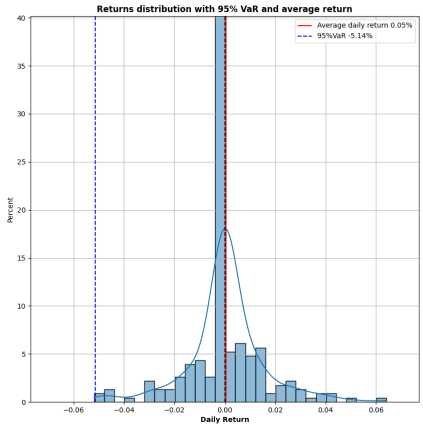
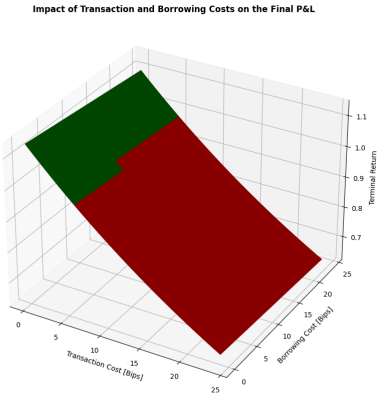


Fig. 4.3: Agent performance vs Benchmark, $GOOGL, 2018

As a matter of fact, as we can see in Figure 4.3, in this case, the Agent managed to interpret extremely well the data coming in from the Environment, in fact, not only it managed to not lose in periods of higher volatility, but it also managed to correctly short sale the asset in some occasions. This last fact is of extreme importance: in fact, one of the main issues we faced in the implementation of the model was the fact that, with many of the hyperparameters configurations tested, the Agent learned a buy and hold strategy in many of the cases, a *flat-only* strategy in some of the cases, and in a very little minority it actually managed to learn to switch position correctly through time. We thus consider this result extremely satisfactory, as even though the Agent might not generate incredibly high rewards, it succeeded in developing an actual strategy, which proved to be over-performing the benchmark in terms of risk metrics, even when it underperformed in terms of raw returns.



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. 4.4: $GOOGL, 2018

In Figure 4.4a and 4.4b, we can see the returns distribution of the Agent and the impact of transaction and borrowing costs on the terminal return of the Agent. In

order to properly present the Agent's results, we will now proceed to show another informative result in the following table.

	$IXIC, 2018	
Statistic	Agent	Benchmark
Volatility (%)	11.59	21.56
Return (%)	-5.5	-10.47
Sharpe Ratio	-0.26	-0.45
Maximum Drawdown (%)	21.72	23.64
Calmar Ratio	-2.83	-3.24
Semi Volatility (%)	15.67	16.66
Sortino Ratio	-0.3	-0.59
Value at Risk (%)	-1.99	-2.43
Conditional Value at Risk (%)	-2.76	-3.33

Table 4.5: Summary statistics for the year 2018 on $IXIC

As we can see above, the Agent did not manage to generate any return this year, but rather ended up with a negative return of -5.5%; however, we must point out that even when practically losing money, the Agent still manages to overperform the buy and hold benchmark: it does so in every statistic, from lower risk statistics to higher risk-adjusted returns.

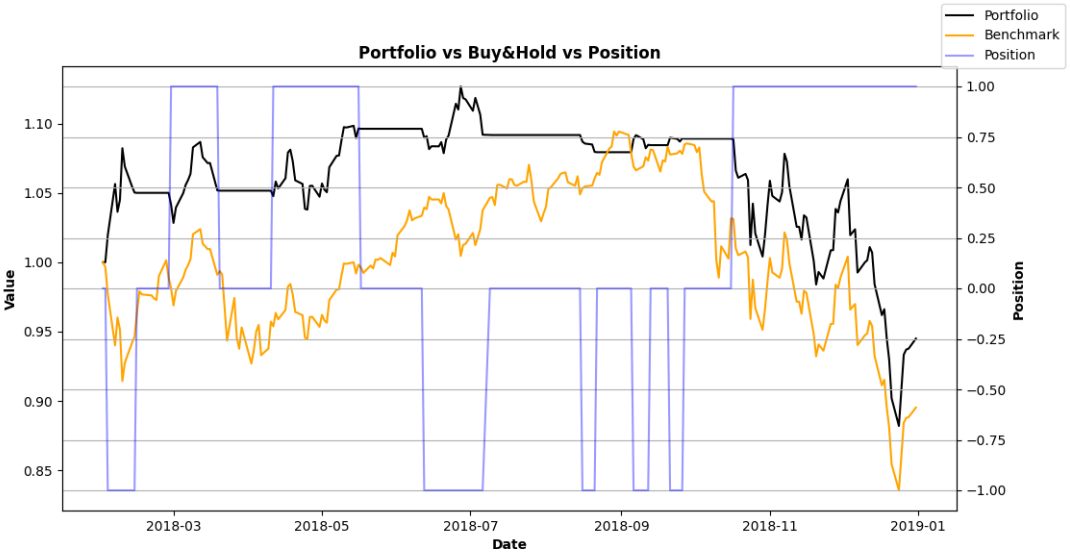
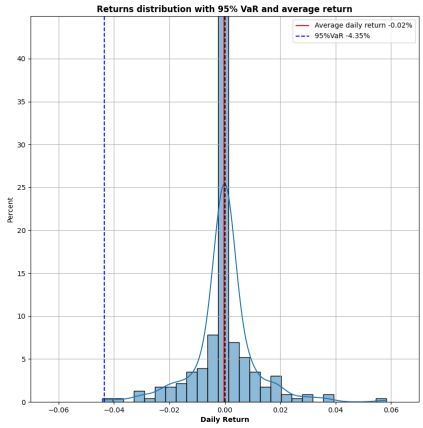
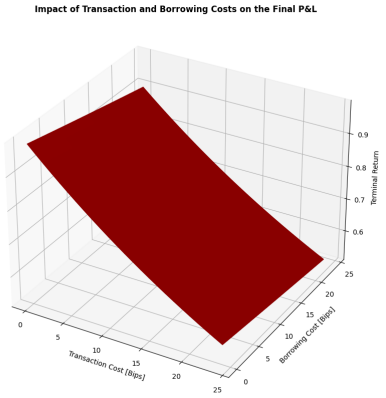


Fig. 4.5: Agent performance vs Benchmark, $IXIC, 2018

In Figure 4.5, we can see that the agent managed to do extremely well in the first half of the year, reaching 10% return already from June 2018. After that point, the agent already has a high Sharpe ratio for the previous month. in fact, we can see graphically that the portfolio has very low volatility and high returns. This condition might have led the agent to consider that any action was likely to receive a lower reward than the one obtained by going flat for the next days and keep the low volatility and high return of the past month. We can see in fact that as we get closer to the month, and thus the window on which the Sharpe ratio moves away from the period of high returns and low volatility, the Agent starts to trade again to increase its reward. This behavior can be seen also throughout the other tests: this is mainly accountable to the fact that the Sharpe Ratio weighs equally the returns on all preceding time steps, thus high returns influence heavily the Agent's strategy for the next month, as opposed to low returns that make the Agent more prone to trade frequently to increase its reward.



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. 4.6: $IXIC, 2018

Chapter 5

Conclusions and Further Developments

5.1 Conclusions

Let us now draw some conclusions on the work done. In this work, we have embarked on a journey through the intricate landscape of reinforcement learning and its application in developing trading strategies. Our exploration has been organized into distinct sections, each contributing to a holistic understanding of the subject.

The technical foundation, presented in Section 1, has provided us with a solid grasp of Markov decision processes (MDPs), dynamic programming, and the core concepts of prediction and control through reinforcement learning. Through meticulous examination, we have established the theoretical groundwork necessary to build and analyze our trading strategies.

Section 2 introduced approximate solution methods, a crucial bridge between theory and practical application. With a focus on function approximation and policy gradient methods, we have unlocked the potential of reinforcement learning;

furthermore, we presented stochastic gradient methods, linear function approximation, and the burgeoning power of neural networks, which equipped us with a versatile toolbox to tackle real-world trading challenges. Actor-critic methods, particularly the Generalized Advantage Estimation and Proximal Policy Optimization algorithms, have further enriched our arsenal of techniques for constructing and refining trading policies.

In Section 3, we presented our methodology: we introduced the core components of our model, including the agent, environment, data loader, and network architecture, which collectively constitute the backbone of our trading system. Detailed insights into data processing, encompassing tickers, features, and data normalization, have ensured that our model is well-prepared to handle the complexities of financial data. Additionally, we have elucidated our approach to performance measurement, a critical facet in evaluating the effectiveness and robustness of our trading strategies.

As we turn our attention to Section 4, we encounter the culmination of our efforts, that is, the results. Preliminary remarks have set the stage for a comprehensive data summary, allowing us to comprehend the intricacies of the financial markets under consideration. Hyperparameters, a pivotal aspect of our work, have been meticulously tuned to achieve optimal performance. We then unveiled the test results: through empirical evidence and careful analysis, we evaluated the effectiveness of our reinforcement learning-based trading strategies in the real-world context. These results showcase the practicality and potential of our work, reinforcing the notion that the integration of cutting-edge reinforcement learning techniques can enhance trading performance. In fact, the Agent managed to overperform the benchmark in most of the risk metrics we have proposed. It must be noted however that our model did have one main issue, that is the reward formulation: in fact, as we have seen in Section 4.5, the monthly Sharpe Ratio (SR) weighs

equally all past rewards in the previous month. In fact, the SR is constructed as the ratio between a simple arithmetic average and a standard deviation, which leads the Agent to be influenced by returns he obtained in the past month. This leads to a behavior that can be seen throughout the majority of the tests: after a period of good trades, the Agent tends to reduce the number of trades to the point that it goes flat most of the time, and on the other hand, after a period of bad trades, the Agent tends to increase the number of trades. We can nonetheless be satisfied with the results of our work: in fact, we managed to develop from scratch an implementation of the PPO algorithm and to train the Agent effectively, so as to obtain good results in the test set. One final remark is that we should consider the results in the later years of the test with more caution: in fact, as we move on with time, e.g. in the 2020 test, we are actually using a model that is two years old, while in practice we would probably have at least trained the model with more recent data, if not changed some configurations in the model.

To conclude, as the financial world continues to evolve, this work offers a glimpse into the exciting possibilities that lie ahead for those who seek to harness the potential of reinforcement learning in the pursuit of trading excellence.

5.2 Further Developments

In this final section, we are going to provide some ideas to extend the work done in this thesis. The first suggestion is to experiment with other choices of function approximation: in fact, in this work, we used a Fully Connected Artificial Neural Network, but the landscape of Machine Learning and Deep Learning offers many more algorithms to carry out function approximation. The second suggestion is again related to function approximation: in fact, in this work, we did not use any sequential data; it would definitely be interesting to see how the Agent's performance changes with the use of architectures such as Convolutional Neural

Networks, Recurrent Neural Networks, or even more state-of-the-art architectures such as Transformers. The last suggestion is to dedicate time and research to the reward: in fact, this is the only way in which we can effectively communicate with the Agent, giving him a direction to follow when trading; one improvement could be to adopt an exponentially weighted formulation of the Sharpe Ratio, so to have a 'smoother' weight decay of the past observations.

Appendix A

Results

A.1 $AAPL

	$AAPL					
	Agent			Benchmark		
	2018	2019	2020	2018	2019	2020
Volatility (%)	23.97	20.82	27.8	29.66	26.23	46.71
Return (%)	-7.99	13.5	10.97	-5.79	85.95	76.71
Sharpe Ratio	-0.26	0.72	0.51	-0.07	2.51	1.45
Maximum Drawdown (%)	37.6	16.89	11.3	36.73	18.16	31.43
Calmar Ratio	-2.19	1.92	3.62	-1.19	12.2	10.78
Semi Volatility (%)	22.25	21.14	21.62	21.03	21.46	34.36
Sortino Ratio	-0.28	0.7	0.66	-0.1	3.07	1.98
Value at Risk	-2.83	-1.65	-2.59	-3.16	-2.07	-4.49
Conditional Value at Risk	-4.03	-2.86	-3.53	-4.33	-3.68	-6.76

Table A.1: Summary statistics for the test years on $AAPL

Year 2018

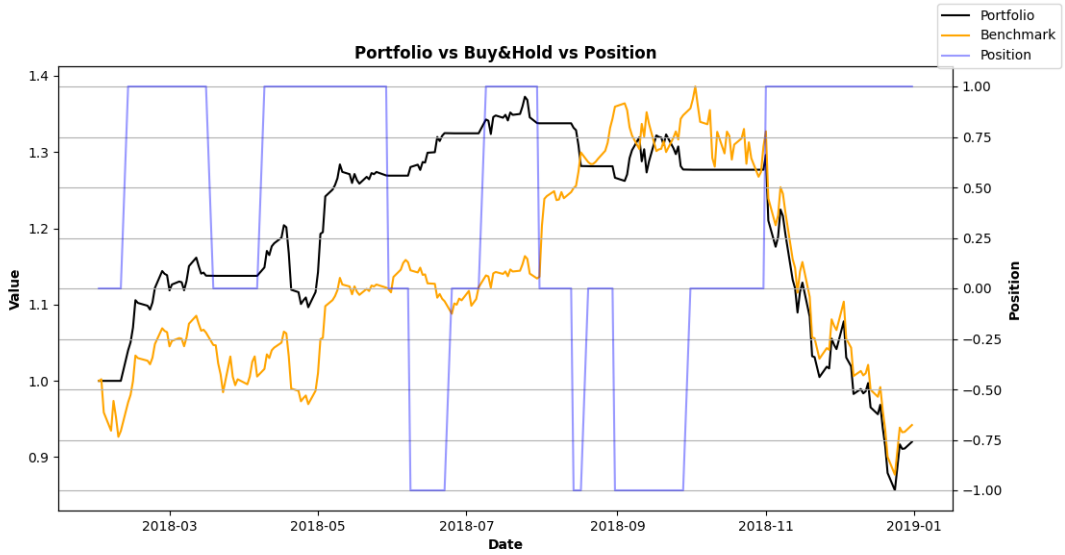
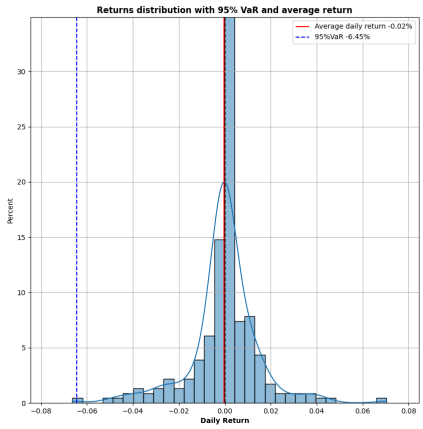
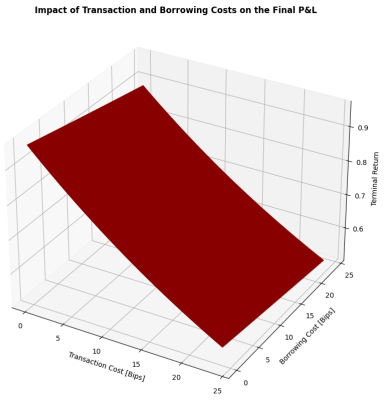


Fig. A.1: Agent performance vs Benchmark, $AAPL, 2018



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.2: $AAPL, 2018

Year 2019

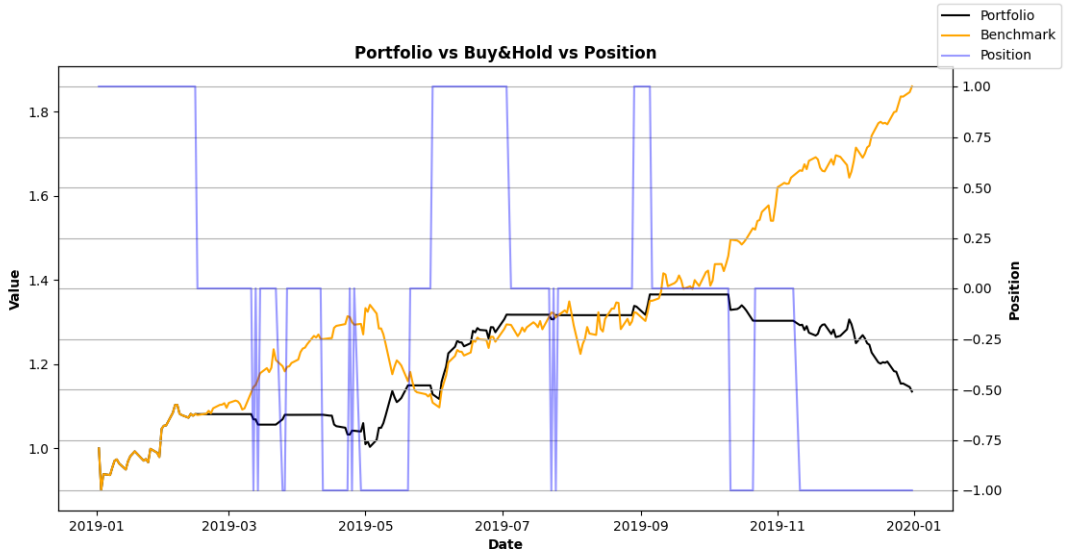
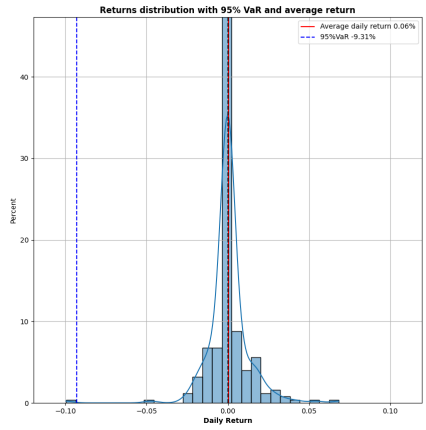
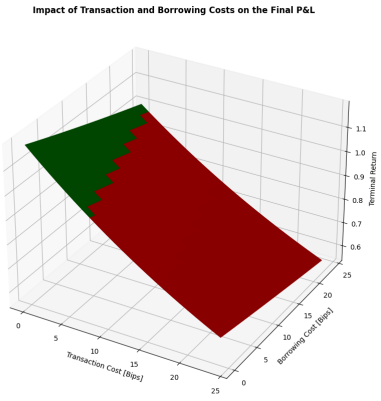


Fig. A.3: Agent performance vs Benchmark, $AAPL, 2019



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.4: $AAPL, 2019

Year 2020

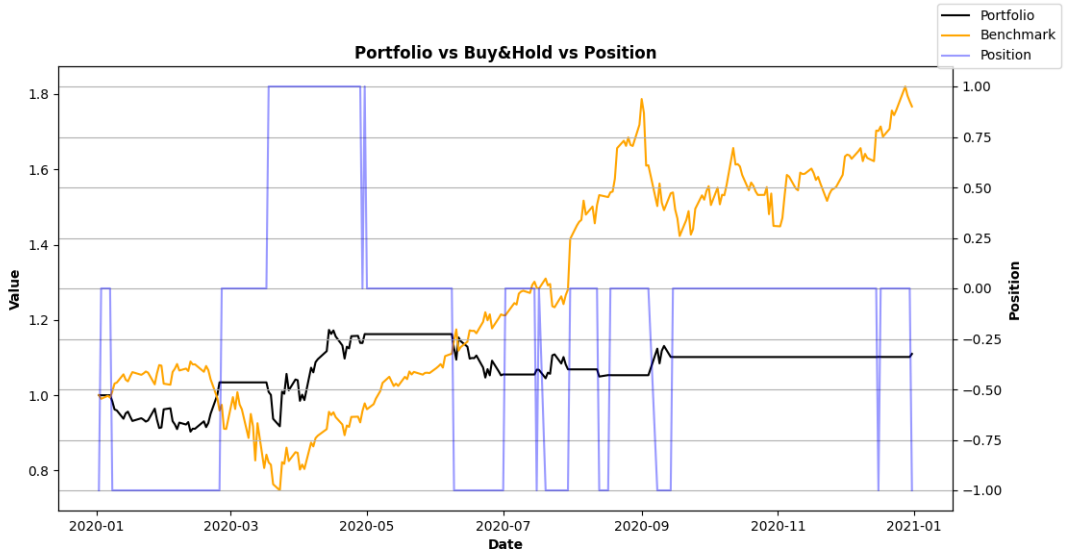
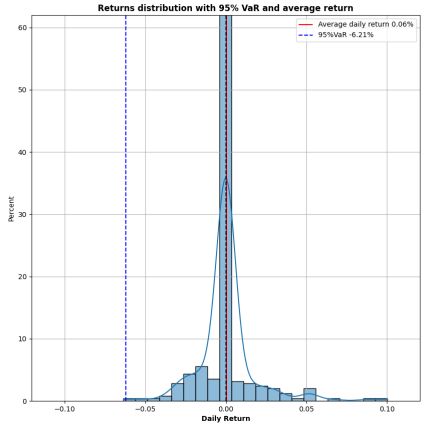
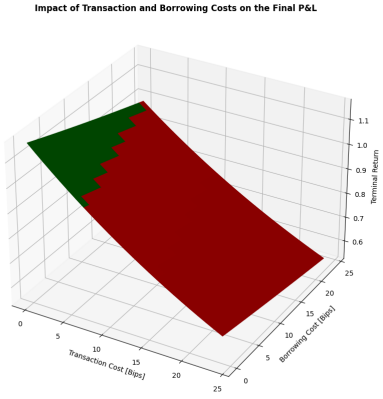


Fig. A.5: Agent performance vs Benchmark, $AAPL, 2020



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.6: $AAPL, 2020

A.2 $ES

	$ES					
	Agent			Benchmark		
	2018	2019	2020	2018	2019	2020
Volatility (%)	15.91	10.66	24.99	17.84	12.86	45.26
Return (%)	7.41	3.18	18.34	3.09	33.99	3.94
Sharpe Ratio	0.57	0.35	0.8	0.28	2.35	0.31
Maximum Drawdown (%)	12.59	11.68	16.21	16.2	8.13	36.85
Calmar Ratio	1.77	2.17	2.91	0.74	14.27	0.4
Semi Volatility (%)	13.03	6.98	25.11	14.13	8.07	34.24
Sortino Ratio	0.7	0.53	0.79	0.35	3.75	0.41
Value at Risk (%)	-1.62	-1.17	-1.68	-1.83	-1.15	-3.71
Conditional Value at Risk (%)	-2.33	-1.37	-3.83	-2.85	-1.64	-6.7

Table A.2: Summary statistics for the test years on $ES

Year 2018

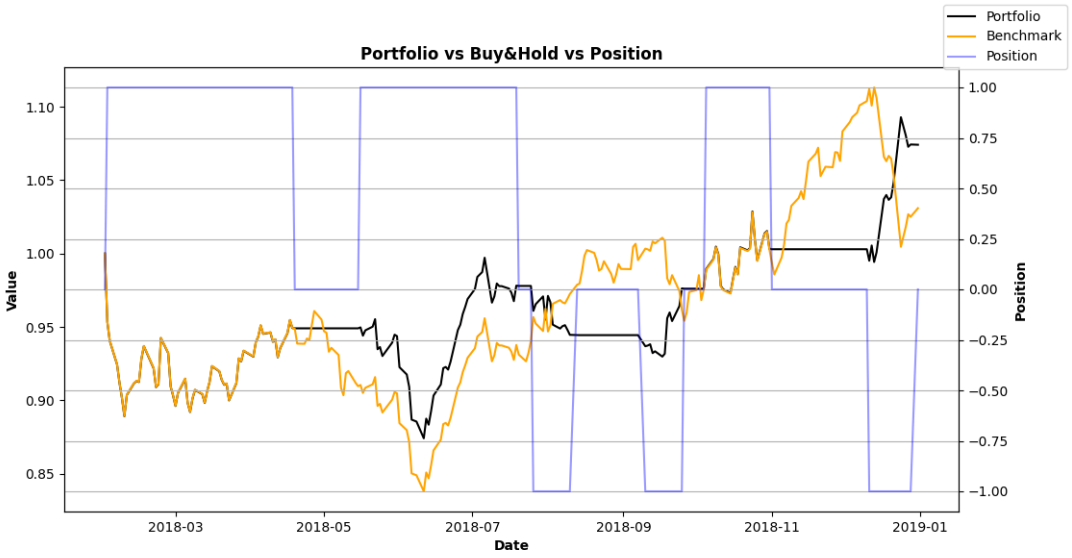
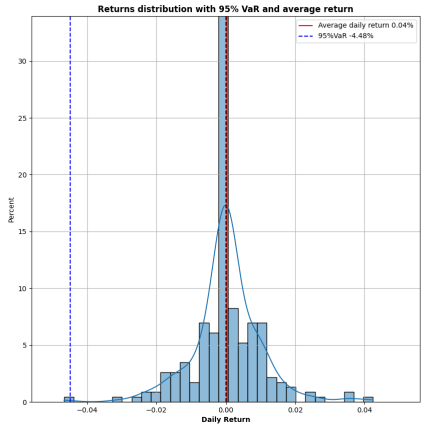
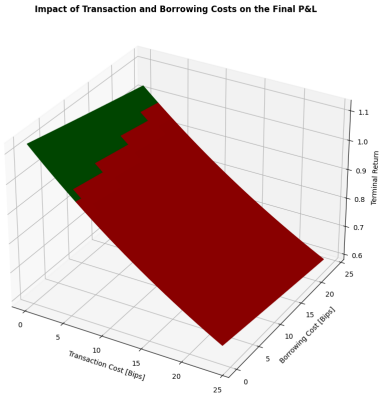


Fig. A.7: Agent performance vs Benchmark, $ES, 2018



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.8: $ES, 2018

Year 2019

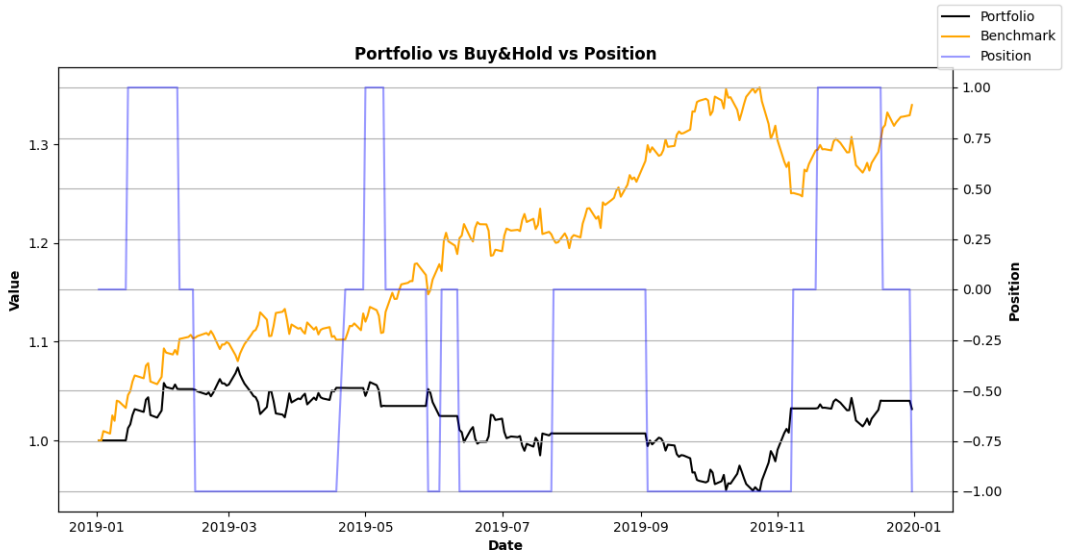
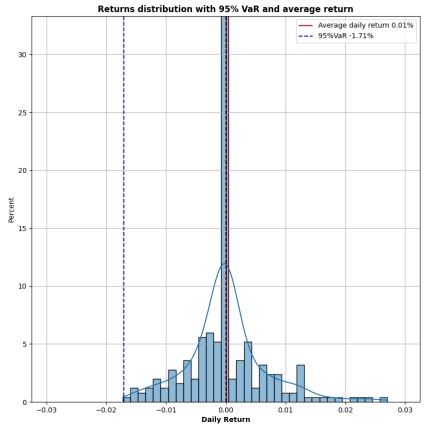
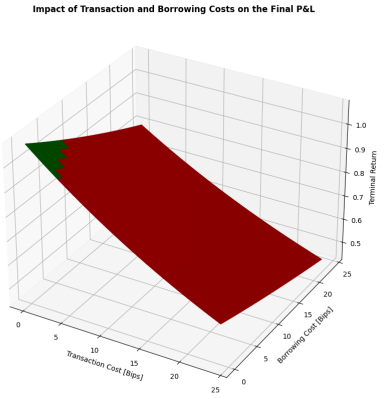


Fig. A.9: Agent performance vs Benchmark, $ES, 2019



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.10: $ES, 2019

Year 2020

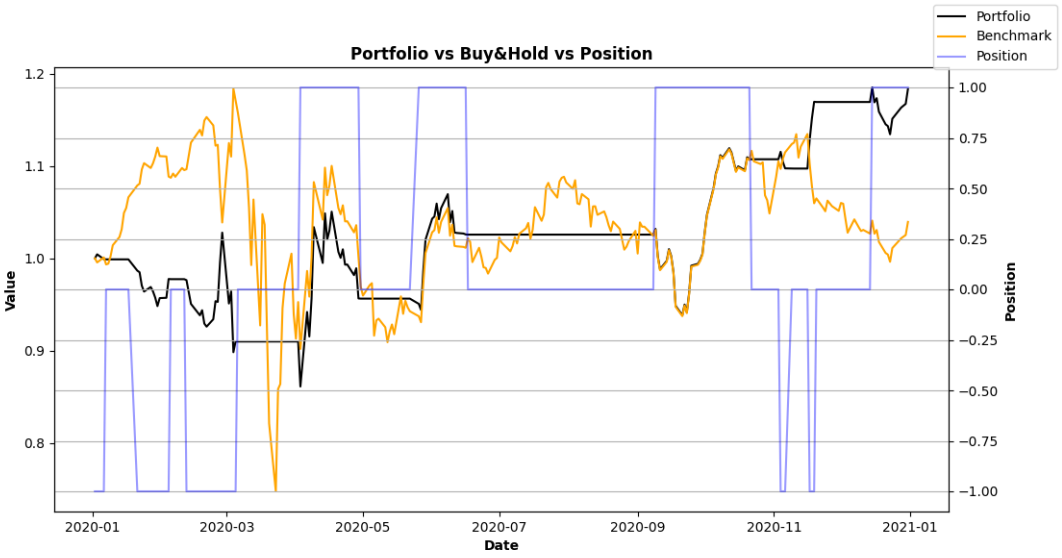
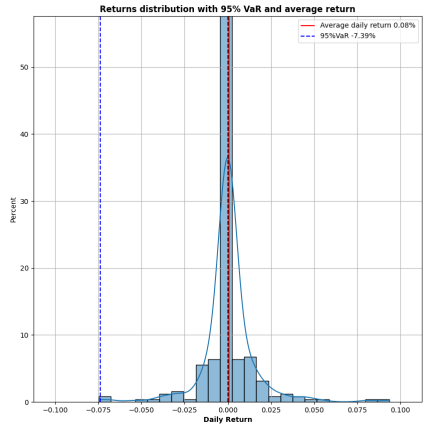
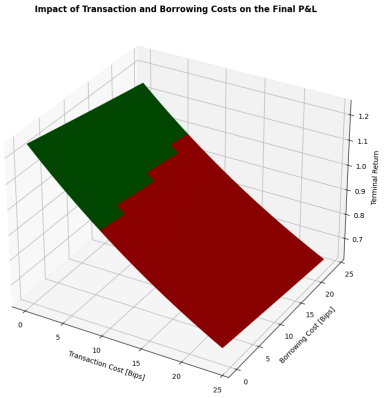


Fig. A.11: Agent performance vs Benchmark, $ES, 2020



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.12: $ES, 2020

A.3 $FTSE

	$FTSE					
	Agent			Benchmark		
	2018	2019	2020	2018	2019	2020
Volatility (%)	11.65	9.47	29.17	13.54	11.88	29.54
Return (%)	-2.24	11.74	-10.8	-10.69	12.0	-15.04
Sharpe Ratio	-0.16	1.24	-0.25	-0.86	1.03	-0.41
Maximum Drawdown (%)	13.55	6.87	33.72	16.41	8.06	34.93
Calmar Ratio	-1.42	9.06	-1.62	-4.59	4.74	-2.88
Semi Volatility (%)	9.94	8.22	25.74	10.26	8.4	24.57
Sortino Ratio	-0.19	1.43	-0.29	-1.14	1.46	-0.5
Value at Risk (%)	-1.21	-0.91	-3.34	-1.37	-1.09	-3.34
Conditional Value at Risk (%)	-1.68	-1.39	-4.68	-1.96	-1.68	-4.68

Table A.3: Summary statistic for the test years on $FTSE

Year 2018

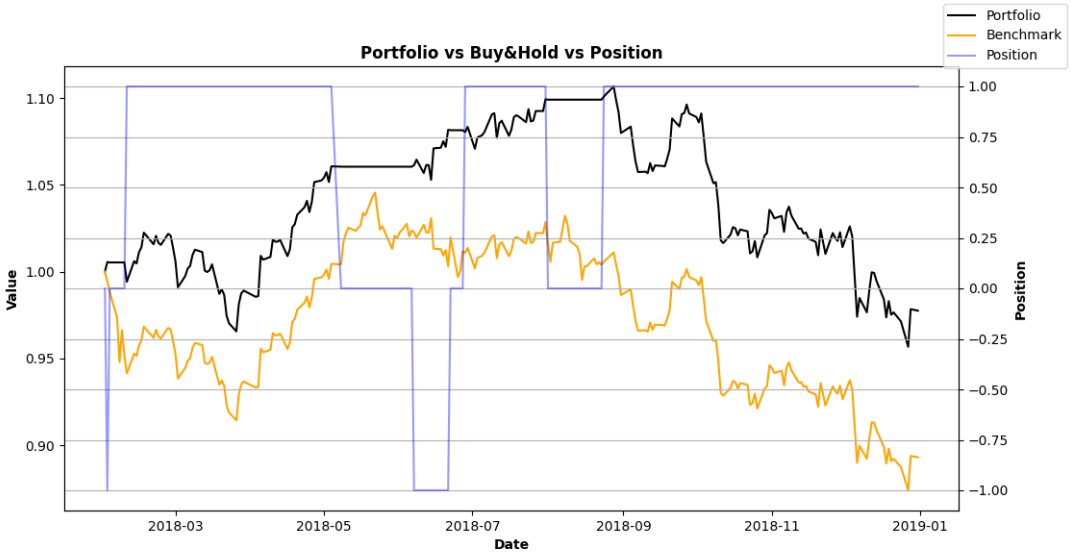
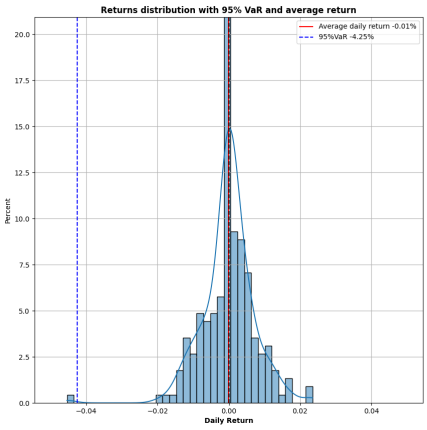
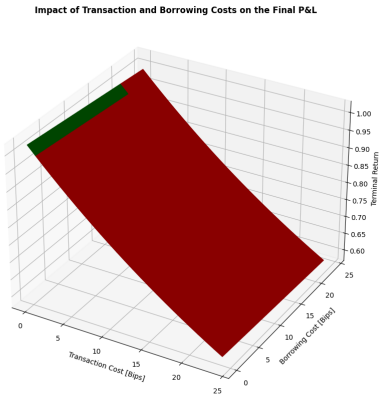


Fig. A.13: Agent performance vs Benchmark, $FTSE, 2018



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.14: $FTSE, 2018

Year 2019

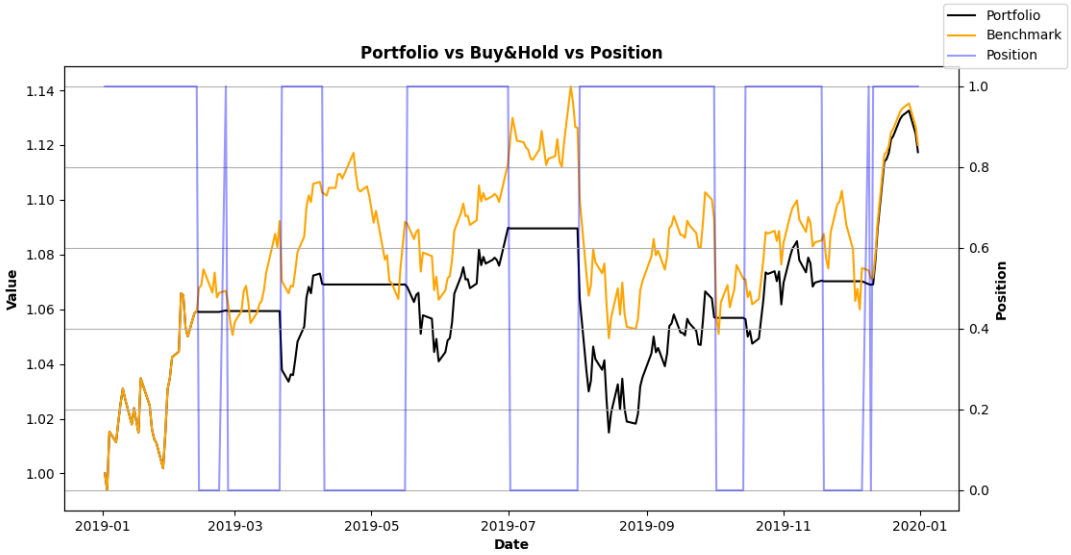
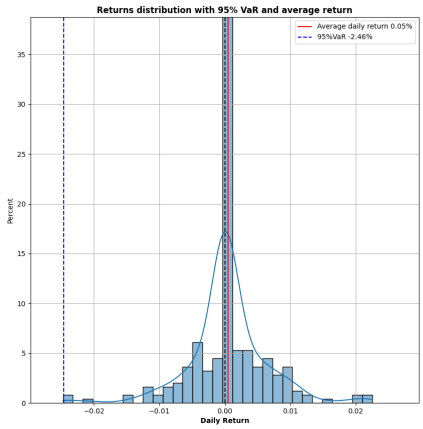
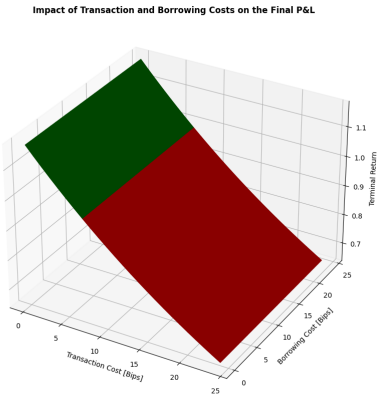


Fig. A.15: Agent performance vs Benchmark, $FTSE, 2019



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.16: $FTSE, 2019

Year 2020

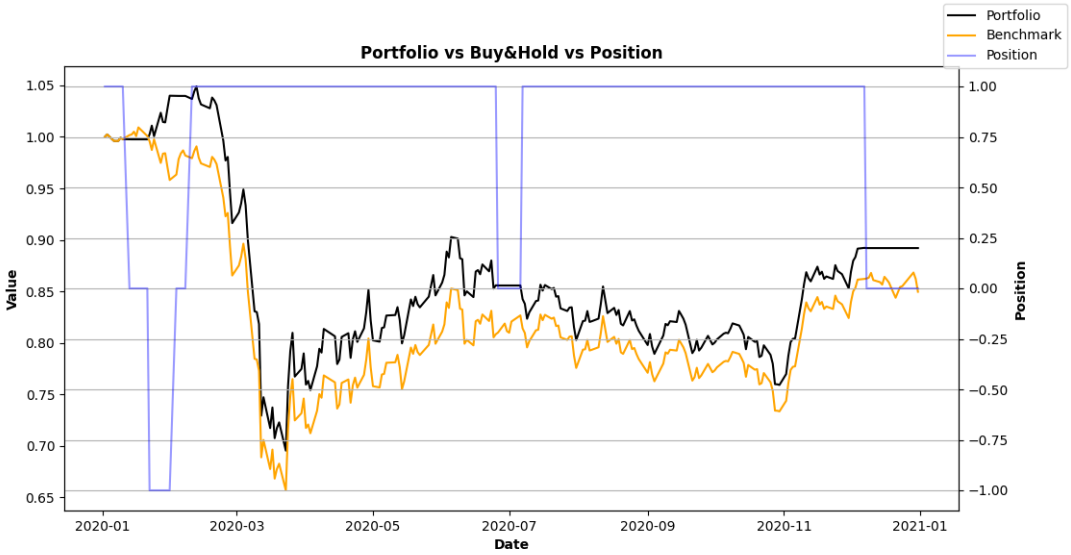
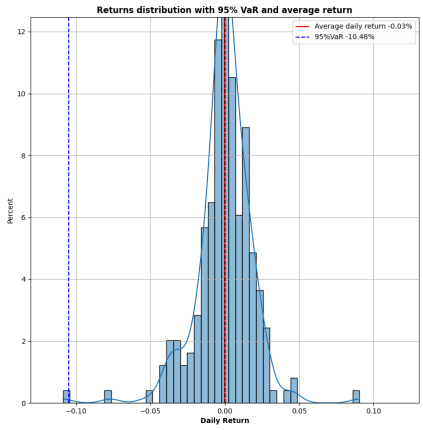
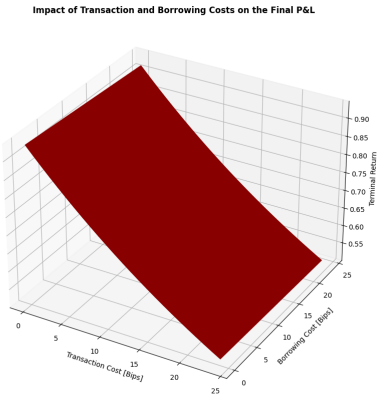


Fig. A.17: Agent performance vs Benchmark, $FTSE, 2020



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.18: $FTSE, 2020

A.4 $GOOGL

	$GOOGL					
	Agent			Benchmark		
	2018	2019	2020	2018	2019	2020
Volatility (%)	24.3	14.18	23.85	29.11	23.73	38.48
Return (%)	8.73	-3.06	32.74	-11.61	26.99	28.05
Sharpe Ratio	0.5	-0.15	1.3	-0.32	1.13	0.84
Maximum Drawdown (%)	18.73	14.36	10.88	23.4	19.86	30.87
Calmar Ratio	2.17	-1.56	9.02	-2.08	4.93	4.13
Semi Volatility (%)	21.26	16.33	19.98	21.53	17.42	30.18
Sortino Ratio	0.57	-0.13	1.56	-0.43	1.54	1.07
Value at Risk (%)	-2.6	-1.38	-2.11	-3.1	-2.17	-4.08
Conditional Value at Risk (%)	-3.79	-2.39	-3.01	-4.5	-3.31	-5.82

Table A.4: Summary statistics for the test years on $GOOGL

Year 2018

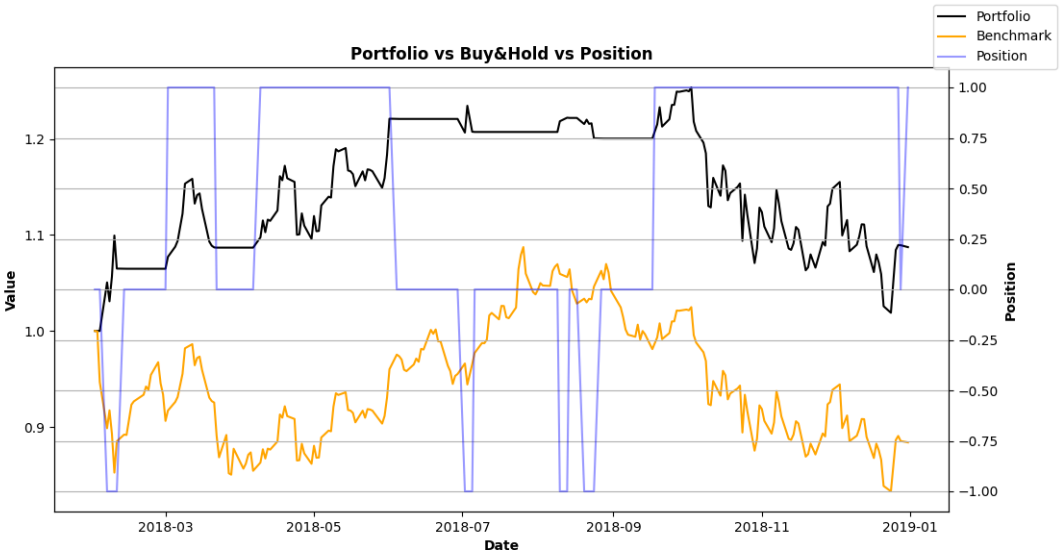
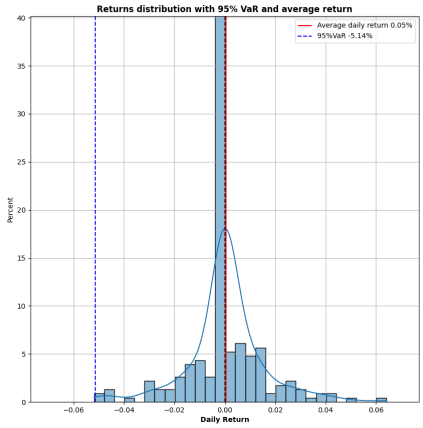
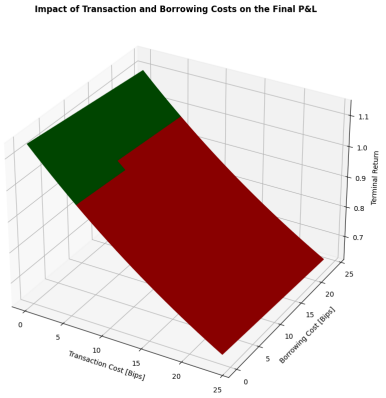


Fig. A.19: Agent performance vs Benchmark, $GOOGL, 2018



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.20: $GOOGL, 2018

Year 2019

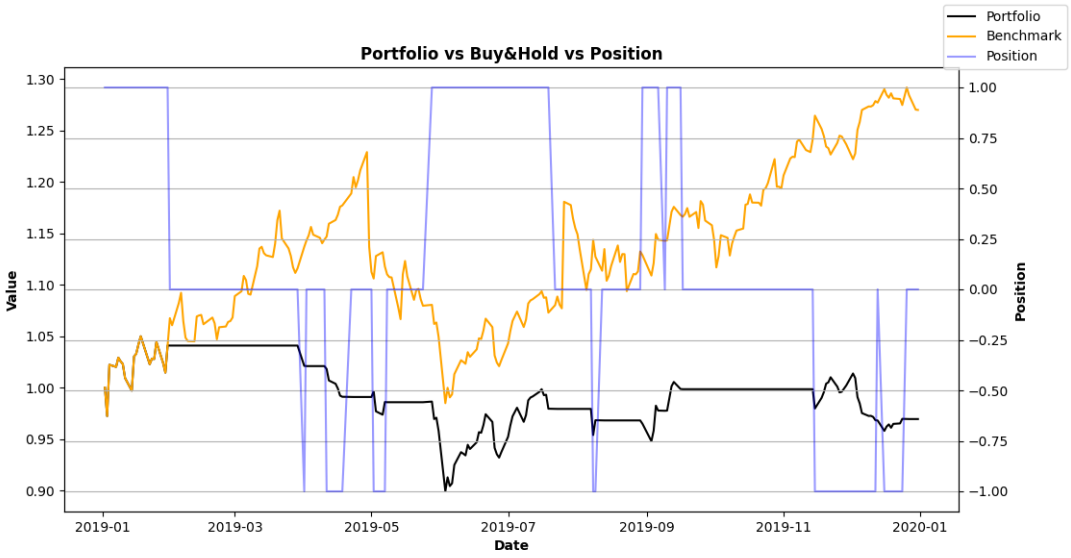
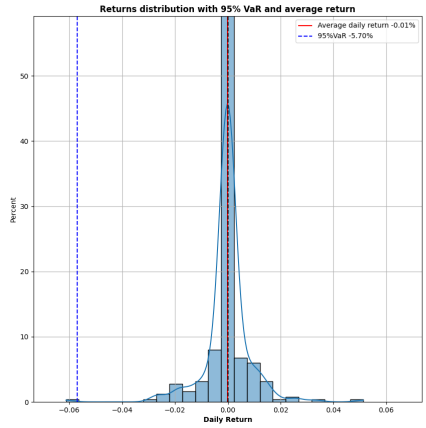
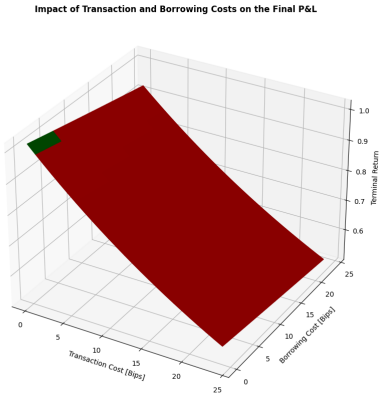


Fig. A.21: Agent performance vs Benchmark, $GOOGL, 2019



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.22: $GOOGL, 2019

Year 2020

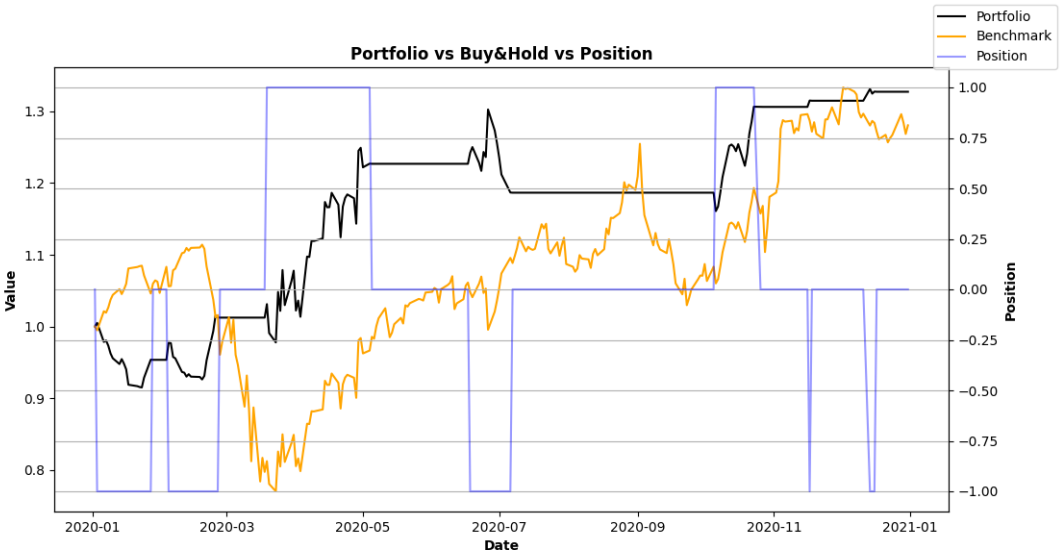
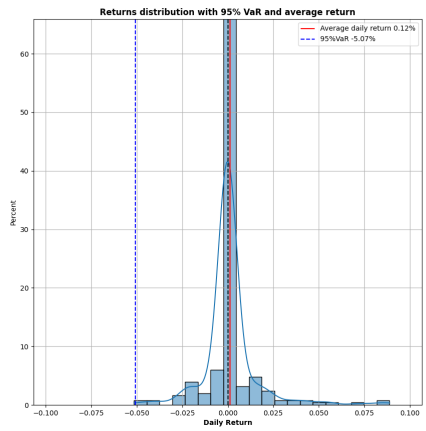
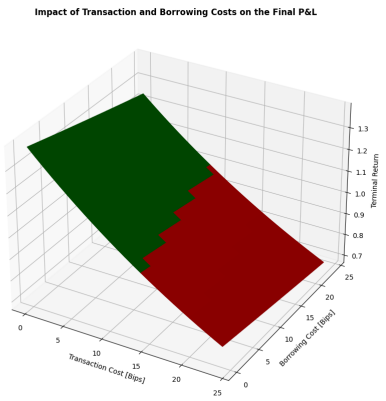


Fig. A.23: Agent performance vs Benchmark, $GOOGL, 2020



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.24: $GOOGL, 2020

A.5 $IXIC

	$IXIC					
	Agent			Benchmark		
	2018	2019	2020	2018	2019	2020
Volatility (%)	17.59	13.62	21.88	21.56	15.69	35.55
Return (%)	-5.5	25.31	15.39	-10.47	34.6	41.75
Sharpe Ratio	-0.26	1.73	0.76	-0.45	1.98	1.16
Maximum Drawdown (%)	21.72	6.12	15.57	23.64	10.18	30.12
Calmar Ratio	-2.83	11.79	5.61	-3.24	12.62	6.13
Semi Volatility (%)	15.67	9.76	14.97	16.66	11.98	31.69
Sortino Ratio	-0.3	2.42	1.11	-0.59	2.59	1.3
Value at Risk (%)	-1.99	-1.16	-1.54	-2.43	-1.51	-3.72
Conditional Value at Risk (%)	-2.76	-1.8	-2.62	-3.33	-2.33	-5.62

Table A.5: Summary statistics for the test years on $IXIC

Year 2018

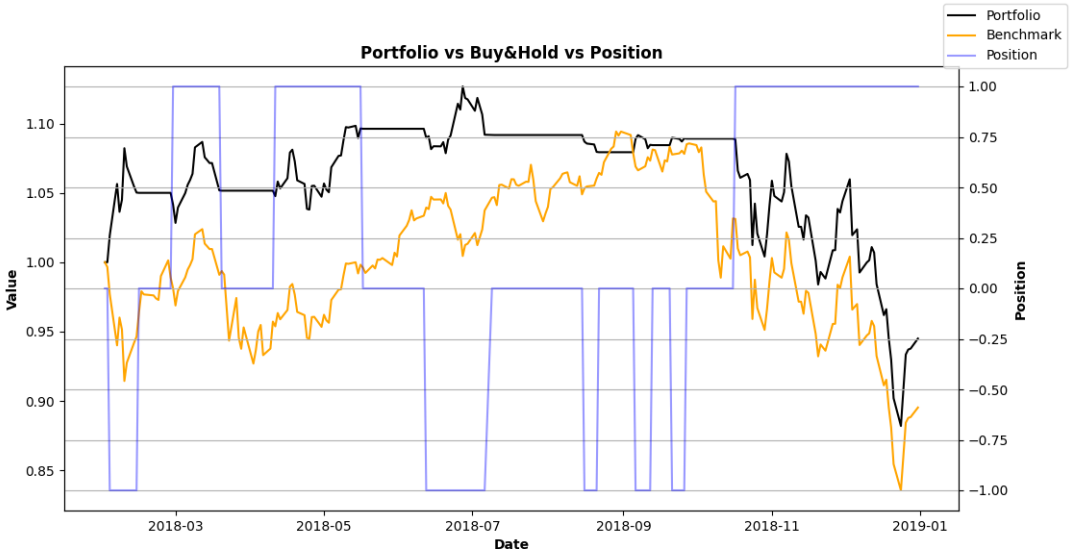
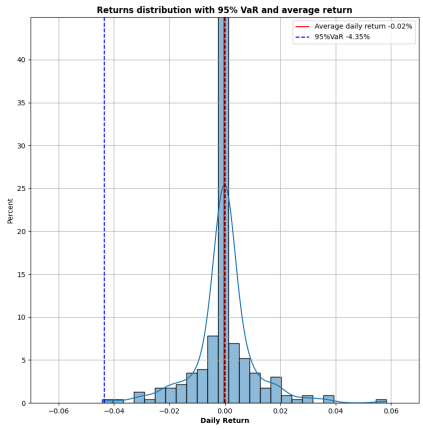
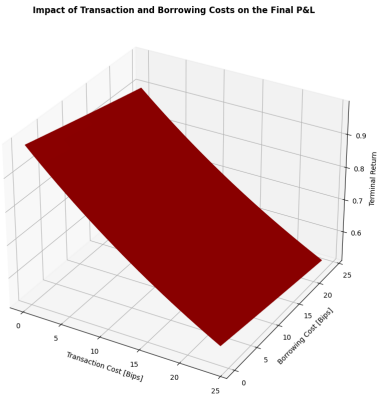


Fig. A.25: Agent performance vs Benchmark, $IXIC, 2018



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.26: $IXIC, 2018

Year 2019

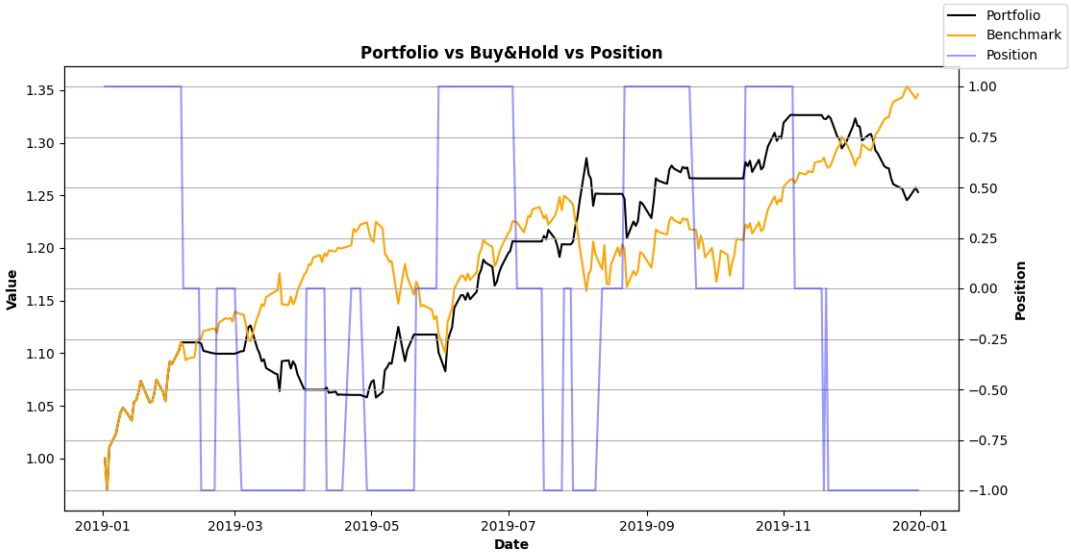
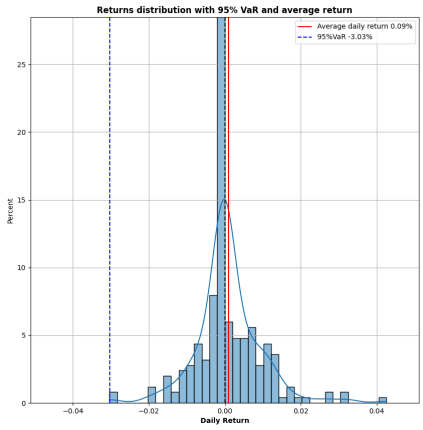
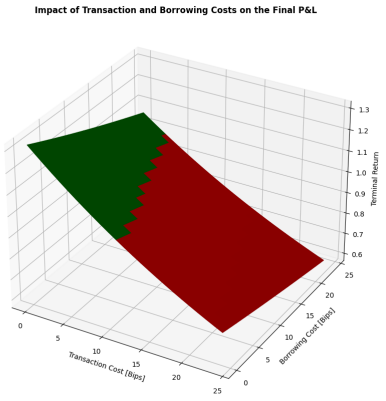


Fig. A.27: Agent performance vs Benchmark, $IXIC, 2019



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.28: $IXIC, 2019

Year 2020

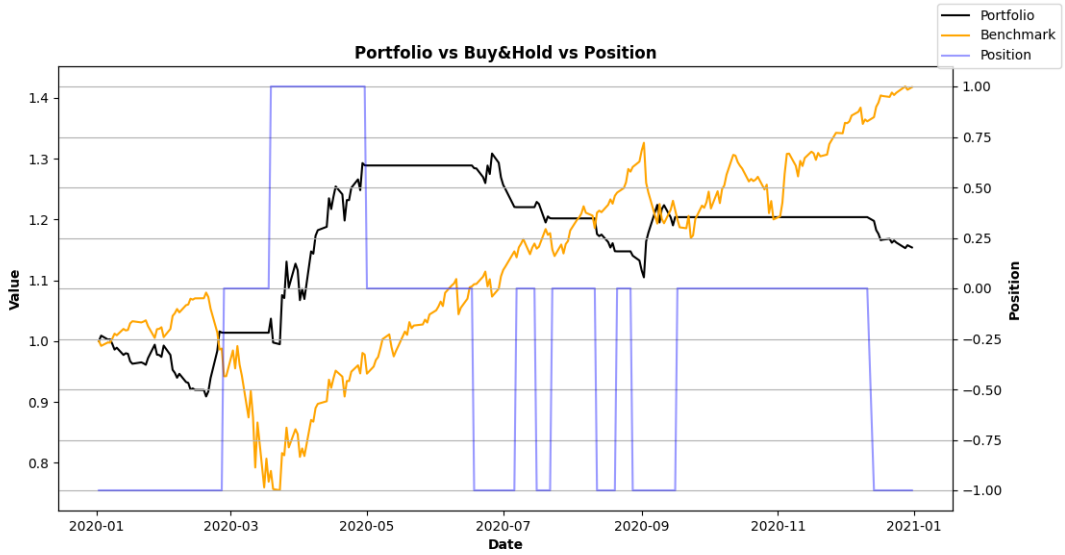
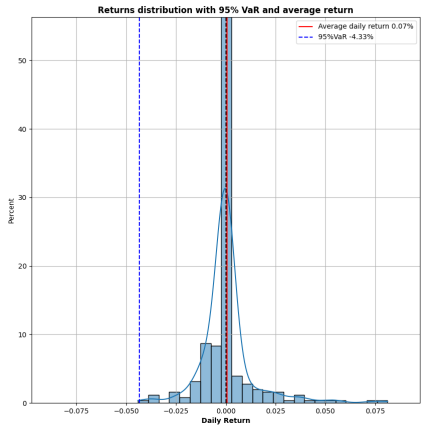
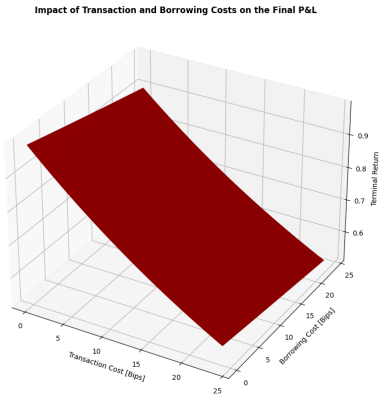


Fig. A.29: Agent performance vs Benchmark, $IXIC, 2020



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.30: $IXIC, 2020

A.6 $N225

	$N225					
	Agent			Benchmark		
	2018	2019	2020	2018	2019	2020
Volatility (%)	15.79	9.77	21.93	19.38	13.91	26.61
Return (%)	6.64	19.63	-12.0	-11.76	20.93	18.27
Sharpe Ratio	0.55	2.04	-0.52	-0.66	1.55	0.81
Maximum Drawdown (%)	14.39	4.32	30.66	20.37	9.17	31.15
Calmar Ratio	1.79	12.78	-2.07	-3.52	10.65	3.76
Semi Volatility (%)	14.49	7.78	23.48	16.03	9.11	18.32
Sortino Ratio	0.6	2.56	-0.49	-0.79	2.37	1.18
Value at Risk (%)	-1.62	-0.88	-2.19	-2.24	-1.37	-2.56
Conditional Value at Risk (%)	-2.64	-1.3	-3.71	-3.29	-1.94	-3.85

Table A.6: Summary statistics for the test years on $N225

Year 2018

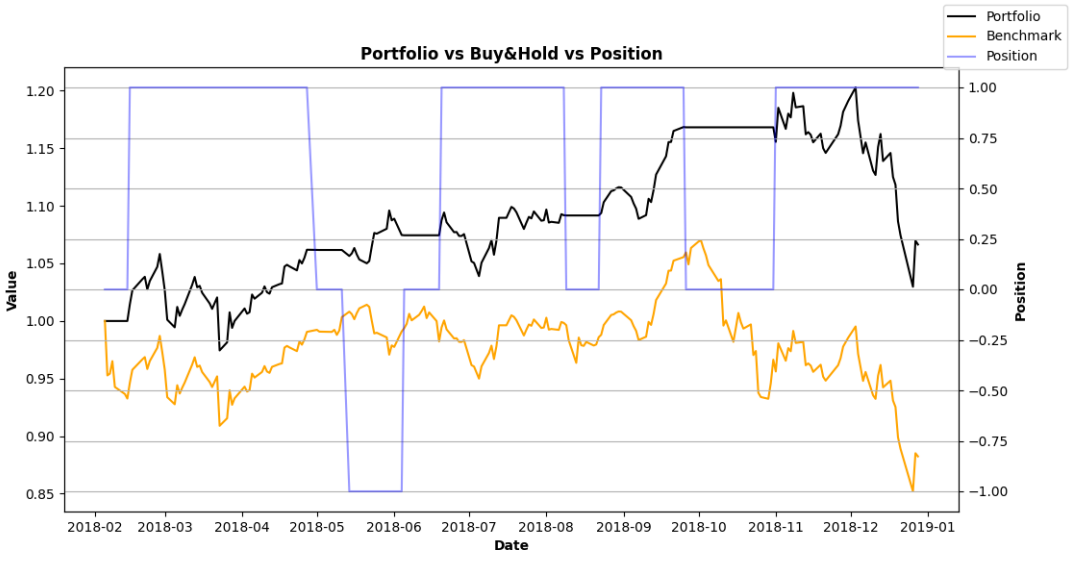
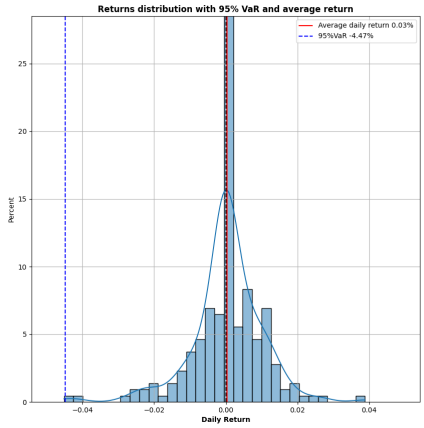
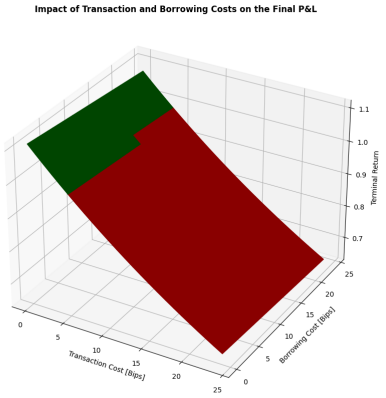


Fig. A.31: Agent performance vs Benchmark, $N225, 2018



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.32: $N225, 2018

Year 2019

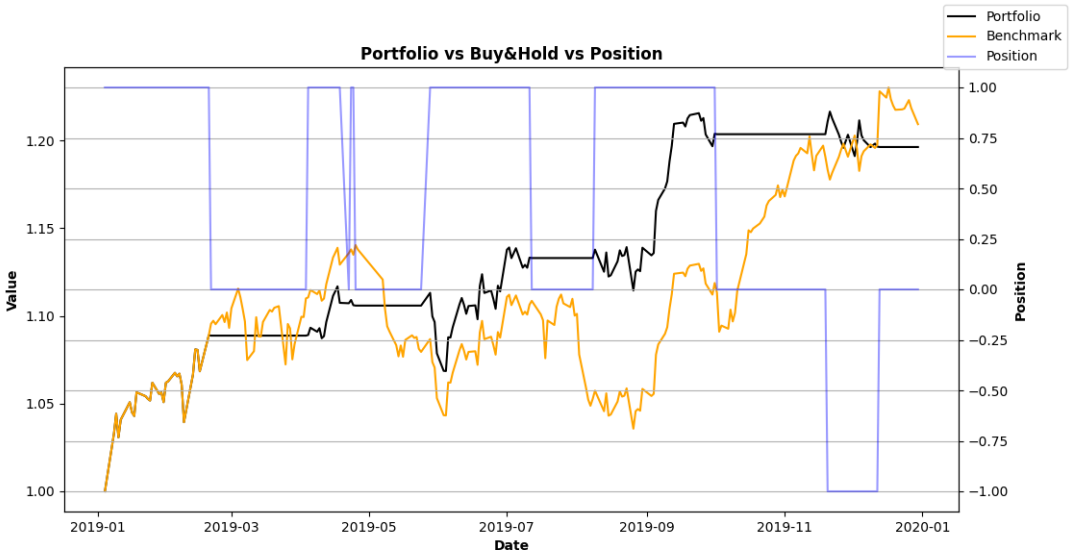
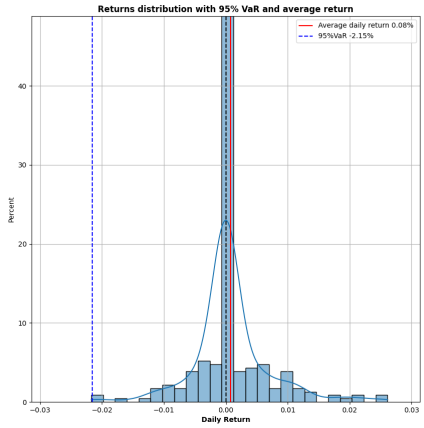
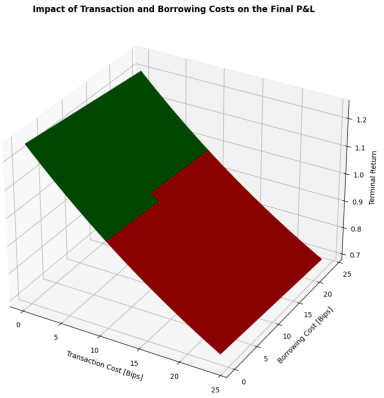


Fig. A.33: Agent performance vs Benchmark, $N225, 2019



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.34: $N225, 2019

Year 2020

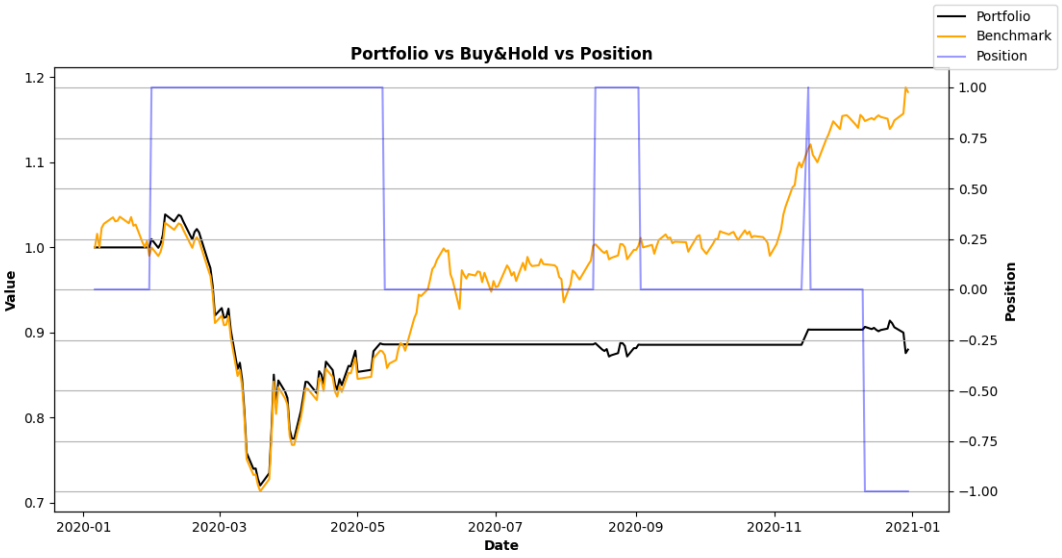
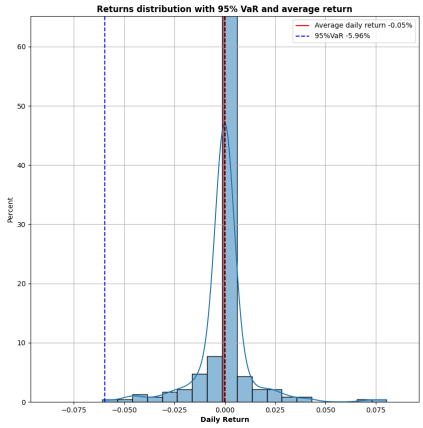
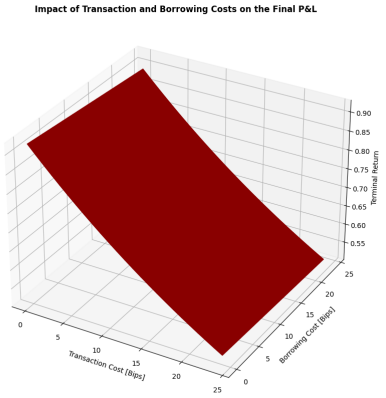


Fig. A.35: Agent performance vs Benchmark, $N225, 2020



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.36: $N225, 2020

A.7 $PG

	PG					
	Agent			Benchmark		
	2018	2019	2020	2018	2019	2020
Volatility (%)	18.73	13.14	27.0	20.01	16.57	32.52
Return (%)	1.34	-5.43	-9.04	6.46	36.83	12.75
Sharpe Ratio	0.17	-0.36	-0.22	0.44	1.98	0.53
Maximum Drawdown (%)	17.85	15.26	21.33	17.84	6.37	23.16
Calmar Ratio	0.44	-4.33	-3.08	1.78	13.53	3.11
Semi Volatility (%)	13.9	10.31	30.34	13.76	11.08	26.23
Sortino Ratio	0.23	-0.46	-0.19	0.64	2.97	0.66
Value at Risk (%)	-1.93	-1.17	-1.73	-1.85	-1.61	-3.26
Conditional Value at Risk (%)	-2.64	-1.79	-4.31	-2.84	-2.26	-5.22

Table A.7: Summary statistics for the test years on $PG

Year 2018

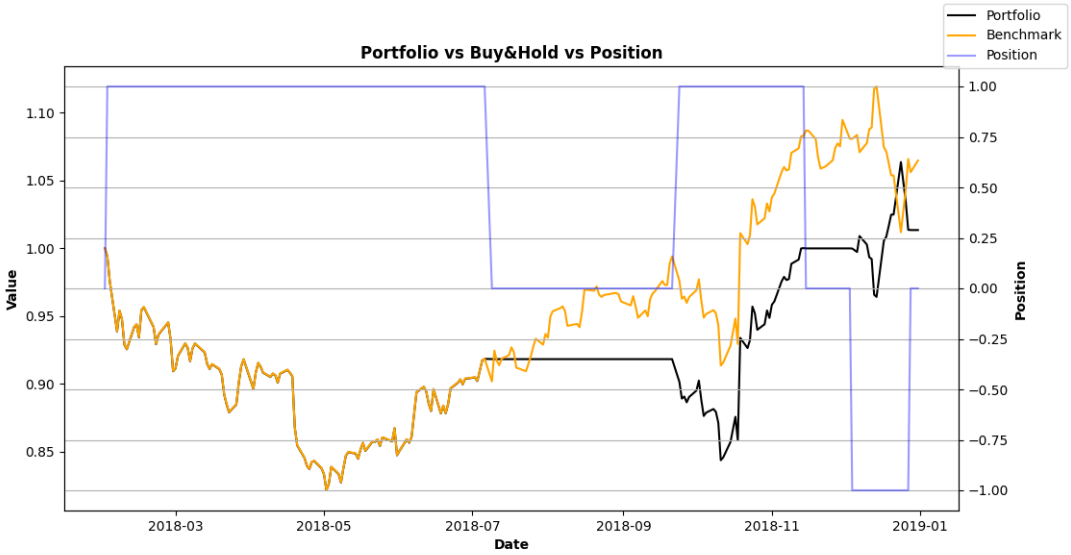
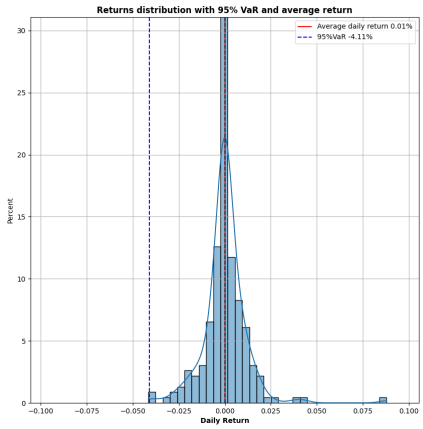
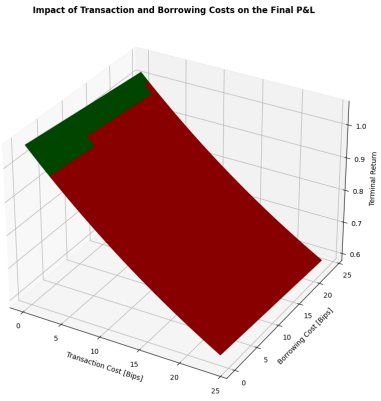


Fig. A.37: Agent performance vs Benchmark, $PG, 2018



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.38: $PG, 2018

Year 2019

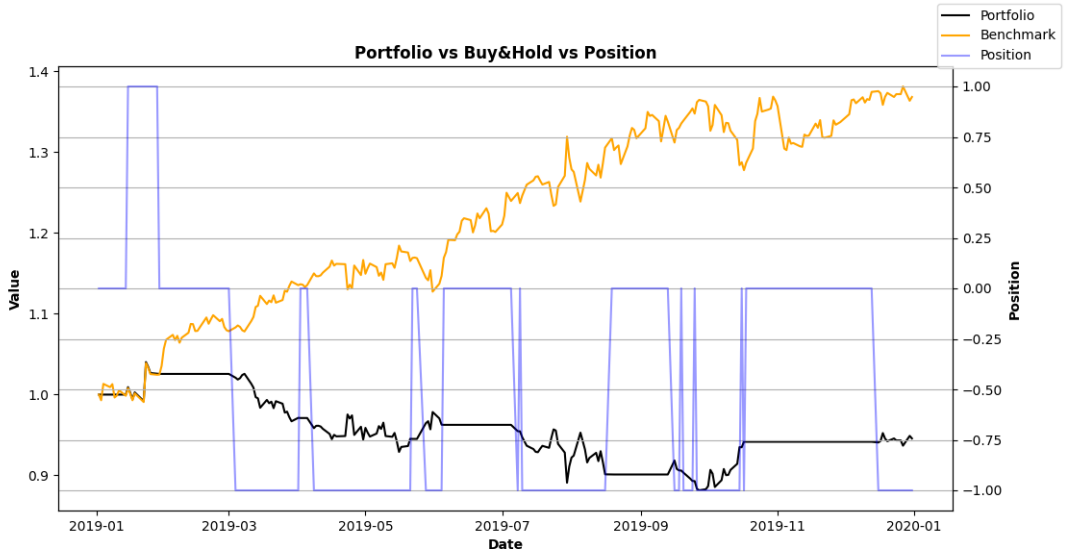
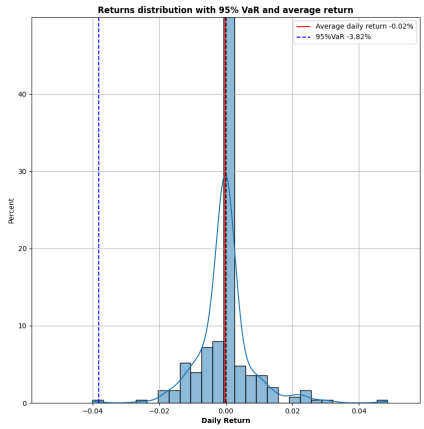
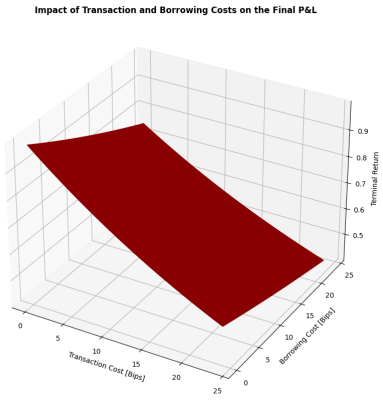


Fig. A.39: Agent performance vs Benchmark, $PG, 2019



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.40: $PG, 2019

Year 2020

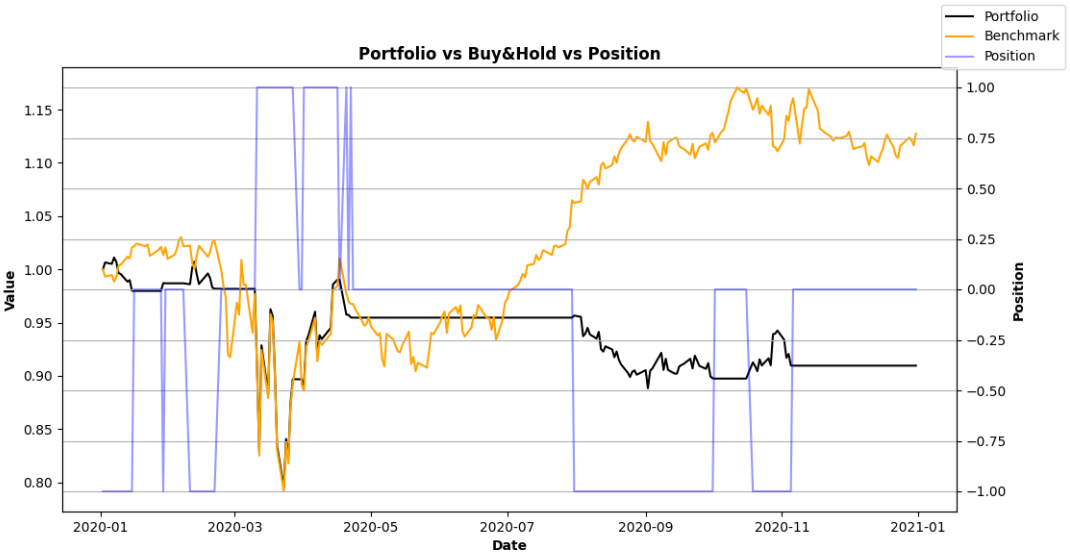
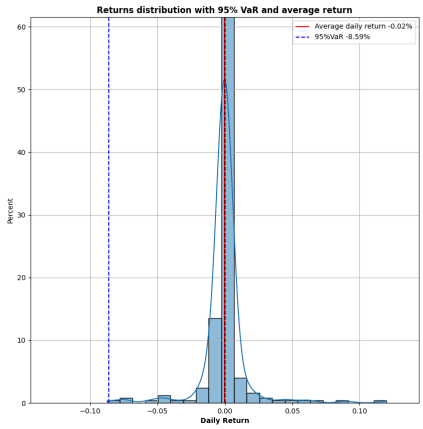
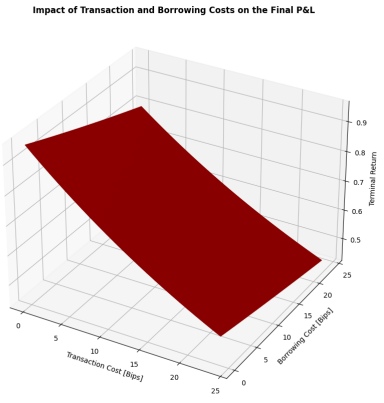


Fig. A.41: Agent performance vs Benchmark, $PG, 2020



(a) Returns distribution of the Agent's Portfolio



(b) Impact of Transaction and Borrowing costs on the raw return of the Agent

Fig. A.42: $PG, 2020

Bibliography

Mean-Risk Analysis with Risk Associated with Below-Target Returns // The American Economic Review. 1975. 67, 2. 116–126.

Agarwal Praveen, Jleli Mohamed, Samet Bessem. Banach Contraction Principle and Applications // Fixed Point Theory in Metric Spaces: Recent Advances and Applications. Singapore: Springer Singapore, 2018. 1–23.

Artzner Philippe, Delbaen Freddy, Jean-Marc Eber, Heath David. Coherent Measures of Risk // Mathematical Finance. 07 1999. 9. 203 – 228.

Bellman Richard. Dynamic Programming. 1957a.

Bellman Richard. A Markovian Decision Process // Journal of Mathematics and Mechanics. 1957b. 6, 5. 679–684.

Bergstra James, Bengio Yoshua. Random Search for Hyper-parameter Optimization // Journal of Machine Learning Research. 2012. 13, Feb. 281–305.

Bishop C.M. Neural networks for pattern recognition. 1995.

Choi Jaehyung. Maximum Drawdown, Recovery, and Momentum // Journal of Risk and Financial Management. 2021. 14, 11.

Franceschi Luca, Donini Michele, Frasconi Paolo, Pontil Massimiliano. Forward and Reverse Gradient-Based Hyperparameter Optimization // Proceedings of

the 34th International Conference on Machine Learning-Volume 70. 2017. 1165–1173.

Hamilton James D. Time Series Analysis. 1994.

Hastie Trevor, Tibshirani Robert, Friedman Jerome. The Elements of Statistical Learning. New York, NY, USA: Springer New York Inc., 2001. (Springer Series in Statistics).

Howard R. A. Dynamic Programming and Markov Processes. Cambridge, MA: MIT Press, 1960.

Huang Chien Yi. Financial Trading as a Game: A Deep Reinforcement Learning Approach. 2018.

Kuhn Max, Johnson Kjell. Applied Predictive Modeling. New York, NY: Springer, 2018.

McNeil Alexander J., Frey Rüdiger, Embrechts Paul. Quantitative Risk Management: Concepts, Techniques and Tools. 2015. Revised edition. (Economics Books).

Mnih Volodymyr, Kavukcuoglu Koray, Silver David, Graves Alex, Antonoglou Ioannis, Wierstra Daan, Riedmiller Martin. Playing Atari with Deep Reinforcement Learning. 2013.

Patro S Gopal, Sahu Dr-Kishore Kumar. Normalization: A Preprocessing Stage // IARJSET. 03 2015.

Pires Ivan, Hussain Faisal, Garcia Nuno, Lameski Petre, Zdravevski Eftim. Homogeneous Data Normalization and Deep Learning: A Case Study in Human Activity Classification // Future Internet. 11 2020. 12.

Rao A., Jelvis T. Foundations of Reinforcement Learning with Applications in Finance. 2022. 1st.

Rasmussen C. E., Williams C. K. I. Gaussian Processes for Machine Learning. 2006.

Rollinger Thomas, Hoffman Sabrina. Sortino Ratio: A Better Measure of Risk // Futures Magazine. February 2013. 40–42.

Rossi R.J. Mathematical Statistics: An Introduction to Likelihood Based Inference. 2018.

Schulman John, Levine Sergey, Moritz Philipp, Jordan Michael I., Abbeel Pieter. Trust Region Policy Optimization. 2017a.

Schulman John, Moritz Philipp, Levine Sergey, Jordan Michael, Abbeel Pieter. High-Dimensional Continuous Control Using Generalized Advantage Estimation. 2018.

Schulman John, Wolski Filip, Dhariwal Prafulla, Radford Alec, Klimov Oleg. Proximal Policy Optimization Algorithms. 2017b.

Sharpe William F. Mutual Fund Performance // Journal of Business. January 1966. 119–138.

Silver David, Huang Aja, Maddison Chris J., Guez Arthur, Sifre Laurent, Driessche George van den, Schrittwieser Julian, Antonoglou Ioannis, Panneershelvam Veda, Lanctot Marc, Dieleman Sander, Grewe Dominik, Nham John, Kalchbrenner Nal, Sutskever Ilya, Lillicrap Timothy, Leach Madeleine, Kavukcuoglu Koray, Graepel Thore, Hassabis Demis. Mastering the game of Go with deep neural networks and tree search // Nature. 2016. 529, 7587. 484–489.

Sutton Richard S., Barto Andrew G. Reinforcement Learning: An Introduction. 2018. Second.

Tesauro Gerald. Temporal Difference Learning and TD-Gammon // Commun. ACM. mar 1995. 38, 3. 58,Äì68.

Watkins Christopher J. C. H., Dayan Peter. Q-learning // Machine Learning. May 1992. 8, 3. 279–292.

Williams Ronald J. Simple statistical gradient-following algorithms for connectionist reinforcement learning // Machine Learning. May 1992. 8, 3. 229–256.

Young Terry W. Calmar Ratio: A Smoother Tool // Futures. October 1 1991.