UCLouvain

FACULTE DES SCIENCES

Faculté des sciences

# Reinforced Learning and Markov Decision Processes

Author: **Joachim NDEZE**
Supervisor: **Eugen PIRCALABELU**
Reader: **Rainer VON SACHS**
Academic year 2023–2024
Master [120] in statistics

# Acknowledgment

I want to express my sincerest gratitude to my advisor, Eugen Pircalabelu for his guidance and advice. The journey to write this thesis was not easy and he did not let me down. I would also like to thank my family and friends for their unconditional support and love. That helped me to stay motivated until the end of the journey.

Finally, I would like to thank Université catholique de Louvain for giving me a high quality education and providing me with all the resources needed to pursue my studies to the full.

# Contents

# Chapter 1

# Introduction

In the field of artificial intelligence and machine learning, the intersection of reinforcement learning and Markov decision processes (MDPs) represents an interesting crossroads, offering interesting possibilities for the development of intelligent systems capable of making autonomous decisions in complex and dynamic environments. This thesis will explore the theoretical foundations of Reinforcement Learning, the relationship between Reinforcement Learning and MDPs, and practical applications.

The goal of this thesis is to study how reinforcement learning uses the structured representation of an environment offered by MDPs, and how MDPs, in turn, benefit from the learning capabilities of reinforcement learning algorithms. First, we will cover the basics of reinforcement learning by explaining some important concepts such as the bandit problem and the exploration-exploitation trade-off. Then, we will present Markov Decision Processes, their properties, their relationship with the Bellman equation and value functions. We will also present some of the most popular reinforcement learning algorithms that are part of the fundamental knowledge on the relationship between reinforcement learning and MDPs. Finally, we will apply the algorithms presented in the thesis to real-life use cases, training intelligent systems using the different algorithms presented and evaluating their performance. We will also have a literature review to understand how reinforcement learning is used in finance and apply reinforcement learning for option pricing. This thesis aims to provide the reader with all the tools and knowledge needed to understand the fundamental concepts of reinforcement learning and MDPs. It also aims to analyze how reinforcement learning can be used in finance and if it can be an alternative to the famous Black & Scholes (B&S) model. Consequently, after reading the thesis, the reader should be armed with all the necessary tools to deepen their knowledge about reinforcement learning and its use of Markov decision processes.

# Chapter 2

# Introduction to Reinforcement Learning

The concept of reinforcement learning, as we know it today, began to be referred to as such in the early 1980s. Before, the subject was evolving in two separate areas. On one hand, there was the study of learning through trial and error, originally used in the psychology of animal learning. On the other hand, there was the study of optimal control using value functions and dynamic programming as a solution. Then, in the early '80s, these study subjects converged to form what we now know as reinforcement learning.

As mentioned in the introduction, reinforcement learning is a framework where an agent learns to adopt the best behavior by interacting with its environment. That interaction is translated into sequences of actions and feedback that can be either a reward or a penalty. So given a specific goal, an agent will find an optimal policy during its learning period by maximizing its total reward. Thanks to the policy obtained, the agent will be able to take optimal decisions in other similar environments.

Let us take the example of an automatic carpet vacuum cleaner. Suppose the vacuum cleaner is placed at a random position on a carpet and has to learn how to clean the carpet without going beyond the area covered by the carpet. Let us also assume that the agent receives a reward of +10 each time he manages to clean the entire surface without exceeding the carpet zone, and a penalty -10 each time he exceeds the carpet zone. When it finishes cleaning the carpet or goes beyond the carpet area, we take the agent back and repeat the experiment several times. After repeating this experience for a sufficiently large number of times, the agent will find an optimal policy to follow and will be able to clean the carpet and apply that policy on a different carpet.

## 2.1 Vocabulary and Definition of Notation

Before going any further, it is necessary to define certain notations which will be used throughout the thesis to make it easier to read. These notations are inspired by Sutton and Barto [1998]. We denote them by

- $t$ a time step;
- $T$ final time step of an interval;
- $S$ set of all possible states;
- $s$ a particular state;
- $s_t$ state observed at time step $t$;
- $A$ a set of actions;
- $a$ particular action;
- $A(s)$ set of possible actions when in state $s$;
- $a_t$ action taken in time step $t$;
- $R$ set of possible rewards;
- $r$ reward;
- $r(a)$ reward received when action $a$ is taken;
- $r_t$ reward received at time $t$;
- $r_t(a)$ reward received at time step $t$ when action $a$ is taken;
- $\pi$ a policy, decision rule;
- $\Pi$ set of policies;
- $\pi(s)$ action/decision taken when in state $s$;
- $\pi(a|s)$ probability of taking the action $a$ when in state $s$.

Now, let us define some important concepts that will be discussed in the thesis.

**Rewards** represent the actual rewards of taking an action. Rewards are generated from an unknown stationary or nonstationary probability distribution that depends on the state and/or the action chosen. We define a stationary probability distribution as a probability distribution that stays unchanged as time steps progress. A nonstationary probability distribution is defined as a probability distribution that changes as time steps progress. By

interacting with the environment, the agent receives feedback, as a form of reward, from the environment that will help it to find a strategy that maximizes its total reward over a given number of time steps.

**Policy** defines the behavior of an agent. It maps a given state to an action. The goal of a reinforcement learning problem is to find an optimal policy that maximizes the total expected reward. Therefore, policies play a major role in reinforcement learning problems. The policy is represented by $\pi$, where $\pi(s)$ represents the action taken when in state $s$ following the policy $\pi$, and $\pi(a|s)$ is the probability of taking action $a$ when in state $s$ following the policy $\pi$.

**Transition probabilities** are the probabilities of moving from one state to another possible successor state.

The **Value function** of a state $s$ represents the value of being in that state. It denotes the total future reward that an agent can expect to receive in that state. The value function determines, for an agent, how good it is in the long run to be in a given state. The value function is important in solving the reinforcement learning problem, as many algorithms are based on value function optimization. The fact that value functions provide information about the long-term worth of being in a state is crucial, as in most reinforcement learning problems, the goal is to optimize the long-term total expected reward.

The **model** refers to a model of the agent's environment. It can predict the next state $s'$ and reward $r$ when an agent is in a given state $s$ and selects an action $a$. In other terms, it is a model that can mimic the reactions of the environment with which the agent interacts. Reinforcement learning is often separated into two different computation strategies: model-free and model-based. In a model-free strategy, the agent will only try to solve a task, while in a model-based strategy, the agent will first try to learn all the rules of its environment before attempting to solve a task.

An **episode** is characterized by a starting state that starts the episode and a terminal state that ends the episode. Between these two states there is a sequence of interactions between an agent and an environment, in the form of actions by the agent and feedback (rewards or penalties) from the environment.

## 2.2 Introductory Example: The Bandit Problem

To illustrate a simple reinforcement learning problem, the three-armed bandit problem is a good example. Suppose an agent faces a machine, similar to a casino slot machine, with three levers named A, B, and C, and has to choose sequentially between the three levers over a total of $T$ time steps. Therefore we can define the set of actions $A = \{$pull lever A, pull lever B, pull lever C$\}$. For each moment $t = 1, 2, ..., T$, the agent takes an

action $a_t \in A$, and receives a reward $r_t$, a random variable, generated from an unknown stationary probability distribution that depends on the lever selected. Suppose that the reward can be +1 or 0. Since the probability distribution is stationary, thanks to interactions with the environment at times $t = 1, 2, ..., T$, the agent should be able to find the best lever and maximize its rewards by concentrating its actions on it. The three-armed bandit problem can be generalized as the $n$-armed bandit problem with $n$ levers instead of three.

For a given set of actions $A$, we define the true value of an action $a \in A$ as $q(a)$. From Sutton and Barto [1998] and Hasselt [2010], the value of an action $a$ is the expected value of the random variable $r(a)$, the reward of taking only the action $a$ which is defined as

$$q(a) = \mathbb{E}[r(a)] \ \forall \ a \in A. \tag{2.2.1}$$

As mentioned at the beginning of the section, the probability distribution of the rewards is unknown. Therefore, a method to estimate $q(a)$ is needed.

A simple and efficient way to do it, is the sample-average method. Suppose we are at a moment $t$, where $t \in \{1, 2, ..., T\}$, and we want to estimate $q(a) \ \forall \ a \in A$. Let us say that at the $t$th moment, the action $a$ has been taken a total of $N_t(a)$ times before and gave the sample of rewards $r_1(a), r_2(a), ..., r_{N_t}(a)$. We assume that the sample $r_1(a), r_2(a), ..., r_{N_t}(a)$ is independent and identically distributed (i.i.d). Then $Q_t(a)$ the sample-average estimated value of $q(a)$, can be obtained as follows:

$$Q_t(a) = \frac{\sum_{i=1}^{N_t(a)} r_i(a)}{N_t(a)}. \tag{2.2.2}$$

From Hasselt [2010] we could say that $Q_t(a)$ is an unbiased estimator of $q(a)$, which can be shown as follows:

$$\begin{aligned} \mathbb{E}[Q_t(a)] &= \mathbb{E}\Big[\frac{\sum_{i=1}^{N_t(a)} r_i(a)}{N_t(a)}\Big] \\ &= \mathbb{E}[r_i(a)] \\ &= q(a). \end{aligned} \tag{2.2.3}$$

Moreover, suppose the rewards $r_1(a), r_2(a), ..., r_{N_t}(a)$ are i.i.d, then by the law of large numbers, we can say that when $N_t(a) \to \infty$, $Q_t(a)$ converges in probability to $q(a)$. When $N_t(a) = 0$, then a default value is assigned to $Q_t(a)$, and in general, that value is set to $Q_t(a) = 0$. This is a simple method to compute action-values but will be useful to show how action-values can be used.

To apply the sample-average method to estimate $q(a)$ for an action $a \in A$, one could use formula (2.2.2). However, the problem with this formula is that the more times the action $a$ is chosen, the more memory and calculations this formula requires to compute $Q_t(a)$.

Inspired by equation (2.2.2), Sutton and Barto proposed to estimate $Q_{t+1}$ recursively as follows:

$$\begin{aligned} Q_{t+1}(a) &= \frac{1}{t}\sum_{i=1}^{t} r_i(a) \\ &= \frac{1}{t}\Big(r_t(a) + \sum_{i=1}^{t-1} r_i(a)\Big) \\ &= \frac{1}{t}\Big(r_t(a) + (t-1)Q_t(a) + Q_t(a) - Q_t(a)\Big) \\ &= \frac{1}{t}\Big(r_t(a) + tQ_t(a) - Q_t(a)\Big) \\ &= Q_t(a) + \frac{1}{t}\Big[r_t(a) - Q_t(a)\Big]. \end{aligned} \tag{2.2.4}$$

Using formula (2.2.4) requires less memory and smaller calculations than formula (2.2.2) for every new reward. This formula is also used as an update rule for new estimates and can be generalized as follows:

*New Estimate* $\leftarrow$ *Old Estimate* + *Step Size* [*Previous Reward* - *Old Estimate*].

Until now, everything presented was applied to the case when the rewards were generated from a stationary probability distribution. Now, we will see how it works when the rewards are generated from a nonstationary probability distribution, which happens more frequently. In this situation, it is good practice to give more weight to recent rewards, and a common way of doing that is to use a constant step-size parameter $\alpha$ [Sutton and Barto, 1998]. Thus the formula (2.2.4) becomes:

$$Q_{t+1}(a) = Q_t(a) + \alpha\Big[r_t(a) - Q_t(a)\Big] \tag{2.2.5}$$

where $\alpha \in (0, 1]$ is constant. This can be rewritten as follows

$$\begin{aligned} Q_{t+1}(a) &= Q_t(a) + \alpha\Big[r_t(a) - Q_t(a)\Big] \\ &= \alpha r_t(a) + \Big(1-\alpha\Big)Q_t(a) \\ &= \alpha r_t(a) + \Big(1-\alpha\Big)\Big[\alpha r_{t-1}(a) + \Big(1-\alpha\Big)Q_{t-1}(a)\Big] \\ &= \alpha r_t(a) + \Big(1-\alpha\Big)\alpha r_{t-1}(a) + \Big(1-\alpha\Big)^2 Q_{t-1}(a) \\ &= \alpha r_t(a) + \Big(1-\alpha\Big)\alpha r_{t-1}(a) + \Big(1-\alpha\Big)^2 \alpha r_{t-2}(a) + ...+ \\ &\quad (1-\alpha)^{t-1}\alpha r_1(a) + (1-\alpha)^t Q_1(a) \\ &= \Big(1-\alpha\Big)^t Q_1(a) + \alpha\sum_{i=1}^{t}(1-\alpha)^{t-i} r_i(a). \end{aligned} \tag{2.2.6}$$

Since the sum of the weights $(1-\alpha)^t + \sum_{i=1}^{t} \alpha(1-\alpha)^{t-i}$ is equal to 1, we can refer to this as a weighted average. One can also clearly see that less weight is assigned to the first rewards, and more weight is assigned to recent rewards to take into account the impact of the non-stationary environment.

To ensure convergence, Schmetterer [1961] presented stochastic approximation theory and demonstrated that the following conditions guarantee convergence with probability 1:

$$\sum_{k=1}^{\infty} \alpha_k(a) = \infty \quad and \quad \sum_{k=1}^{\infty} \alpha_k^2(a) < \infty \tag{2.2.7}$$

where $\alpha_k(a)$ is the step size parameter used after selecting the action $a$ for $k$ times. In the stationary case, where $\alpha_k(a) = \frac{1}{k}$, the conditions are respected to ensure convergence. However, in the nonstationary case, the second condition is not respected. This implies that the estimator does not converge in probability to $q(a)$, and instead, they fluctuate based on the most recent rewards received.

## 2.3 The Exploration Versus Exploitation Tradeoff

Based on its current knowledge, one method an agent can use to maximize its total expected reward is to take the greedy action $a_t^*$ at each time step $t = 1, 2, 3, ..., T$. This method is also known as the **greedy strategy**, and $a_t^*$ can be defined as:

$$a_t^* = \arg\max_a Q_t(a). \tag{2.3.1}$$

When an agent adopts a greedy strategy, it only exploits its current knowledge and never explores its environment. Kaelbling et al. [1996] explained that unlucky sampling reflecting the agent's current knowledge might show that the best action to take is not the actual optimal action. Therefore, the agent can be stuck with a combination of suboptimal actions instead of the actual optimal combination of actions and suboptimal actions will always be picked, and the agent will not achieve its goal. Similarly, suppose the agent only chooses to explore its environment and never uses the information at its disposal. In that case, the agent might miss the optimal combination and not maximize its expected reward.

An alternative to greedy strategies is the **$\epsilon$-greedy strategy**. The idea of $\epsilon$-greedy is to behave greedily with probability $1-\epsilon$ and to select a random action uniformly among all the possible actions with probability $\epsilon$. Tokic and Palm [2011] defined $\epsilon$-greedy strategy as follows

$$a_t = \begin{cases} \text{uniformly chosen at random from } A(s_t) & \text{with probability } \epsilon \\ a_t^* & \text{with probability } 1-\epsilon \end{cases} \tag{2.3.2}$$

where $a_t$ is the action of the agent at time $t$, and $A(s_t)$ is the set of possible actions when the state $s_t$ occurs. This method ensures that if the number of plays tends to infinity,

then for all $a$, as $N_t(a) \to \infty$, $Q_t(a)$ converges to $q(a)$ [Sutton and Barto, 1998]. So, as the number of plays increases, the action-value estimates become better. The $\epsilon$-greedy algorithm can be implemented numerically as in Algorithm 1.

**Algorithm 1** Pseudo code to implement $\epsilon$-greedy algorithm on a decision making problem

**Input** : $\epsilon$

Initialize $Q(a)$
**for** $t = 1 \to T$ **do**
    With probability (1-$\epsilon$):
        Select $a = \arg\max_a Q_t(a)$
    otherwise:
        Select action $a$ uniformly at random among $A(s)$
    $Q(a) \leftarrow Q_t(a)$
**end for**

To have a better understanding of how the epsilon-greedy strategy works, let us look at the results of the $\epsilon$-greedy algorithm on a 3-arm bandit problem when $\epsilon = 0.00$, $\epsilon = 0.01$, and $\epsilon = 0.10$ over five thousand time steps. The environment of the 3-arm bandit is designed such that at each time step, lever A gives a reward of 3 with a probability of 0.25 or a reward of 0 with a probability of 0.75. Lever B gives a reward of 3 with a probability of 0.50 or a reward of 0 with a probability of 0.50. Lever C gives a reward of 3 with a probability of 0.75 or a reward of 0 with a probability of 0.25. The agent does not know that information and must find the best lever. Before going directly into the results, let us have a numerical illustration of how the algorithm works and what it does. In table 2.3.1,

| 0.1-greedy | | | | | |
|---|---|---|---|---|---|
| Iteration | Random | Action | Reward | $Q_t$(Action) | average reward |
| 1 | no | lever A | 0 | 0.000 | 0.000 |
| 2 | no | lever A | 0 | 0.000 | 0.000 |
| 3 | no | lever A | 0 | 0.000 | 0.000 |
| ... | ... | ... | ... | ... | ... |
| 59 | yes | lever B | 3 | 3.000 | 0.813 |
| ... | ... | ... | ... | ... | ... |
| 9998 | no | lever C | 3 | 2.241 | 2.137 |
| 9999 | no | lever C | 3 | 2.242 | 2.137 |
| 10000 | no | lever C | 0 | 2.241 | 2.137 |

Table 2.3.1: Illustration of the 0.1-greedy algorithm on a 3-arm bandit problem, where Lever A, B and C can give a reward of 3 (or 0) with probability 0.25 (0.75), 0.5 (0.50) and 0.75 (0.25) respectively.

we can see that for the first three iterations, the algorithm does not randomly select an

action and selects lever A. For those three first steps, no reward (0) was received, therefore the $Q_t$(lever A) stays equals to zero as well as the average reward. Then at iteration 59, where the algorithm has randomly selected a lever, it selected lever B, which gave a reward of 3, and the value of $Q_t$(lever B) became 3.000. The average reward on the 59th step is updated to 0.813. Finally, we can see the three final steps of the algorithm and we can see that the algorithm select the lever C, which is the best one.

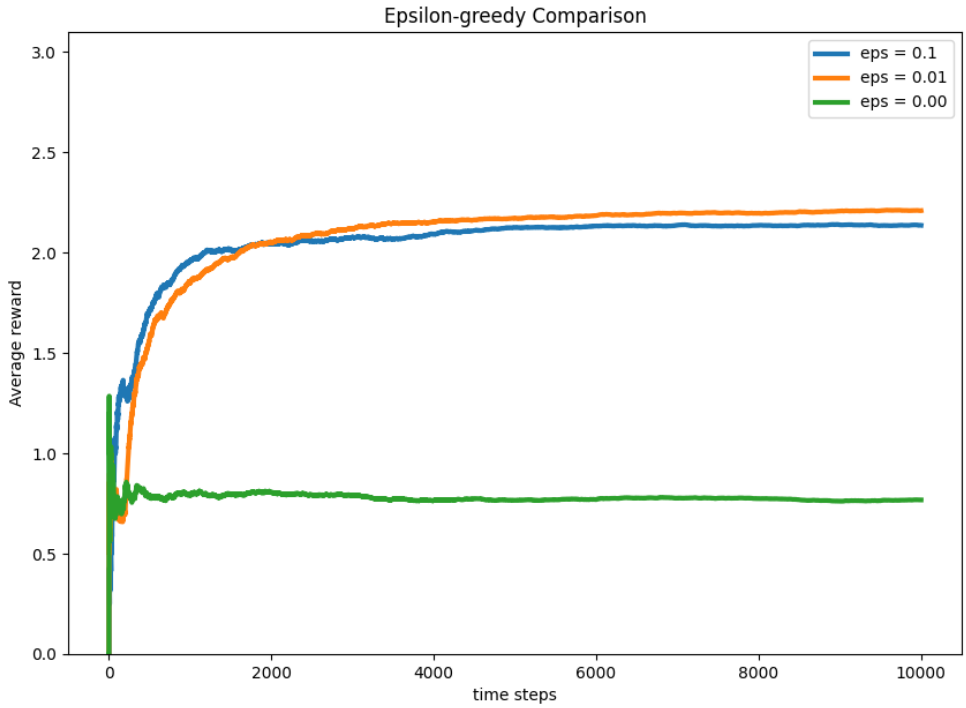



Figure 2.3.1: An $\epsilon$-greedy strategy on a 3-arm bandits problem in three different exploring scenarios. For the blue line, the exploring factor $\epsilon = 0.1$. For the green line $\epsilon = 0.01$. For the yellow line $\epsilon = 0.00$

In figure 2.3.1, what is interesting is that when $\epsilon = 0.01$ and $\epsilon = 0.10$, they outperform the case where $\epsilon = 0$ because no exploration is done when $\epsilon = 0$. Therefore, there is no chance to find the best lever. Then it also appears that when $\epsilon = 0.10$ the algorithm performs better at the beginning, but after approximately 2000 iterations, $\epsilon = 0.01$ accomplishes better performances. This can be explained because in the long run, when $\epsilon = 0.10$, the agent will tend to continue exploring its environment, and therefore the agent will often choose sub-optimal actions which will reduce the average reward. Since less exploration is done in the case where $\epsilon = 0.01$, in the long run it performs better because the impact of

selecting sub-optimal actions is smaller.

A popular variation of that algorithm is to decrease the probability of choosing at random at each successive round. That method is supported by the fact that after a large number of iterations, exploration is no longer useful since $Q_t(a)$ converges to $q(a)$, and therefore no more exploration is needed because the solution will appear. We can clearly see in Figure 2.3.1 that exploration is not optimal in the long run.

Choosing whether an agent should go for greedy strategies, $\epsilon$-greedy, decaying $\epsilon$-greedy, or simply choosing the value of $\epsilon$, depends on the task and the dynamics of the environment.

# Chapter 3

# Markov Decision Processes And Their Application To Reinforcement Learning

According to Sutton and Barto [1998], if one understands Markov Decision Processes (MDP), he is able to understand 90% of modern reinforcement learning. MDPs are used to explain reinforcement learning in a broader sense. The thesis will focus on finite MDPs.

This chapter will cover MDPs, starting with the fundamental concepts and going all the way through to the links with reinforcement learning. It will also introduce important concepts such as value functions, optimal value functions and the Bellman equation.

Finally, at the end of the chapter an example will be used to illustrate all the concepts presented.

## 3.1 Fundamentals of Markov Decision Processes

### 3.1.1 Sequential Agent-Environment Interaction

MDPs are known to be sequential decision models because of the relationship between the agent and the environment it interacts with. At a specified moment $t$, where $t = 1, 2, 3, ..., T$, the agent faces an environment and is in the state $s_t$. Depending on the state $s_t$, the agent will take an action $a_t$ from the set $A(s_t)$. As a result of this action, the agent will receive an immediate reward (or immediate cost) $r_{t+1}(a_t)$, and the environment will evolve towards a new state $s_{t+1}$ based on a probability distribution driven by the choice of $a_t$. At this new time step, the agent will face a new state $s_{t+1}$ and a new set of actions $A(s_{t+1})$ to choose from. The interaction between the agent and the environment is illustrated on figure 3.1.1. Putterman [1994] presented the key components for a sequential decision model as follows:

- A set of decisions times $t = 1, 2, 3, ...$;

- A set of states $s_t$;
- A set of actions $A(s_t)$ for all $t = 1, 2, 3, ...$;
- A set of states $s_t$ and actions $a_t$ on which the reward $r_{t+1}$ depends directly;
- A set of state $s_t$ and action $a_t$ on which depends the transition probability to the state $s_{t+1}$ which is defined as $p(s_{t+1} = s', r_{t+1} = r|s_t = s, a_t = a)$ where $\forall s', s \in S$. The transition probability is also written as follows in some cases $p(s', r|s, a)$. That probability represents the probability that the state and reward at time $t + 1$ are respectively equal to $s'$ and $r$, knowing that the current state is $s_t$ and the action is $a_t$

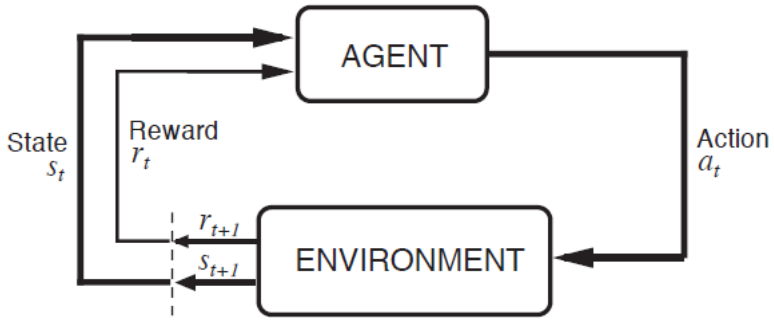



Figure 3.1.1: Sequential agent-environment interaction in Markov Decision Processes. source: Sutton [1997]

At each step, the decision maker applies a mapping to make a choice that will help achieve its goal. This mapping gives the probability of selecting an action when the agent is in a given state. It is called the agent's policy and is denoted as $\pi_t$, where $\pi_t(a|s)$ is the probability that the action $a_t$ is taken when the agent is in state $s$ at time step $t$. The agent's goal is to find an optimal policy that will maximize its total rewards in the long run.

The boundary between the agent and the environment is the limit of the agent's absolute control of the state in which it finds itself. The boundary is not as obvious as one might think. If we take the example of a robot, the boundary between the agent and the environment is not the same as the physical boundary. Generally, it is closer to the agent. To illustrate that, we can take the example of the recycling robot given by Sutton and Barto [1998]. In that example, they explained that a robot's motors, mechanical linkages, and sensors should be considered as parts of the environment instead of parts of the agent. The same applies to rewards; they are likely to be calculated inside the physical body of the learning agent, but they are considered external.

Let us explain the example of the recycling robot from Sutton and Barto [1998] more precisely. A mobile robot is responsible for picking up and depositing empty soda cans in a workplace. Equipped with sensors to identify the cans, an arm, and a gripper to

take hold of them and place them in a trash can, the robot is operated by a rechargeable power source. Its governing system contains parts for interpreting external data, traveling, and controlling the arm and gripper. A reinforcement learning agent executes high-level decisions about searching for cans based on the current battery level. This agent needs to decide whether it should (1) search for a can intensely for a specific time span, (2) stand still and wait that someone comes to throw away a can, or (3) go back to its original station to recharge its battery. Such evaluations must be made periodically or when exclusive occasions occur, like when a can is located. Therefore, the agent has three possible actions, and its state is affected by the battery level. Rewards are oftentimes zero but can be positive if the robot succeeds in catching an empty can or negative if the battery drains completely. In this example, the reinforcement learning agent is not equal to the whole robot. The factors it controls refer to conditions internal to the robot itself, not situations in its outer environment. The agent's environment thus involves the rest of the robot, potentially containing other complicated decision-making systems, plus the robot's external environment.

As mentioned earlier, the thesis will focus on finite MDPs. The principal difference between finite and infinite MDPs is in the number of possible states and/or actions that the agent can face in a given environment. On one hand, there is only a finite number of states and actions that the agent can face for finite MDPs. On the other hand, there is an infinite (continuous) number of states and/or actions that the agent can face for infinite MDP.

## 3.2 Goals, Rewards And Returns

In reinforcement learning, the agent has a goal that is generalized by a sequence of rewards from the environment. Rewards, goals, and returns play essential roles in reinforcement learning, especially given their interconnected nature.

As presented by Singh et al. [2009], reinforcement learning has two different forms of rewards. First, there is the primary reward, which is the result produced by a "critic" that assesses the actions taken by the reinforcement learning agent. Then, there are secondary rewards, which act as predictors of primary rewards and are used by reinforcement learning systems that compute value functions. Both rewards associate each state-action pair with a real value, helping the agent to make optimal decisions regarding the state it is in. Rewards can be positive (e.g., a reward for completing a given task) or negative (e.g., a penalty for not completing a given task).

Goals represent the task or purpose the agent is trying to achieve. They may not be explicitly provided but can be computed from the reward structure.

In the case of an episodic Markov Decision Process (MDP), where the initial state is

reinitialized after each episode, the succession of states, actions, and rewards in an episode constitutes a trajectory of the policy. This situation is called the finite-horizon model [Hauskrecht, 2000]. In this case, Hauskrecht [2000] and Arulkumaran et al. [2017] defined the return as the total rewards accumulated from the environment during an episode, which is denoted by $G_T$:

$$G_T = r_1 + r_2 + r_3 + ... + r_T = \sum_{t=1}^{T} r_t \tag{3.2.1}$$

where $T$ is the length of an episode. This formula is used to evaluate the performance of different policies and guide the agent's decision-making process. The best policy for an agent is the one that maximizes its expected return.

Let $r_{t+1}, r_{t+2}, r_{t+3}, ...$, be the reward received after the $t$th time step. Thus if an agent is at the $t$th time step, the total cumulative reward the agent needs to maximize is the sum of the rewards coming after $t$th time steps, that we define as

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T = \sum_{i=1}^{T-t} r_{t+i}. \tag{3.2.2}$$

In many cases, the agent-environment interactions cannot be easily separated into specific episodes and continues without ending. That case is called the infinite horizon model [Hauskrecht, 2000]. For this type of model, maximizing $G_t$ as defined in 3.2.2, no longer makes sense because the horizon becomes infinite and it follows that the return that needs to be maximized becomes also infinite. Therefore, from Hauskrecht [2000] and Arulkumaran et al. [2017], we have that one must maximize the discounted return which is equal to

$$G_T = r_1 + \gamma r_2 + \gamma^2 r_3 + ... = \sum_{i=1}^{\infty} \gamma^{i-1} r_i, \tag{3.2.3}$$

where $\gamma$ is the discount rate parameter with $0 \leq \gamma \leq 1$, and $T = \infty$. The discount rate is used to value the present worth of future rewards. So, the value of a reward that will be received at the $t$th time step is $\gamma^{t-1} r_t$. In the case of an infinite horizon model, having $\gamma < 1$, ensures that $G_T$ has a finite value. When $\gamma = 0$, the agent is only interested in maximizing its immediate reward $r_1$. When $\gamma$ approaches 1, the agent becomes more forward-looking because more value is given to the future rewards when it tries to maximize its expected discounted return. Finally when $\gamma = 1$, equation (3.2.3) is the same as equation (3.2.1). To make the notation more general, in the rest of the thesis we will use $G_T$ as defined in equation (3.2.3). In the case of an infinite horizon model, equation (3.2.2) becomes

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}. \tag{3.2.4}$$

Defining goals as a sequence of rewards has proven to be very flexible and adaptable to a large range of use cases. However, sometimes, formulating goals in terms of rewards can

confuse the understanding of a goal. Therefore, it is crucial to formulate the goal clearly because that task may be more complex than expected. This is illustrated by the credit assignment concept introduced by Van Otterlo and Wiering [2012].

Let us take the example of a chess-playing agent. One might think that to achieve the agent's goal, which is to maximize the sum of its rewards, it will receive a reward for each good action, such as taking control of the center of the grid or capturing the pieces of its opponent. However, if a reward is given for each sub-goal, this can mislead the agent and encourage him to focus on these sub-tasks instead of achieving his goal, which is to win the game. Therefore, when training a chess-playing agent, you should only reward him for each game it wins. Or give less credit to the sub-task and more credit to winning a game so that the agent is not stuck in achieving sub-goals. It is crucial to understand that reward signals are used to communicate to the agent **what** it needs to do, not **how** it has to do it. To fulfill its goal, it is essential to implement a reward system that truly defines the agent's goal.

## 3.3 The Markov Property

As mentioned in their names, Markov Decision Processes must respect the Markov property. So, let us define what the Markov property is, and then we will see how it applies to reinforcement learning.

Let us define $S = \{s_0, s_1, s_2, ...\}$, a state space, and every $s \in S$ is a state. Let $X$ be a random variable with values in $S$ and probability distribution $\lambda$. So we have that

$$\lambda_s = P(X = s) \; \forall \; s \in S. \tag{3.3.1}$$

Then, for $n \geq 0$, $X_n$ is a Markov chain if [Norris, 1998]

- $X_0$ has distribution $\lambda$,
- the conditional distribution of $X_{n+1}$, given that $X_n = s$, has distribution $p_{su} : u \in S$ where $p_{su} = p(X_{n+1} = u | X_n = s)$ and is independent of $X_0, ...X_{n-1}$.

More explicitly, it states that for $n \geq 0$ and $s_0, ..., s_{n+1} \in S$

- $p(X_0 = s_0) = \lambda_{s_0}$
- $p(X_{n+1} = s_{n+1} | X_0 = s_0, ..., X_n = s_n) = p_{s_n s_{n+1}}$.

**Theorem 3.3.1** (Norris [1998], Theorem 1.1.1)**.** *A discrete-time random process $X_n$, such that $n \in 0 \leq n \leq N$, is Markovian if and only if for all $s_0, ..., s_N \in S$*

$$p(X_0 = s_0, X_1 = s_1, ..., X_N = s_N) = \lambda_{s_0} p_{s_0 s_1} ... p_{s_{N-1} s_N}. \tag{3.3.2}$$

*Proof.* Suppose $X_n$, such that $n \ni 0 \leq n \leq N$, is Markovian, then

$$\begin{aligned} p(X_0 = s_0, X_1 = s_1, ..., X_N = s_N) &= p(X_0 = 0)p(X_1 = s_1|X_0 = s_0) \\ &\quad ...p(X_N = s_N|X_0 = s_0, ..., X_{N-1} = s_{N-1}) \\ &= \lambda_{s_0} p_{s_0 s_1} ... p_{s_{N-1} s_N}. \end{aligned}$$

On the other hand, if (3.3.2) holds for $N$, then by summing both sides over $s_N \in S$ and using $\sum_{u \in S} p_{su} = 1$ we see that (3.3.2) holds for $N - 1$ and, by induction

$$p(X_0 = s_0, X_1 = s_1, ..., X_n = s_n) = p_{s_0} p_{s_0 s_1} ... p_{s_{n-1} s_n}$$

for all $n = 0, 1, ..., N$. In particular, we can continue to show that, for $p(X_0 = s_0) = \lambda_{s_0}$ and, for $n = 0, 1, ..., N - 1$,

$$\begin{aligned} &p(X_{n+1} = s_{n+1}|X_0 = s_0, ..., X_n = s_n) \\ &= p(X_0 = s_0, ..., X_n = s_n, X_{n+1} = s_{n+1})/p(X_0 = s_0, ..., X_n = s_n) \\ &= p_{s_n s_{n+1}}. \end{aligned}$$

So $X_n$, such that $n \ni 0 \leq n \leq N$, is Markovian. □

Now that we defined when a discrete-time random process is Markovian let us see the Markov property and prove that Markov chains have no memory. Before let us define $\delta_{su}$ as

$$\delta_{su} = \begin{cases} 1 & \text{if } s = u \\ 0 & \text{otherwise} \end{cases} \tag{3.3.3}$$

where $s, u \in S$.

**Theorem 3.3.2** (Norris [1998], Markov property Theorem 1.1.2)**.** *Let $X_n$, such that $n \geq 0$, be Markovian. Then, conditional on $X_m = s$, $(X_{m+n})$, where $m \geq 1$, is Markovian and is independent of the random variables $X_0, ..., X_m$.*

*Proof.* We have to show that for any event $E$ determined by $X_0, ..., X_m$ we have

$$\begin{aligned} &p(\{X_m = s_m, ..., X_{m+n} = s_{m+n}\} \cap E|X_m = s) \\ &= \delta_{s s_m} p_{s_m s_{m+1}} ... p_{s_{m+n-1} s_{m+n}} p(E|X_m = s) \end{aligned} \tag{3.3.4}$$

then the result follows by Theorem 3.3.1. First, consider the case of elementary events

$$E = \{X_0 = s_0, ..., X_m = s_m\}.$$

In that case we have to show that

$$\begin{aligned} &p(X_0 = s_0, X_1 = s_1, ..., X_{m+n} = s_{m+n} \text{ and } s = s_m)/p(X_m = s) \\ &= \delta_{s s_m} p_{s_m s_{m+1}} ... p_{s_{m+n-1} s_{m+n}} \\ &\quad \times p(X_0 = s_0, ..., X_m = s_m \text{ and } s = s_m)/p(X_m = s) \end{aligned}$$

which is true by Theorem (3.3.1). In general, any event $E$ determined by $X_0, ..., X_m$ may be written as a countable disjoint union of elementary events

$$E = \cup_{k=1}^{\infty} E_k$$

where $E_k$ is a set of elementary events. Then the desired identity (3.3.4) for $E$ follows by summing up the corresponding identities for $E_k$. □

So in reinforcement learning, a sequence of states and actions $(s_0, a_0, ..., s_t, a_t)$ respects the Markov property if the current state and action $(s_t, a_t)$ can summarize the past information such that all relevant information is kept to predict the reward $r_{t+1}$ and the next state $s_{t+1}$. If we take the example of a chess game, the state $s_t$ would be the current configuration of the board, and the action $a_t$ would be the move done at time step $t$. Even though a lot of information about the sequence has been lost, the state and action combination $(s_t, a_t)$ can be considered as Markovian because it summarizes everything relevant about the chess game. In other words, the future reward and the future state do not depend on all the past actions and states but only on the current state and action $(s_t, a_t)$.

Now that the Markov property has been defined (see Theorem 3.3.2), let us have a more general explanation of how it works in reinforcement learning with the following equation from Sutton and Barto [1998]:

$$p(s_{t+1} = s', r_{t+1} = r' | s_t, a_t) = p(s_{t+1} = s', r_{t+1} = r' | s_0, a_0, r_1, ..., s_{t-1}, a_{t-1}, r_t, s_t, a_t) \quad (3.3.5)$$

When a sequence of states and actions respect the Markov property, then it is possible to predict the next state, $s_{t+1}$, and the next expected reward, $r_{t+1}$, given the current state and action $(s_t, a_t)$. So, an agent can predict its state $s_{t+1}$ and reward $r_{t+1}$ by knowing only its current state $s_t$ and action $a_t$, as well as if the agent knew all the states, actions and rewards prior to $t + 1$. Therefore, the Markov property plays an important role in reinforcement learning, particularly in policy optimization.

Based on the Markov property (Theorem 3.3.2) and the work of Sutton and Barto [1998], we have that given any state $s_t \in S$ and action $a_t \in A(s_t)$, for $t = 1, 2, ..., T$ and $T$ being the horizon, the probability that the next state-reward $(s_{t+1}, r_{t+1})$ pair is equal to the $(s', r')$ pair is defined as:

$$p(s', r' | s, a) = p(s_{t+1} = s', r_{t+1} = r' | s_t = s, a_t = a) \quad (3.3.6)$$

and that thanks to expression (3.3.6), when one is at moment $t$, it is possible to compute the expected next reward for a state-action pair using the formula from Sutton and Barto [1998]:

$$\mathbb{E}[r_{t+1} | s_t = s, a_t = a] = \sum_{r' \in R} r' \sum_{s' \in S} p(s', r' | s, a) \quad (3.3.7)$$

where $R$ is the set of possible rewards. That formula can be generalize for $n = 1, 2, 3, ...$ as

$$\mathbb{E}[r_{t+n}|s_t = s, a_t = a] = \sum_{r' \in R} r' \sum_{s' \in S} p(s', r'|s, a) \tag{3.3.8}$$

To simplify writing we will refer to $r_{t+n}(s, a)$ as $r_{t+n}$. One can also compute the state-transition probabilities [Sutton and Barto, 1998]

$$p(s'|s, a) = \sum_{r' \in R} p(s', r'|s, a). \tag{3.3.9}$$

## 3.4 Value function

Most reinforcement learning problems require the estimation of value functions, which are used to estimate how "good" a given state, action, or state-action pair is for an agent when following a policy $\pi$.

The study of Markov Decision Processes has shown that we can use value functions to quantify how "good" a state, an action, or both are for an agent. Value functions can be obtained thanks to the conditional expected total rewards depending on the policy the agent is following. From Watkins [1989] and Dietterich [2000], we define the value function as follows:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s] = \mathbb{E}_\pi\Big[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}|s_t = s\Big] \tag{3.4.1}$$

where $\mathbb{E}_\pi[G_t|s_t = s]$ is the expected return, in a infinite horizon model, when being in state $s$ and following the policy $\pi$. The Bellman equation [Bellman, 1957] explains more clearly the value functions and what it computes. Equation (3.4.1) is the state-value function for policy $\pi$. The value function $v_\pi(s)$ gives the expected return of following policy $\pi$ starting from state $s$. For undiscounted rewards, one just needs to set $\gamma = 1$ in equation (3.4.1).

Let us also introduce the action-value or q-value function from Watkins [1989] and Watkins and Dayan [1992], which does the same as the value function but takes the action $a$ into consideration. Let

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a] = \mathbb{E}_\pi\Big[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}|s_t = s, a_t = a\Big] \tag{3.4.2}$$

where $q_\pi(s, a)$ gives the expected return of selecting action $a$ when starting from $s$ and then following the policy $\pi$.

The goal of an agent when facing a reinforcement learning problem is to find the optimal policy. State-value function and/or q-function are used to assess policies. For example, if an agent decides to follow a policy $\pi$ and keeps an average, for each state visited, of the

returns obtained after a state occurred, then the average will converge to $v_\pi(s)$, as the number of visits to that state converges to infinity [Sutton and Barto, 1998]. Similarly, if averages are maintained separately for each state-action pairs encountered, the averages will converge to $q_\pi(s, a)$.

### 3.4.1 Bellman equation

An important property of value functions is that they can be written recursively. This is thanks to Bellman's principle of optimality [Bellman, 1957], which states that:

**Principle of Optimality** *An optimal policy has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal policy about the state resulting from the first decisions.*

The principle of optimality leads to the concept of the "Bellman equation," which is a recursive relationship that expresses the value of a state or a decision in terms of the values of subsequent states or decisions. The Bellman equation is the foundation of dynamic programming algorithms like the value iteration algorithm for Markov decision processes and others.

Starting from equation (3.4.1), Lewis et al. [2012] proposed a backward recursion for the value function $v_\pi(s)$. So, thanks to the Chapman-Kolmogorov identity [Lockhart, 2006], and Theorem (3.3.2) (the Markov Property), the backward recursion of $v_\pi(s)$ is

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|s_t = s] \\ &= \mathbb{E}_\pi[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}|s_t = s] \\ &= \mathbb{E}_\pi\Big[r_{t+1} + \gamma \sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i+1}|s_t = s\Big] \\ &= \sum_a \pi(a|s) \sum_{r'} \sum_{s'} p(s', r'|s, a)\Big[r' + \gamma \mathbb{E}_\pi[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i+1}|s_{t+1} = s']\Big] \\ &= \sum_a \pi(a|s) \sum_{r',s'} p(s', r'|s, a)\Big[r' + \gamma v_\pi(s')]\Big]. \end{aligned} \tag{3.4.3}$$

This is the Bellman equation for $v_\pi(s)$ from Bellman [1957], where $s \in S$, $r' \in R$, $\pi(a|s)$ is the probability of selecting action $a$ when the agent is in state $s$ and $v_\pi(s')$ is the state-value function of the next state $s'$. The Bellman equation for $v_\pi(s)$ (equation 3.4.3) is important for finding the optimal policy $\pi^*$.

### 3.4.2 Optimal value function

As mentioned above, the goal of an agent facing a reinforcement learning problem is to find the optimal policy $\pi^*$. A policy $\pi$ is chosen over a policy $\pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for

all $s \in S$, the policy with the largest expected return is the optimal policy. We defined that policy as

$$\pi^* := \arg\max_{\pi} \ v_\pi(s), \ \forall s \in S. \tag{3.4.4}$$

It is also possible to have multiple optimal policies. When it happens, the policies with the same state-value functions share the optimal state-value function. That optimal state-value function is defined as follows [Watkins and Dayan, 1992]:

$$v_*(s) = \max_{\pi} \ v_\pi(s), \ \forall s \in S. \tag{3.4.5}$$

As for the state-value function, optimal policies also share the same $q$-function, and that function is defined as follows [Watkins, 1989]:

$$q_*(s,a) = \max_{\pi} \ q_\pi(s,a), \ \forall s \in S \ and \ a \in A(s). \tag{3.4.6}$$

Sutton and Barto [1998] state that since the action-value function gives the expected return of taking an action $a$ in state $s$ when following a policy $\pi$, it is possible to write $q_*(s,a)$ as a function of $v_*(s)$. That is,

$$q_*(s,a) = \mathbb{E}[r_{t+1} + \gamma v_*(s_{t+1}) | s_t = s, a_t = a], \tag{3.4.7}$$

where $v_*(s)$ refers to the value function for an optimal policy. The value function $v_*$ must meet the self-consistency condition given by the Bellman equation (equation 3.4.3). This condition, known as the Bellman optimality equation, can be rewritten without referencing any policy. Sutton and Barto [1998] has shown that we can define it as follows:

$$\begin{aligned}
v_*(s) &= \max_{a \in A(s)} \ q_{\pi^*}(s,a) \\
&= \max_{a \in A(s)} \ \mathbb{E}_{\pi^*}[G_t | s_t = s, a_t = a] \\
&= \max_{a \in A(s)} \ \mathbb{E}_{\pi^*}\Big[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i} | s_t = s, a_t = a\Big] \\
&= \max_{a \in A(s)} \ \mathbb{E}_{\pi^*}\Big[r_{t+1} + \gamma \sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i+1} | s_t = s, a_t = a\Big] \\
&= \max_{a \in A(s)} \ \mathbb{E}\Big[r_{t+1} + \gamma v_*(s_{t+1}) | s_t = s, a_t = a\Big] \\
&= \max_{a \in A(s)} \ \sum_{r',s'} p(s', r' | s, a)\Big[r' + \gamma v_*(s')\Big]
\end{aligned} \tag{3.4.8}$$

The last two lines of equation 3.4.8 represent two types of the Bellman optimality equation for $v_*(s)$. The idea of the Bellman optimality equation is that it describes how the value of a state under an optimal policy is equal to the expected return of the best action from that state.

There is also a Bellman optimality equation for $q_*(s,a)$, defined as follows.

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[r_{t+1} + \gamma \max_{a'} \ q_*(s_{t+1}, a') | s_t = s, a_t = a] \\ &= \sum_{r',s'} p(s', r'|s,a)\Big[r' + \gamma \max_{a'} \ q_*(s', a')\Big]. \end{aligned} \tag{3.4.9}$$

When $v_*(s) \ \forall \ s \in S$, is available, find the optimal policy becomes easy. Indeed, for each state $s \in S$, any policy that give a nonzero probability to actions that maximize $v_*(s)$, is an optimal policy. When $q_*(s,a) \ \forall \ s \in S$ and $\forall \ a \in A(s)$, is available, it is even easier to find the optimal policy. One must choose the action that maximizes $q_*(s,a)$ for any state $s$. Thanks to $q_*(s,a)$, the optimal expected long-term return is immediately available for all possible pairs $(s,a)$. Having $q_*(s,a)$ allows to take the optimal action without knowing information about the possible next states and their values.

## 3.5 Gridworld Example

Now that the concept of the value function and optimal value function has been covered, let us show how one can use value functions to find optimal value functions and an optimal policy for a given decision problem. The example that will be covered is an agent facing a GridWorld and has to find the shortest path to a square that is a terminal state. So,

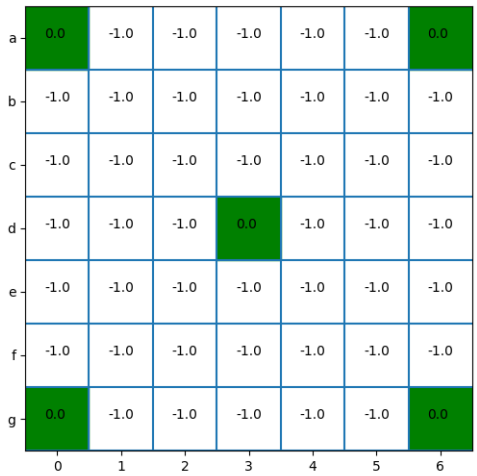



Figure 3.5.1: Gridworld where on the white square the agent receives a penalty of -1 and on the green sqaure no penalty is given. The terminal states are the squares in green.

suppose an agent is facing the GridWorld in figure 3.5.1. One can see that the reward

received by the agent is -1 when he is on a white square and 0 when he is on a green square. The agent's goal is to find the closest green square on the grid when he is on a white square. Thanks to the Bellman equation, it is possible to compute the value of a policy and find an optimal path regardless of where the agent starts.

We assume that the agent has no prior knowledge on the environment. Therefore a common technique is to set the value of all states to 0, as in figure 3.5.2. Starting from that initial value function, one can compute the value function of a policy by applying the Bellman equation several times and replacing the initial values of the value function, with those calculated using the Bellman equation.

In that given environment, when an agent is at a position on the grid, it can choose to move up, down, left or right with equal probability. Let us say the agent is following the policy in figure 3.5.3 with the associate value function from figure 3.5.3. Now suppose an agent is in state (c,1), then the policy of the agent is to go up and the value of state (c,1) is -1.9. Let us see how this value is computed using the Bellman equation (equation 3.4.3). The policy on figure 3.5.3 only says to go up when in state (c,1), and the possible reward when following that policy is certain and is -1. Then, if we set the discount factor to 0.9 and call that policy $\pi_1$, we have that

$$\begin{aligned} v_{\pi_1}((c,1)) &= \sum_a \pi_1(a|(c,1)) \sum_{r',s'} p(s',r'|s,a)\Big[r' + \gamma v_{\pi_1}(s')\Big] \\ &= 1 * 1 * (-1 + 0.9 * -1) \\ &= -1.9. \end{aligned}$$

For the state (b,1) we have

$$\begin{aligned} v_{\pi_1}((b,1)) &= \sum_a \pi_1(a|(b,1)) \sum_{r',s'} p(s',r'|s,a)\Big[r' + \gamma v_{\pi_1}(s')\Big] \\ &= 1 * 1 * (-1 + 0.9 * 0) \\ &= -1. \end{aligned}$$

For the state (a,1) we have

$$\begin{aligned} v_{\pi_1}((a,1)) &= \sum_a \pi_1(a|(a,1)) \sum_{r',s'} p(s',r'|s,a)\Big[r' + \gamma v_{\pi_1}(s')\Big] \\ &= 1 * 1 * (0 + 0.9 * 0) \\ &= 0. \end{aligned}$$

For the state (a,0), we have that

$$v_{\pi_1}((a,0)) = 0,$$

since it is a terminal state. This small example illustrated how Bellman equation is used to calculate the value function of a policy.

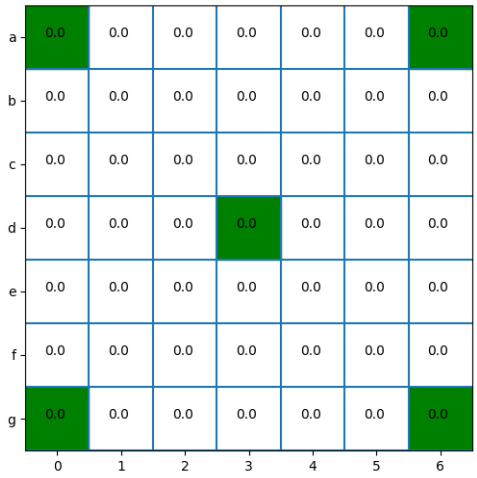



Figure 3.5.2: Initial value function. Since the agent has no prior knowledge on the environment, the value of all states is set to 0

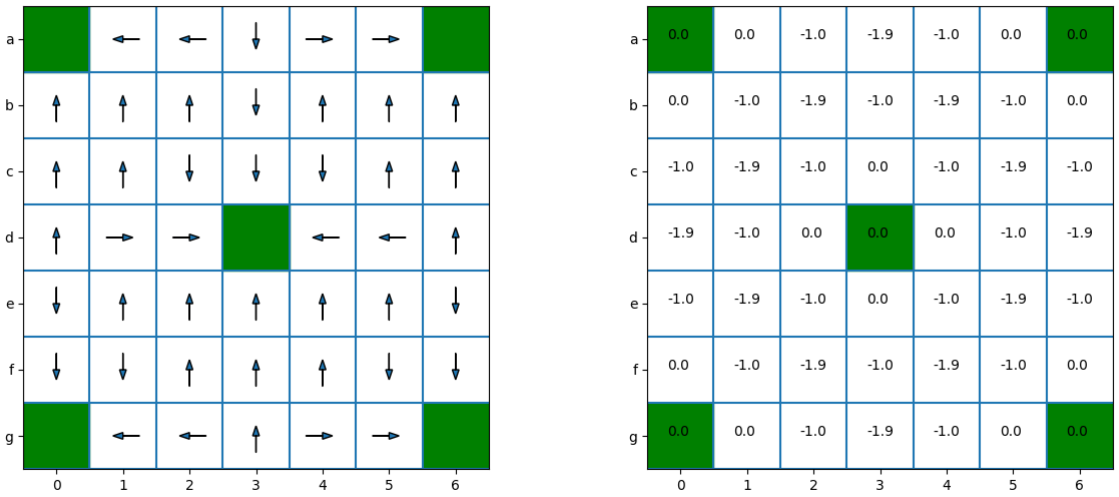



Figure 3.5.3: On the left we can see the policy of the agent. On the right there is the value function associated to the policy computed with Bellman equation

# Chapter 4

# Dynamic Programming Methods In Reinforcement Learning

Dynamic programming (DP) is an optimization technique that solves algorithmic problems by breaking them into subproblem, solves the subproblem and then put them together to find the solution to the original problem. These algorithms require that the Reinforcement Learning problem has a finite horizon, satisfies the Markov property, and that the agent has complete knowledge of the environment and perfect control over it. The state and action spaces must be finite. If these assumptions are not met, DP algorithms may not be applicable, and other Reinforcement Learning techniques such as Monte Carlo methods may be more appropriate.

In this chapter we will see how the Bellman equation can be use iteratively to compute the optimal value function and finding an optimal policy. It will start with an introduction to dynamic programming and how it works. Then, the chapter will go over on how dynamic programming is used to evaluate policies. Finally, two of the most popular DP algorithms will be presented. It will explain how they work and illustrate how they can be applied on a use case.

## 4.1 Introduction to Dynamic Programming

A great advantage of Dynamic Programming is that it does not work more than necessary. When subproblems overlap, which happens when subproblems share the same sub-subproblems, it solves each sub-subproblem only once and then stores its answer. This avoids computing the answer to a sub-subproblem that has already been seen before. In general, Dynamic Programming algorithms aim to find an optimal solution for optimization problems. Note that these kinds of algorithms try to find an optimal solution and not the optimal solution because such problems can have several optimal solutions. According to Cormen et al. [2009], one has to follow a four-step sequence when developing a DP algorithm, and these steps are the following:

1. **Characterize the structure of an optimal solution**: Identify the problem that needs to be solved and determine the optimal solution to that problem. This involves identifying the key parameters that will be used to measure the quality of different solutions.

2. **Define the value of an optimal solution**: Define a recursive function that calculates the value of an optimal solution in terms of the values of smaller subproblems. This involves breaking down the problem into subproblems and defining how the solution to the original problem can be obtained by combining the solutions to the subproblems.

3. **Compute the value of an optimal solution**: Use the recursive function defined in step 2 to find the optimal value function for all subproblems of interest.

4. **Construct an optimal solution**: With the optimal value functions from step 3, combine them to obtain an optimal policy for the original problem.

Cormen et al. [2009] also states that steps 1 to 3 constitute the foundation of a dynamic programming response to a problem. If only the value of an optimal solution is needed, the fourth step can be omitted.

To apply a Dynamic Programming solution to an optimization problem, that problem must have two major elements: **optimal substructure** and **overlapping subproblems**.

Let us discuss **optimal substructure**. As mentioned before, the first step to implementing a dynamic programming solution is to characterize the structure of an optimal solution. A problem has an optimal substructure when an optimal solution to the problem has optimal solutions to subproblems. According to Lew and Mauch [2006], the validity of this principle relies on the idea that if a policy has a subpolicy that is not optimal, replacing the subpolicy with an optimal subpolicy would improve the original policy. The elements of optimal substructure come from Bellman's principle of Optimality (Section 3.4.1). The key idea here is that the optimal solution to the original problem can be obtained by recursively finding optimal solutions to smaller subproblems. This allows solving complex problems by breaking them down into simpler subproblems and solving them individually which is more efficient way. Cormen et al. [2009] explains that from one problem to another, the optimal substructure is differentiated by two factors:

1. the number of subproblems that are required to be solved to construct an optimal solution to the original problem;

2. the number of possible subproblems that can be used to construct an optimal solution to the original problem.

The execution time of a dynamic programming algorithm depends on the total number of subproblems and the number of possible inputs for each subproblem. Therefore, in general, it is a good idea to start with a simple and efficient implementation of the subproblems in

a dynamic programming algorithm and then expand it as necessary to handle larger input sizes or more complex subproblems.

Implementing a solution using DP techniques should be simpler because instead of solving one big problem, DP algorithms can separately solve smaller and easier problems. However when breaking down the original problem into subproblems, it is important to be careful with the subproblem space because if it becomes too large and too complex it can affect the efficiency of DP algorithms. So when dividing the problem one should balance the trade-off between subproblem space and complexity, because if it is not carefully balanced it can lead to impractical DP algorithms. In general the choice of the subproblem space depends on the problem to solve, the computational resources at disposal and a thorough analysis.

Now, let us discuss the second major element, which is **overlapping subproblems**. When breaking down a problem into smaller problems, if at least one subproblem is counted several times in the set of subproblems, then there are overlapping subproblems. To deal with that, instead of solving the same subproblem multiple times, DP programming solves overlapping subproblems once and stores the solution. Then if it has to solve a subproblem that has already been treated, it can just look up for the solution. This allows algorithms to reach a final solution faster. To apply DP algorithms, the subproblems must be independent, which means that the solution of a suproblem does not depend on the solution of another subproblem. This allows to treat the subproblems in any order and to combine their solution to solve the general problem.

For overlapping subproblems, DP programming algorithms also use memoization. The idea of memoization is that when an expensive function is called with the same inputs multiple times to find the solution to subproblems, instead of recalculating the results each time, the solution is stored. So, if the function is called with a set of inputs that have already been processed, it can simply look up the answer instead of calculating it.

The Fibonacci sequence is a popular example of a problem that can be solved using dynamic programming. The Fibonacci sequence is defined as follows:

- $fib(0) = 0$
- $fib(1) = 1$
- $fib(2) = 1$
- $fib(3) = 2$
- $fib(4) = 3$
- $fib(n) = fib(n-1) + fib(n-2)$, for $n > 1$

where $fib(n)$ is a function that computes the $n$th term in the Fibonacci sequence. In this sequence, each term is the sum of the previous two terms. We can use memoization to

solve this problem using dynamic programming to avoid computing the same subproblems multiple times. It works as follows:

1. Start with an empty table to store the solutions of subproblems.
2. When $fib(n)$ is called, if the solution is already stored return it.
3. If not, compute the solution using $fib(n) = fib(n-1) + fib(n-2)$ and store it.

This ensures that it only computes each subproblem once and then reuses the solution whenever needed. With the Fibonacci sequence, the optimal substructure is apparent because the $n$th term in the sequence can be constructed from the $(n-1)$th and $(n-2)$th terms, both of which are subproblems of size $(n-1)$ and $(n-2)$ respectively. Moreover, the concept of overlapping subproblems is also apparent. Indeed, to compute $fib(5)$, it is needed to compute $fib(4)$ and $fib(3)$. But to compute $fib(4)$, it is needed to compute $fib(3)$ and $fib(2)$, which is already solved when $fib(3)$ was computed.

## 4.2 Policy Evaluation

Before introducing algorithms that provide optimal solutions to decision-making problems, let us first look at a policy evaluation technique. The method that will be presented is iterative policy evaluation, which is very popular and used by many algorithms.

The algorithm starts with initial estimates, $v_{\pi;0}(s)$, of the value function $v_\pi(s)\ \forall\ s \in S$, given a policy $\pi$ to be evaluated, and iteratively updates the estimates, $v_{\pi;n}(s)$ for $n = 1, 2, ..., N$, where $N$ is the total number of iterations, using the Bellman equation (equation 3.4.3). At each iteration, the algorithm updates the estimate of $v_{\pi;n+1}(s)$ for each state $s \in S$ by computing a new estimate based on the estimate of $v_{\pi;n}(s)$, the reward received $r' \in R$ when following the policy, and the values of the possible successor states $s' \in S$. This update is performed using the following equation:

$$\begin{aligned} v_{\pi;n+1}(s) &= \mathbb{E}_\pi[r' + \gamma v_{\pi;n}(s') | s_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r'} p(s', r'|s, a)\Big[r' + \gamma v_{\pi;n}(s')\Big]. \end{aligned} \tag{4.2.1}$$

The algorithm applies this update until the estimates of $v_{\pi;n+1}(s)\ \forall\ s \in S$, converge numerically. Sutton and Barto [1998] presented the iterative policy evaluation using the following pseudo code:

**Algorithm 2** Pseudo code to implement Iterative Policy Evaluation for reinforcement learning and decision making problems

**Input** :

- $\pi$, Policy to evaluate
- $v(s) = 0, \forall s \in S$

1. Repeat
2. $\quad \Delta \leftarrow 0$
3. $\quad$ For each $s \in S$:
4. $\quad\quad v = v(s)$
5. $\quad\quad v(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_k(s')\Big]$
6. $\quad\quad \Delta = \max(\Delta, |v - v(s)|)$
7. Until $\Delta < \theta$ (a small positive number)
8. Return $v \approx v_\pi$

Under certain conditions, that will be developed, the algorithm 2 is guaranteed to converge to the true value function.

**Definition 4.2.1** (*Distance Function* [Hunter and Nachtergaele, 2001])**.** Let us assume that $W$ is a non-empty set. A distance function on $W$ is a function

$$d : W \times W \to \mathbb{R}, \tag{4.2.2}$$

with the following properties:

- $d(w, y) \geq 0 \ \forall \ w, y \in W$, and $d(w, y) = 0$, iff $w = y$;
- $d(w, y) = d(y, w), \ \forall \ w, y \ \in W$;
- $d(w, y) \leq d(w, z) + d(z, y), \ \forall \ w, y, z \in W$.

We define a metric space $(W, d)$ as the couple formed by $W$ with a metric $d$.

To give an example, the set of real number $\mathbb{R}$ with the distance function $d(x, y) = |x - y|$ is metric space. So in the case of value functions, we will define $d(v_{\pi;n}(s), v_{\pi;n+1}(s)) = |v_{\pi;n}(s) - v_{\pi;n+1}(s)|$. Now let us define what a contraction mapping is.

**Definition 4.2.2** (*Contraction Mapping*)**.** Let $(W, d)$ be a metric space. A mapping $T : W \to W$ is a *contraction mapping*, or *contraction*, if there exists a constant $c$, with $0 \leq c < 1$, such that

$$d(T(w), T(y)) \leq cd(w, y) \ \forall \ w, y \in W \tag{4.2.3}$$

where $d(w, y) = |w - y|$

The two conditions that must be met so that the contraction property can be applied are:

- the state space is finite.
- the policy $\pi$ is a fixed (deterministic) policy.

So, when these two conditions are met, we can apply the contraction property from the contraction mapping theorem.

**Theorem 4.2.1** (Hunter and Nachtergaele [2001], Contraction Mapping Theorem)**.** *If $T : W \to W$ is a contraction mapping on a complete metric space $(W, d)$, then there is exactly one solution $w \in W$ for $T(w) = w$.*

*Proof.* Let $w_0$ be any point in $W$. We define a sequence $(w_n)$ in $W$ by

$$w_{n+1} = T(w_n) \; for \; n \geq 0.$$

To simplify notation, the parentheses around the argument of a map are often omitted. The $n$th iterate of $T$ is denoted by $T^n$, so that $T^n w_0$.

First, we show that $(w_n)$ is a Cauchy sequence. If $n \geq m \geq 1$, then from 4.2.3 and the triangle inequality, we have:

$$\begin{aligned} d(w_n, w_m) &= d(T^n w_0, T^m w_0) \\ &\leq c^m d(T^{n-m} w_0, w_0) \\ &\leq c^m \Big[ d(T^{n-m} w_0, T^{n-m-1} w_0) + d(T^{n-m-1} w_0, T^{n-m-2} w_0) + ... + d(T w_0, w_0) \Big] \\ &\leq c^m \Big[ \sum_{k=0}^{n-m-1} c^k \Big] d(w_1, w_0) \\ &\leq c^m \Big[ \sum_{k=0}^{\infty} c^k \Big] d(w_1, w_0) \\ &= \Big( \frac{c^m}{1-c} \Big) d(w_1, w_0), \end{aligned}$$

this implies that $(w_n)$ is Cauchy. Since $W$ is complete, $(w_n)$ converges to a limit $w \in W$. The fact that the limit $w$ is a fixed point of $T$ follows from the continuity of $T$:

$$Tw = T \lim_{n \to \infty} w_n = \lim_{n \to \infty} T w_n = \lim_{n \to \infty} w_{n+1} = w$$

Finally, if $w$ and $y$ are two fixed points then

$$0 \leq d(w, y) = d(Tw, Ty) \leq c d(w, y).$$

Since $c < 1$, we have $d(w, y) = 0$, so $w = y$ and the fixed point is unique □

So given the definition 4.2.2 and theorem 4.2.1 if we set

$$v_{\pi;n+1}(s) = T(v_{\pi;n}(s)) = \sum_a \pi(a|s) \sum_{s',r'} p(s', r'|s, a)\Big[r' + \gamma v_{\pi,n}(s')\Big]$$

where $T$ is called the Bellman operator and maps a value function $v_\pi$ to a new value function $T(v_\pi)$. If the state space is finite and the policy $\pi$ is a fixed policy, then applying the Bellman operator iteratively in policy evaluation leads to a contraction mapping, and we have a unique solution for the value function $v_\pi(s)$.

In practice, the algorithm will converge to an approximation of the true value function after a finite number of iterations, but it will not converge to the exact value function. Sutton and Barto [1998] presented a popular stopping condition for the iterative policy evaluation algorithm, which is to check whether the maximum change in the value function between two consecutive iterations is less than a pre-defined threshold. The idea is to check after each iteration if $|v_{\pi;n+1}(s) - v_{\pi;n}(s)| < \theta$ for all states $s$, where $\theta > 0$, but small enough.

## 4.3 The Policy Iteration Algorithm

Now that Policy Evaluation has been introduced, we will present a popular algorithm in reinforcement learning, which is the policy iteration algorithm.

Policy Iteration is a DP algorithm that finds an optimal policy for MDPs. The algorithm is separated in two steps: policy evaluation and policy improvement. For the policy evaluation step, the algorithm simply applies the iterative policy evaluation algorithm (algorithm 2) to evaluate a policy. For the policy improvement step, the algorithm constructs a new policy by selecting the action with the highest $q$-value ($q$-function) for each state based on the new state values from the evaluation step. Suppose one already has $v_\pi(s)$ and wants to know if it is better to follow policy $\pi$ or modify the policy for a given state $s$. One way to do it is to consider selecting $a \neq \pi(s)$ when in state $s$ and follow policy $\pi$ for the states that come after $s$. To do that, one can use the $q$-function.

$$q_\pi(s, a) = \mathbb{E}_\pi[r_{t+1} + \gamma v_\pi(s_{t+1})|s_t = s, a_t = a]. \tag{4.3.1}$$

If $q_\pi(s, a)$ is greater than $v_\pi(s)$, then it is better to choose action $a$ when in state $s$, and one should modify its policy $\pi$ so that when the agent is in state $s$, it selects action $a$, making the new policy better than the original policy $\pi$. This is why the policy improvement step uses $q$-values. This step is based on a fundamental theorem known as the Policy Improvement Theorem.

**Theorem 4.3.1** (Sutton and Barto [1998], Policy Improvement Theorem)**.** *Let $\pi$ and $\bar{\pi}$ be two deterministic policies where $\pi$ is the original policy and $\bar{\pi}$ is the modified policy that is identical to $\pi$ except that $\bar{\pi}(s) = a \neq \pi(s)$ such that $v_\pi(s) \leq q_\pi(s, \bar{\pi}(s))$ for all $s \in S$, then*

*$v_\pi(s) \leq v_{\bar{\pi}}(s)$ for all $s \in S$. If $v_\pi(s) < q_\pi(s, \bar{\pi}(s))$ then the modified policy $\bar{\pi}$ is better than original $\pi$.*

*Proof.* Let us start with the assumption $v_\pi(s) \leq q_\pi(s, \bar{\pi}(s))$ for all $s \in S$ and use equation (4.3.1). Then it is possible to show that

$$\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \bar{\pi}(s)) \\
&= \mathbb{E}_{\bar{\pi}}[r_{t+1} + \gamma v_\pi(s_{t+1}) | s_t = s] \\
&\leq \mathbb{E}_{\bar{\pi}}[r_{t+1} + \gamma q_\pi(s_{t+1}, \bar{\pi}(s_{t+1})) | s_t = s] \\
&= \mathbb{E}_{\bar{\pi}}\Big[r_{t+1} + \gamma\, \mathbb{E}_{\bar{\pi}}[r_{t+2} + \gamma v_\pi(s_{t+2}) | s_t = s]\Big] \\
&= \mathbb{E}_{\bar{\pi}}\Big[r_{t+1} + \gamma r_{t+2} + \gamma^2 v_\pi(s_{t+2}) | s_t = s\Big] \\
&\leq \mathbb{E}_{\bar{\pi}}\Big[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 v_\pi(s_{t+3}) | s_t = s\Big] \\
&... \\
&\leq \mathbb{E}_{\bar{\pi}}\Big[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + ... \, | s_t = s\Big] \\
&= v_{\bar{\pi}}(s).
\end{aligned}$$

□

As mentioned previously, the policy iteration algorithm has two major steps: the policy evaluation step and the policy improvement step. However, when implementing the algorithm, one has to add an initialization step, and the algorithm works as follows:

**Algorithm 3** Pseudo code to implement the Policy Iteration Algorithm for reinforcement learning and decision-making problems

1. **Initialization**: initialize $v(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$
2. **Policy Evaluation**: Run Algorithm 2 to find the value of the policy
3. **Policy Improvement**:
   For each $s \in S$:
   - $a \leftarrow \pi(s)$
   - $\pi(s) \leftarrow \arg\max_a q_\pi(s, a)$
   - if $a = \pi(s)$ $\forall$ $s \in S$ stop and return $v_\pi$ and $\pi$, else go to 2 and repeat

We can use the Close Enough Lemma from Williams and Baird III [1993], to show that policy iteration algorithm converges to the optimal policy. Before that, let us introduce $\delta(s)$, which Williams and Baird III [1993] defined as:

$$\delta(s) = \min_{\pi \in \Pi} \{v_{\pi^*}(s) - v_\pi(s)\} \backslash \{0\}. \tag{4.3.2}$$

It represents the smallest nonzero difference, for any state $s \in S$, between the optimal value function, $v_{\pi^*}(s)$, and the value function for any policy $\pi$, $v_\pi(s)$ for all $\pi \in \Pi$.

**Lemma 4.3.1** (Close Enough Lemma)**.** *Let $v, v'$ be two value functions generated from the policy iteration algorithm. Let $\pi$ be such that $v_\pi(s) = v_*(s)$ and let $\pi'$ satisfy $v'_{\pi'}(s) \geq v'_\pi(s)$. Then*

$$|v_\pi(s) - v'_\pi(s)| + |v'_{\pi'}(s) - v_{\pi'}(s)| < \delta(s)$$

*implies that $v_{\pi'}(s) = v_\pi(s)$.*

*Proof.*

$$\begin{aligned} 0 \leq v_\pi(s) - v_{\pi'}(s) &= v_\pi(s) - v'_\pi(s) + v'_\pi(s) - v_{\pi'}(s) \\ &\leq v_\pi(s) - v'_\pi(s) + v'_{\pi'}(s) - v_{\pi'}(s) \\ &\leq |v_\pi(s) - v'_\pi(s)| + |v'_{\pi'}(s) - v_{\pi'}(s)| \\ &< \delta(s). \end{aligned} \tag{4.3.3}$$

Since $\delta(s) = \min_{\pi \in \Pi}\{v_{\pi^*}(s) - v_\pi(s)\}\backslash\{0\}$, it follows that $v_\pi(s) - v_{\pi'}(s) = 0$. □

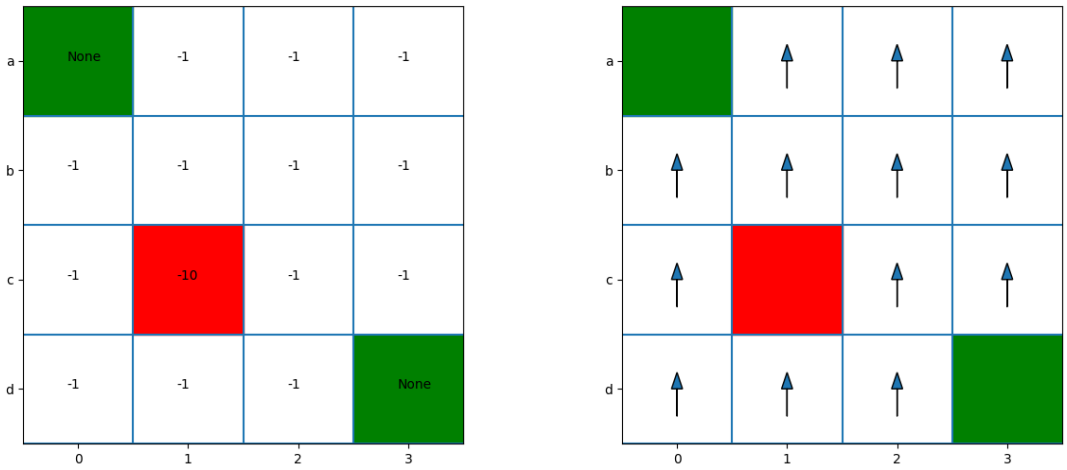
Theorem 6.4.6 from Putterman [1994] also states that policy iteration converges to the optimal value function. Section 2.2.3 from Bertsekas and Tsitsiklis [1996] states that policy iteration always terminates finitely and can be a good alternative to value iteration (Section 4.4) in some cases. Finally, Section 1.3.3 from Bertsekas [2007] also explains that the policy iteration algorithm finds an optimal solution in a finite number of iterations and states that this property is one of the main advantages of policy iteration compared to other dynamic programming algorithms.

### 4.3.1 Policy Iteration Algorithm Example

As for Section 3.5, we will use a GridWorld Example to show how policy iteration works. To illustrate the policy iteration algorithm, we use the gridworld environment in Figure 4.3.1 (a) and the initial policy in Figure 4.3.1 (b). Green squares represent terminal states, and the red square is an obstacle the agent must avoid. Given its starting state, the agent's goal is to find the shortest path to the closest terminal state. As one can see, the initial policy is to move up regardless of the agent's position, which is not optimal.

So, the policy iteration starts by evaluating the initial policy and then improves that policy. Then it repeats these two steps for new policies obtained from the improvement step until there is no more improvement, which means that an optimal solution has been found.

On Figure 4.3.2, one can see on the left, the policies obtained at each iteration from the algorithm. On the right, one can see the value functions. It appears that at iteration 3 and iteration 4 the results are the same because no more improvement is done by the algorithm and therefore it stops. The algorithm started from the initial policy and environment from figure 4.3.1. Then it applied algorithm 3 on the initial policy and this lead to an optimal policy that we can see at iteration 4 with the associated value function. During every




(a) (b)

Figure 4.3.1: On figure (a), one can see the environment of the gridworld that the agent will face. On figure (b), one can see the initial policy of the agent.

evaluation step in the process, we decided to set the initial value to zero for every state of the environment. This was done since it is a popular technique in reinforcement learning that has been shown to be very effective.

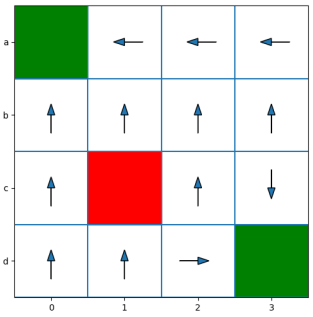
## 4.4 The Value Iteration Algorithm

Let us present another popular dynamic programming algorithm that is also based on Bellman's principle of optimality, which is called the Value Iteration algorithm.

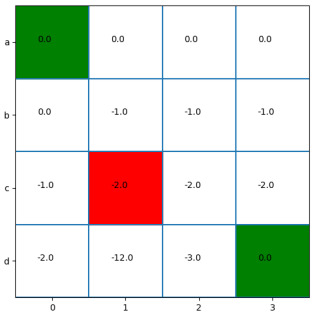
According to Sutton and Barto [1998], the Value Iteration algorithm is a particularly simple backup operation that combines the policy improvement and truncated policy evaluation steps. So, based on Putterman [1994] and Sutton and Barto [1998], Value iteration solves iteratively the following equations to find the optimal policy.

$$\begin{aligned} v_{n+1}(s) &= \max_a \ \mathbb{E}[r_{t+1} + \gamma v_n(s_{t+1}) | s_t = s, a_t = a] \\ &= \max_a \sum_{s',r'} p(s', r'|s, a)\Big[r' + \gamma v_n(s')\Big] \end{aligned} \tag{4.4.1}$$

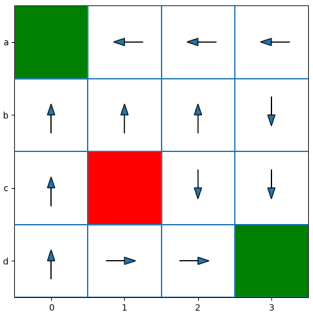
for all $s \in S$.
The algorithm uses uses equation (4.4.1) at each iteration to compute a new estimate of the value function for every $s \in S$. By repeatedly updating the value function using equation



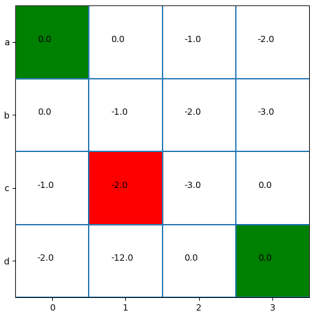

policy at iteration 1



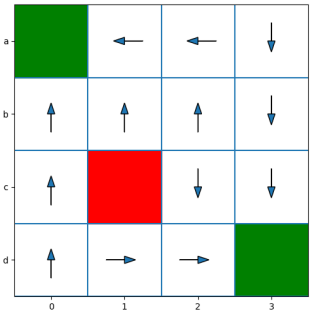

value at iteration 1



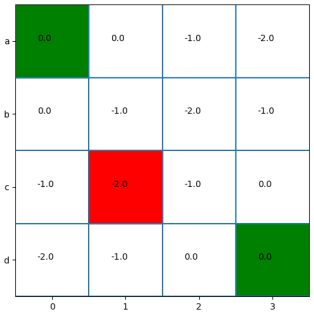

policy at iteration 2



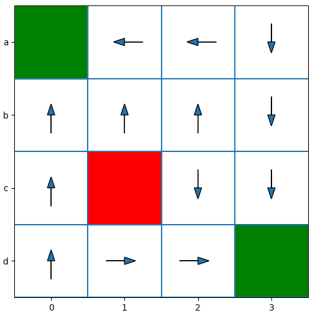

value at iteration 2



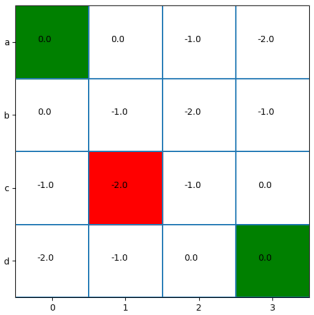

policy at iteration 3




value at iteration 3




policy at iteration 4




value at iteration 4

Figure 4.3.2: Policy Iteration algorithm evolution to an optimal policy on 4x4 grid

4.4.1, the algorithm converges to the optimal value function for the MDP and can be used to derive an optimal policy for the agent.
According to Putterman [1994] and Sutton and Barto [1998], one can implement the Value Iteration Algorithm as follows:

**Algorithm 4** Pseudo code to implement the Value Iteration algorithm for reinforcement learning and decision making problems

1. Initialize $v(s)$ arbitrarily $\forall\ s \in S$
2. Repeat
   - $\Delta \leftarrow 0$
   - For each $s \in S$:
     - $v \leftarrow v(s)$
     - $v(s) \leftarrow \max_a \sum_{s',r'} p(s', r'|s, a)\Big[r' + \gamma v(s')\Big]$
     - $\Delta \leftarrow max(\Delta, |v - v(s)|)$
3. Until $\Delta < \theta$
4. Return $\pi$ such that $\pi(s) = \max_a \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v(s')\Big]$

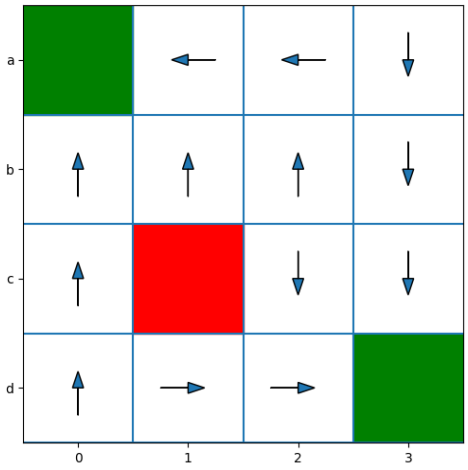
Using the Close Enough Lemma (Lemma 4.3.1), one can easily show that the value iteration algorithm converges to an optimal solution. Section 6.3.2 from Putterman [1994] and Section 1.3.1 from Bertsekas [2007] explain how by choosing an error bound threshold $\theta$ small enough, it is possible to ensure that the algorithm finds an optimal solution or near-optimal solution.

### 4.4.1 Value Iteration Algorithm example

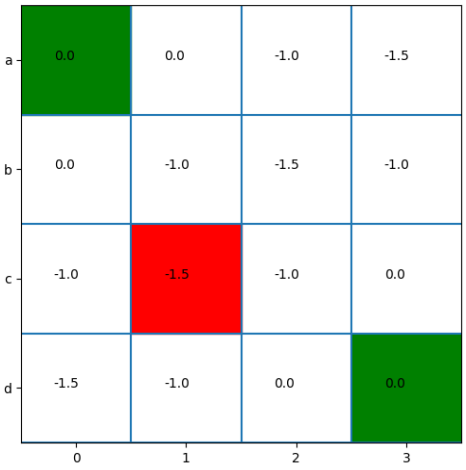
For the value iteration, we used the same example as the one used in section (4.3.1). So, for the Value Iteration illustration, we took the same gridworld environment as shown in figure 4.3.1 (a). Unlike the Policy Iteration, it does not require an initial policy. The algorithm only needs to have an initial state value function, which can be chosen arbitrarily or based on some knowledge that the agent has. In general, the initial state value function is set to 0 for all the states.

In the value iteration method, the algorithm iterates over each state $s \in S$ in the environment and selects the action that maximizes the value function $v(s)$ as shown in Algorithm 4. So one can see on the figure 4.4.1 that the results obtained at iteration 1 and iteration 2 are the same. Therefore in just one iteration the algorithm could find an optimal solution. This is because the gridworld used is small and not very complex. Moreover, value iteration often finds an optimal solution faster than policy iteration because

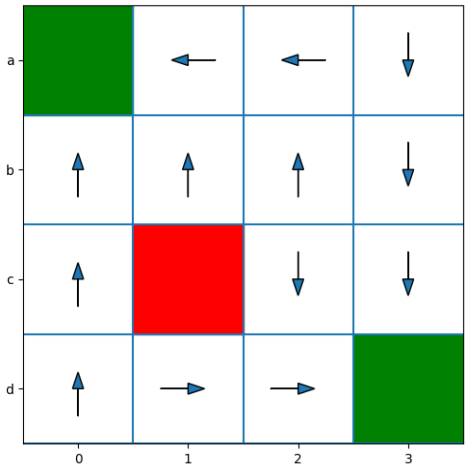
policy evaluation and policy improvement are done in the same loop with only one backup operation using equation (4.4.1).



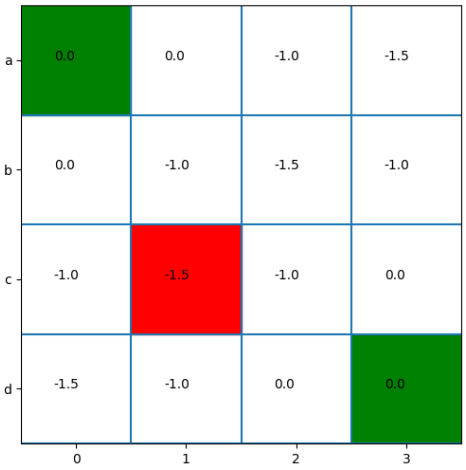

policy at iteration 1




value at iteration 1




policy at iteration 2




value at iteration 2

Figure 4.4.1: Value Iteration algorithm evolution to an optimal policy on a 4x4 grid

# Chapter 5

# Monte-Carlo Methods In Reinforcement Learning

Now that Dynamic Programming Methods, and particularly, two of the most popular Dynamic Programming algorithms, have been introduced, let us delve into other methods known as Monte Carlo Methods.

Dynamic programming methods, such as value iteration and policy iteration that we introduced previously, require knowledge of the transition dynamics and the reward function of the environment. In contrast, Monte Carlo methods estimate the value function using only the returns obtained from a sequence of actions. Monte Carlo methods do not require knowledge of the transition dynamics or of the reward function, as they only require experiences, which are samples of sequences of states, actions, and rewards from real or simulated exchanges of an agent with an environment. This is very useful because although these methods learn from real or simulated experiences and have no prior knowledge of the environment, they can still obtain an optimal solution.

## 5.1 Monte Carlo Policy Evaluation

An important aspect of Monte Carlo methods is that they solve reinforcement learning problems by using the sample's average returns. In that framework, a sample refers to a sequence of states, actions, and rewards starting from the beginning to the end of an episode. The general idea of Monte Carlo Methods is to generate as many samples as possible, given a policy $\pi$, and to obtain a state-value function estimate $v_\pi(s)$ for a state $s$ by averaging all the returns that followed that state. Since Monte Carlo methods require a complete episode of interactions with the environment before updating the value function estimate, it makes them particularly well-suited for episodic tasks, and the value function estimate can be updated at the end of each episode.

Singh and Sutton [1996] distinguished two specific algorithms in Monte Carlo methods, which they defined as:

- **Every-visit Monte Carlo**: where one estimates the value of a state as the average of the returns that came after all visits to that state.
- **First-visit Monte Carlo**: where one estimates the value of a state as the average of the returns that came after the first visit to that state. The first visit can be define as the first time during an episode that the state is visited.

The rest of the thesis will be focused on First visit Monte-Carlo. Depending on the problem an agent is facing, it can use equation (3.2.1) of the cumulative reward or equation (3.2.3) of the discounted cumulative reward to apply the First-visit Monte-Carlo algorithm. Based on the work of Singh and Sutton [1996] and the work of Bertsekas and Tsitsiklis [1996], we can define the cumulative reward of a sample $n$, $\forall\ n \in \mathbb{N}$, in the Monte-Carlo framework, as follows:

$$\begin{aligned} G_t^n &= r_{t+1}^n + r_{t+2}^n + r_{t+3}^n + ... + r_T^n \\ &= \sum_{i=1}^{T-t} r_{t+i}^n \end{aligned} \tag{5.1.1}$$

where $T$ is the horizon of the $nth$ sample and $r_t^n$ is the reward received at moment $t$ from the $nth$ sample. To have the discounted cumulative reward from sample $n$, one can use the following formula:

$$\begin{aligned} G_t^n &= r_{t+1}^n + \gamma r_{t+2}^n + \gamma^2 r_{t+3}^n + ... \\ &= \sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}^n \end{aligned} \tag{5.1.2}$$

where $0 < \gamma < 1$ is a discount factor.
Thus, we can define the first visit Monte-Carlo state value estimate of $v(s)$ after generating $n$ samples as:

$$v^n(s) = \frac{1}{n} \sum_{i=1}^{n} G_t^i(s),\ \forall s \in S \tag{5.1.3}$$

where $n$ is the number of samples generated and $G_t^n(s) = \sum_{i=1}^{T-t} r_{t+i}^n I(r_{t+i}^n = r_{t+i}^n(s))$. Now, let us have a deeper analysis of the Monte-Carlo method properties and start with the bias. First, let us define $v(s) = \mathbb{E}[G_t | s_t = s]$ as the true state value of a state $s$.

**Theorem 5.1.1** (Singh and Sutton [1996], Theorem 6)**.** *The First Visit Monte-Carlo is unbiased, i.e, Bias*$(v^n(s)) = v(s) - \mathbb{E}[v^n(s)] = 0\ \forall\ n > 0$.

*Proof.* The first-visit MC estimate is unbiased because the total reward on a sample path from the start state $s_0$ to the terminal state $s_T$ is, by definition, an unbiased estimator of the expected total reward across all such paths. Therefore, the average of the estimators obtained from $n$ independent sample paths is also unbiased. □

Through the work of Bertsekas and Tsitsiklis [1996] **example 5.5** and the definition of $v(s)$, we can show that $v^n(s)$ is unbiased as follows:

$$\begin{aligned} Bias(v^n(s)) &= v(s) - \mathbb{E}[v^n(s)|s_t = s] \\ &= v(s) - \mathbb{E}[\frac{1}{n}\sum_{i=1}^{n} G_t^i|s_t = s] \\ &= v(s) - \mathbb{E}[G_t^i|s_t = s] \\ &= 0. \end{aligned} \tag{5.1.4}$$

For the variance, we can use the following theorem from Singh and Sutton [1996].

**Theorem 5.1.2** (Singh and Sutton [1996], Theorem 8)**.** *The variance of first-visit MC is*

$$Var\Big[v^n(s)\Big] = \frac{1}{n} Var\Big[v^1(s)\Big]. \tag{5.1.5}$$

*Proof.* From Singh and Sutton [1996] (annex 5), we can prove it as follows

$$\begin{aligned} Var\Big[v^1(s)\Big] &= Var\Big[G_t^1(s)\Big] \\ &= \mathbb{E}\Big[(G_t^1(s))^2\Big] - \Big(\mathbb{E}\Big[G_t^1(s)\Big]\Big)^2. \end{aligned} \tag{5.1.6}$$

The first-visit Monte Carlo estimator after $n$ trials is the sample average of the $n$ independent trials (see equation 5.1.3). Hence,

$$Var\Big[v^n(s)\Big] = \frac{1}{n} Var\Big[v^1(s)\Big]. \tag{5.1.7}$$

□

From Theorem 5.1.2, we can conclude that $\lim_{n\to\infty} \mathrm{Var}[v^n(s)] = 0$. Since $v^n(s)$ is unbiased and its variance converges asymptotically to 0, the estimator is consistent. Moreover, we can also show that the Mean Square Error of the algorithm converges to 0 as $n \to \infty$. Let us recall the Mean Square Error formula for an estimator $\hat{\theta}$ defined as follows:

$$MSE(\hat{\theta}) = (Bias(\hat{\theta}))^2 + Var(\hat{\theta}) \tag{5.1.8}$$

replacing $\hat{\theta}$ by $v^n(s)$ in equation (5.1.8) we have that

$$\begin{aligned} MSE(v^n(s)) &= (Bias(v^n(s)))^2 + Var(v^n(s)) \\ &= \frac{Var[v^n(s)]}{n}. \end{aligned} \tag{5.1.9}$$

Hence, $MSE(v^n(s)) = 0$ as $n \to \infty$.

If one wants to evaluate a policy $\pi$, it is possible to use the first-visit Monte-Carlo method. Simply replace $v^n(s)$ with $v^n_\pi(s)$, and the first visit Monte-Carlo in equation (5.1.3) becomes:

$$v^n_\pi(s) = \frac{1}{n}\sum_{i=1}^{n} G^i_{\pi;t} \tag{5.1.10}$$

where the return $G^i_{\pi;t}$, is the cumulative return obtained when generating $n$ samples under the policy $\pi$. Moreover, all the properties demonstrated for $v^n(s)$ apply to $v^n_\pi(s)$. Therefore, we can say that $v^n_\pi(s)$ is consistent, and $MSE(v^n_\pi(s)) = 0$ as $n \to \infty$.

Sutton and Barto [1998] proposed to implement first-visit Monte-Carlo to evaluate a policy by using the pseudo code in algorithm 5. When implementing first-visit Monte-Carlo,

**Algorithm 5** Pseudo code to implement the First Visit Monte Carlo algorithm for reinforcement learning and decision-making problems

1. Initialize:
   - $\pi$, a policy to evaluate
   - $v$, an arbitrarily state-value
   - $Return(s)$, an empty return list
2. For 1 to $N$, where $N$ is the total iterations:
   - Generate an episode from $\pi$
   - for each state $s$ that occurs in the episode:
     - $G \leftarrow$ return after the first appearance $s$
     - Append $G$ to $Returns(s)$
     - $v(s) \leftarrow average(Returns(s))$

a popular stopping rule is to terminate the algorithm after a fixed number of episodes. Typically, a larger number of episodes will result in more accurate estimates since the estimator is consistent, but the computational cost will also increase.

## 5.2 The Monte Carlo Control Algorithm

Now that we know how Monte Carlo policy evaluation works, let us see how to use it for finding an optimal policy. The main idea when using Monte Carlo methods for finding an optimal policy is to proceed in the same way as for dynamic programming. Therefore, two main steps have to be taken for finding the optimal policy: policy evaluation and policy

improvement, as in policy iteration. When an initial policy is to be evaluated and improved, the value function is modified over the episode to approximate the value function of the current policy more precisely, and the policy is repeatedly improved according to the new value function that has a better approximation of the true value of the policy.

Monte Carlo control is a model-free algorithm because it does not require any explicit knowledge of the environment's dynamics. Instead, it estimates the optimal policy by simulating complete episodes of interaction with the environment and updating the estimates of the $q$-function based on the observed returns. In Monte Carlo control, the policy is improved by making it more greedy with respect to the estimated $q$-function. Moreover, we can say that, with Monte Carlo Control algorithm, for any $q$-function, the greedy policy is the one that chooses the action $a$ that maximizes the action value and can be defined as:

$$\pi(s) = \arg\max_a q(s, a) \ \forall s \in S \tag{5.2.1}$$

Thanks to equation (5.2.1), theorem 4.3.1 (Policy Improvement Theorem) can be applied and we have that for any policies $\pi_t$ and $\pi_{t+1}$, where policy $\pi_t$ is the original policy and $\pi_{t+1}$ is the modified policy using the Policy Improvement Theorem (theorem 4.3.1) we have that

$$\begin{aligned} q_{\pi_t}(s, \pi_{t+1}(s)) &= q_{\pi_t}(s, \arg\max_a q_{\pi_{t+1}}(s, a)) \\ &= \max_a q_{\pi_{t+1}}(s, a) \\ &\geq q_{\pi_t}(s, \pi_t(s, a)) \\ &= v_{\pi_t}(s) \ \forall s \in S. \end{aligned} \tag{5.2.2}$$

From Theorem 4.3.1 (Policy Improvement Theorem), we have that each $\pi_{t+1}$ is uniformly better than $\pi_t$, or as good as $\pi_t$, and in that last case, they both are optimal policies.

With Monte Carlo methods, as for policy iteration, a common approach for finding the optimal solution is to alternate between the evaluation step and the improvement step, episode by episode. So first an episode is generated, then using the feedback from that episode, the value function is updated. This can be considered as the evaluation step. Then once the value function is updated, the improvement step can be done to evaluate if modifying the current policy can lead to better results. If yes, the policy is improved, and if not then an optimal solution has been found. A popular algorithm in reinforcement learning that applies that approach is Monte Carlo ES, which stands for Monte Carlo with Exploring Starts.

Monte Carlo ES works as we just explained but with one more assumption which is the exploring start. This assumes that for all episodes generated with Monte Carlo ES, the state-action pair that will start the episode is randomly chosen amongst any state-action pair possible in the environment. This assumption is important because it ensures that every state-action pair is considered by the algorithm even those that may not be considered

by the current policy. The action value $q_\pi(s, a)$ for all possible state-action pairs $(s, a)$ is updated by averaging the returns observed after the pair $(s, a)$ has occurred, over all the simulated episodes. One can implement Monte Carlo ES as explained in algorithm 6.

**Algorithm 6** Pseudo code to implement the Monte Carlo ES algorithm for reinforcement learning and decision-making problems

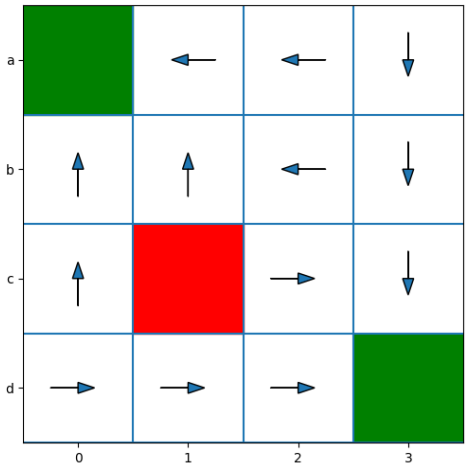
1. Initialize $\forall s \in S, a \in A(s)$:
   - $q(s, a) \leftarrow$ arbitrary
   - $\pi(s) \leftarrow$ arbitrary
   - $Returns(s, a) \leftarrow$ empty list
2. For 1 to $N$, where $N$ is the total iterations:
   - choose $s_0 \in S$ and $a_0 \in A(s_0)$ such that all pairs have probability $> 0$
   - generate an episode from $\pi$ that starts with state $s_0$ and action $a_0$
   - for each state-action pair $(s, a)$ that occurs in the episode:
     - $G \leftarrow$ return after the first appearance $(s, a)$
     - Append $G$ to $Returns(s, a)$
     - $q(s, a) \leftarrow average(Returns(s, a))$
   - for each $s$ that occurs in the episode:
     - $\pi(s) \leftarrow \arg\max_a \ q(s, a)$

## 5.3 The Monte Carlo ES Algorithm Example

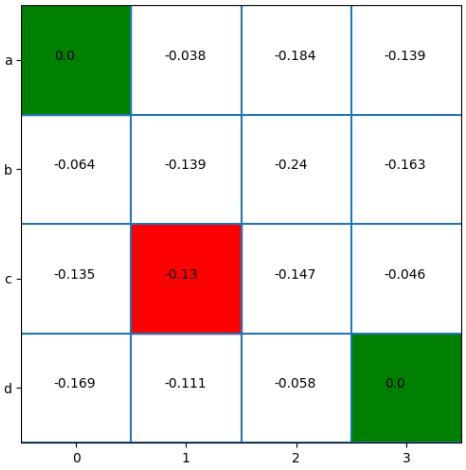
To illustrate how the algorithm works, we took the same example as for Policy Iteration and Value Iteration. As such, we will apply Monte-Carlo ES on the same gridworld environment as we have in figure 4.3.1 (a), and we took the same initial policy as in figure 4.3.1 (b).

For the example, we used algorithm 6 (Monte-Carlo with Exploring Start) with thousand simulations to find an optimal solution. For the first simulation we have started from the same initial policy in figure 4.3.1 (b). Then we applied algorithm 6 to find an optimal policy and get the value functions of that policy. For example, to obtain the value function for the state (b,2), we simply calculated the average return that has followed the state-action $(s, a)$ proposed by the updated policy over the thousands simulations executed. From that same state, the action to go up was taken because on all actions possible from that state, it was the one that maximized the $q$-function using Monte Carlo ES.

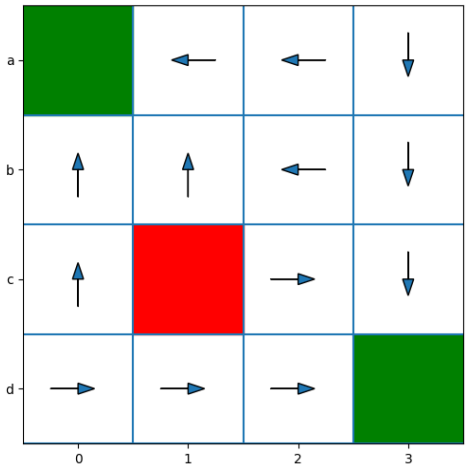
Finally on figure 5.3.1 we can see that the Monte Carlo ES algorithm also finds an optimal policy quickly, in only two iterations.



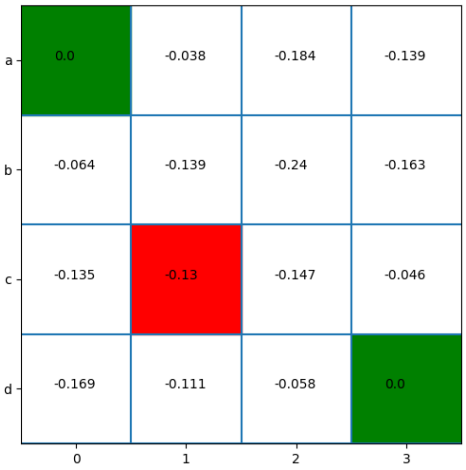

Monte Carlo ES at iteration 1




Monte Carlo ES at iteration 1




Monte Carlo ES at iteration 2




Monte Carlo ES at iteration 2

Figure 5.3.1: Monte Carlo ES algorithm evolution to optimal policy on the 4x4 grid

# Chapter 6

# Application of Reinforcement Learning

Now that dynamic programming algorithms and Monte Carlo methods have been presented, let us examine their performance on a real use case. We will apply the different algorithms presented on the famous board game Tic-tac-toe.

First, we will introduce the game of Tic-tac-toe and its rules. Then, we will apply Policy Iteration, Value Iteration, and Monte-Carlo ES to analyze and compare the optimal solutions proposed by the three different algorithms. To evaluate the optimal policies obtained, we will use game simulations. The policies will compete against a random player in each simulation, and we will analyze the results obtained from these games. Since Tic-tac-toe is a solved game with a controllable set of possible states, we expect that the policies obtain draws for worst-case scenarios. We will also see how the three different policies perform when they play against each other.

To train the models, we will use game simulations with simulated data. One of reinforcement learning's most significant advantages is that it does not require historical data to train a model. Indeed, if the environment is correctly implemented, simulations can make it possible to find an optimal policy. Therefore, one of the most significant challenges of training reinforcement learning algorithms is implementing the environment correctly.

## 6.1 Tic-Tac-Toe rules

Tic-Tac-Toe is played by two players competing against each other on a 3x3 board. Each player is represented by either **x** or **o**. During the game, players alternately choose an available square on the board until the game ends with a win for a player or a draw.

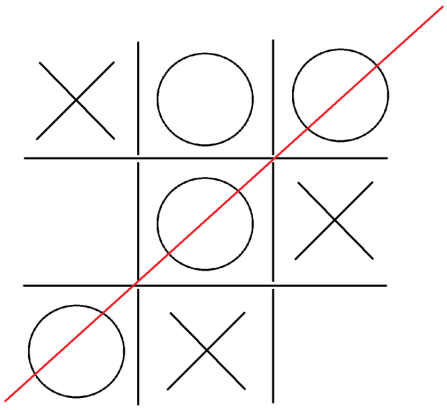


Figure 6.1.1: Tic-tac-toe board with a winning combination of player **o** . Source[1]

The goal for a player is to align three identical items horizontally, vertically, or diagonally. The first player, represented by either **o** or **x**, can be chosen randomly or according to a specified rule. Choosing a square that has already been selected is against the rules and is not permitted during a game. The simplicity of the rules will be further explained in the following section, where the environment setup will be explained.

## 6.2 Environment Implementation

As mentioned previously, reinforcement learning models do not necessarily need historical data to be trained. Still, they require setting up an appropriate environment replicating the real-world scenario or problem the reinforcement learning model aims to address. Several aspects must be considered when developing a reinforcement learning environment.

Let us begin with the action space. In tic-tac-toe, it represents all the empty squares on the board, which is a small and simple action space. during a game the agent can only choose empty squares, and if there are no more empty squares on the board, it means that the game ended with either a winner or a draw.

Now, moving to the state space. The state space is defined as the set of all possible states that can be realized during a game. For tic-tac-toe, since it is played on a 3x3 board where each position can be taken by **x**, **o**, or left empty, there are a total of 19683 possible states ($3^9$). However, some of these states are invalid and must be removed from the environment. For example, states that have more than one position taken by just one player are not valid. States where the difference between the number of position taken by **x** and by **o** is greater than one are also invalid. Indeed, because during a game the players

[1] https://dailycampus.com/2020/01/29/2020-1-29-tic-tac-toe-rubber-ducks-and-the-process-of-discoverynbsp/

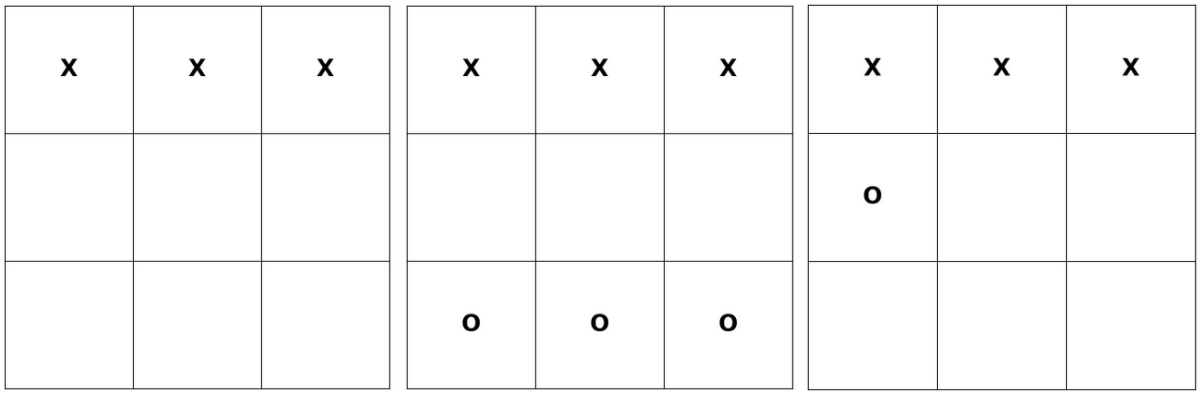



Figure 6.2.1: Examples of invalid tic-tac-toe states. On the first board, only one player has played. On the second board, there are two winning players. On the third board, there is a difference of more than one between **x** and **o**.

play in turns, it is impossible to end up with that kind of state during game. Thus, the difference in the number of positions taken by players during a match must be of maximum one. After removing all invalid states, the state space contains 8725 states. These states represent all possible configurations during a tic-tac-toe game and will be used to train the agent to make optimal moves whatever configuration of the board it faces. Reducing the state space to only valid states is important because it gives the agent an environment that has the same dynamic as real games and leads to better optimal policies. Moreover since the state space is now containing 8725 states, the algorithms have to iterate over a smaller state space and is less time consuming.

As already discussed, environment feedback plays an important role in training reinforcement learning algorithms. Therefore it is crucial that the environment gives feedback relevant for our use case so that the agent can learn to master its environment. For tic-tac-toe, we have set up the environment such that if the agent's action leads to a win, a positive reward of +1 is given. If the agent's action leads to the opponent's victory, a -1 penalty is given. Finally if the agent's action results in a draw, or if the next decision of the opponent does not end the game, then a feedback of 0 is given. We have implemented the feedback dynamic in that way because we wanted the agent to learn optimal policies that lead to win. Because if a reward is given when sub-goals are achieved, it can lead the agent to learn to optimize the sub-goals to the detriment of the original objective.

When setting up the environment, another essential aspect is to accurately model the dynamics of the real-world system. This involves understanding how the state evolves in response to the agent's actions. Therefore, transition probabilities from one state to another in the environment are needed to mimic real-world dynamics. Transition probabilities in Tic-Tac-Toe are easy to understand. For example, let us say the opponent (**o**) has the first hand and decides to choose the square in the middle. Then the probability of being

in a next possible state $p(s') = \frac{1}{\#possible\ next\ states}$, $\forall\ s' \in S$. So in that case, since the number of available squares equals eight if the opponent choose the middle square, we have that $p(s'|s) = \frac{1}{8}$. So the transition probabilities in Tic-Tac-Toe are quite straight forward. Indeed, let us consider the previous assumption about the opponent. If the agent (**x**) chooses the square just above the one in the middle, then the probability $p(s'|s, a) = 1$ because the only state possible when the agent takes that action, is the one where **o** is in the middle and **x** is just above. So when there is a given state on the board and one chooses a valid action there is only one possible next state. These dynamics enable the agent to learn the optimal move by anticipating the opponent's next move, reducing the opponent's chance of winning, and increasing its chances. Accurately defining an environment leads to an efficient learning process for the agent, ensuring stable training and accelerating convergence to optimal policies. This, in turn, makes it easier to analyze and interpret the agent's behavior. Moreover, an accurately implemented environment is valuable for debugging and simplifying the identification of potential issues.

## 6.3 Programming Setup

Since the hardware and software significantly influence the performance of code execution, it is important to present and explain how the code was developed. All the code and computation were done on a Lenovo Ideapad 320S, running on a 64-bit Windows operating system. The machine has 8GB of memory and an Intel(r) core(tm) i3-7100u CPU with two cores.

The programming language used is Python. Python was chosen because it is the most popular programming language in the machine learning community. Therefore, it was better and easier to find useful resources for reinforcement learning. The Python version used is 3.8.18. We also used Anaconda 3 to create a virtual environment and manage Python dependencies between packages.

## 6.4 Metrics Used to Evaluate And Compare The Policies

To evaluate the performances of the different models, we will use the winning rate against a random player. We will conduct simulations, providing the winning rate of the agent across 10,000 games played against a random player given two different scenarios. The first scenario represents the case where the trained agent is the first player. The second scenario represents the case where the random player is the first player. That will help to evaluate if the policies are sensitive to being the first player or not. We use the term random player to define a player that plays randomly without following any policy.

Afterwards, we opposed the three different policies against one another. The goal is to combine the results from the oppositions with the results from the simulations against random players and evaluate if one policy is better. Finally, we repeated the same analyses, adding random noise to the policies obtained with the three different algorithms. For the games against random players, we decided to study the effect of adding random noise on a percentage range from 10% to 90% of policies. We did that analysis for the case when the trained agent plays first and when it plays second. Then, we also did the same for the games that opposed the three different policies. For each game played, we add random noise on the policy for states that should occur when two agents play against one another and evaluate if there is a difference in the results with the games without perturbation.

## 6.5 Performance analysis

### 6.5.1 Analysis of policies obtained by policy iteration, value iteration and Monte Carlo control

Before going into the analysis of the policies, we performed a sanity check to see what the winning rate should be when two random players play tic tac toe. We simulated ten thousand games and we obtained the distribution shown in table 6.5.1, where we can see

| | Winning rate | | |
|---|---|---|---|
| | Random Player 1 | Draw | Random Player 2 |
| Random Player 1 vs Random player 2 | 0.592 | 0.125 | 0.283 |

Table 6.5.1: Table of winning rate when two random player play against each other. Random Player 1 is the player that starts and Random Player 2 is the player that plays second.

that playing first gives an advantage when two random players play against each other. Therefore this will be our baseline to benchmark the policies used for the analysis. When the different trained agents play against a random player, the winning rate when a trained agent starts should be at least equal to the one of Random Player 1 from table 6.5.1, 59.2%. For the case when a trained agent does not start the game, the winning rate should be at least equal to the one of Random Player 2 from table 6.5.1, 28.3%.

Now, let us look at the winning rate of the three policies when they play against random players.

In table 6.5.2, we can see that when the Agent starts, it achieves good performances no matter what policy it follows. Nevertheless, for the Monte Carlo policy, we see that it loses 0.9% of the games. If we suppose that for a win the agent receives a reward of +1, a draw a reward of 0 and a lost a reward of -1, then, if the agent wants to maximize its expected reward, when the agent starts a game, the agent should follow the Monte Carlo Policy even though there is a small chance of losing. For the case when the Random player starts, it should follow the Policy Iteration or Value Iteration since they will maximize its expected

| | First Player | | | | | |
|---|---|---|---|---|---|---|
| | Agent | | | Random Player | | |
| Outcome | Win | Draw | Lose | Win | Draw | Lose |
| Policy Iteration | 0.822 | 0.178 | 0.000 | 0.822 | 0.178 | 0.000 |
| Value Iteration | 0.903 | 0.097 | 0.000 | 0.824 | 0.176 | 0.000 |
| Monte Carlo ES | 0.988 | 0.003 | 0.009 | 0.769 | 0.137 | 0.093 |

Table 6.5.2: Table of winning rate for the three different agents. On the left part, there is the winning rate of the trained agent when it starts the game and uses Policy Iteration, Value Iteration or Monte-Carlo. On the right part, the same result but when the random player starts the game. Random players do not follow a specific policy and randomly choose their position on the board.

reward. Moreover, we can see that, for the case when the agent starts or when the random player starts, the winning rate of the three policies is higher than the one from table 6.5.1.

Now that we have the results for the policy when they play against a random player, let us look at the results when the policies play against each other. Since policies are deterministic, for policy opposition games, we only played one game of each opposition. In table 6.5.3,

| | | Second Player | | |
|---|---|---|---|---|
| | | Policy Iteration | Value Iteration | Monte Carlo ES |
| First Player | Policy Iteration | Draw | Draw | Policy Iteration |
| | Value Iteration | Draw | Draw | Draw |
| | Monte Carlo ES | Policy Iteration | Value Iteration | Monte Carlo ES 2 |

Table 6.5.3: Table of Policy Opposition results where on the left we have the first player of the game and on top, we have the second player. In the middle, we have the winner of the opposition game. The 2 index stands for the second player. For example, in the table above for Monte Carlo ES vs Monte Carlo ES, Monte Carlo ES 2 is the second player in the opposition.

we can see that Policy Iteration is achieving the best results against the Value Iteration and Monte Carlo ES. Indeed, if we suppose that the agent's goal is still to maximize its expected reward, then if it plays against another trained agent, it should follow Policy Iteration. Unlike the results against Random Players, the Monte Carlo policy is the worst policy to adopt when playing against other policies since it is the policy that has the lowest expected reward.

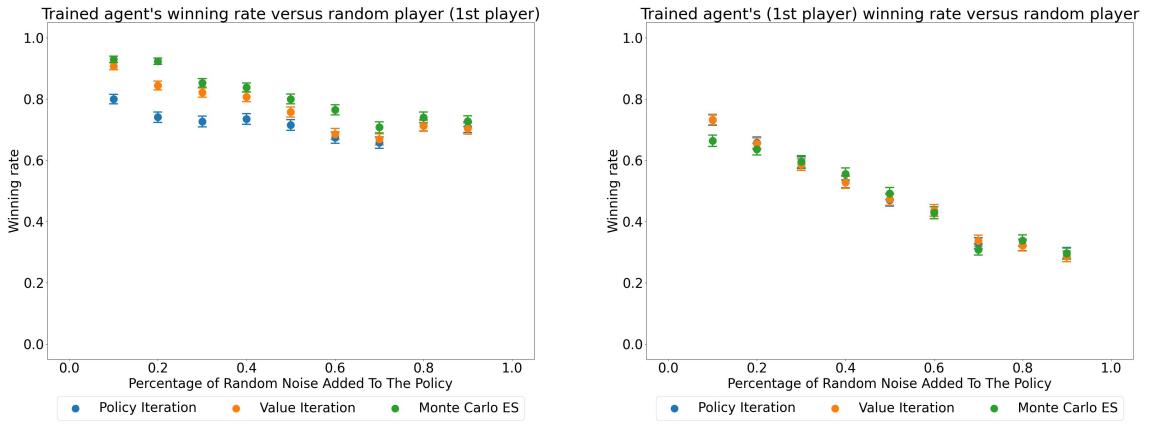
Based on our first analysis, we can conclude that when the agent is playing against a random player, it should follow the Monte Carlo ES policy. It should follow the Policy Iteration when it plays against another trained agent. Thus, we can also say that the Policy Iteration is almost insensitive to the game's first player, while value iteration and Monte Carlo ES are sensitive to the first player. The Value Iteration Policy still obtains good

results, but for the Monte Carlo's ES policy, sensitivity to the first player is more important. For games against random players, it goes from being the best policy when the agent starts to being the worst when the random player starts. Moreover, it is also the worst policy when it plays against other policies.

### 6.5.2 Analysis of policies obtained by policy iteration, value iteration and Monte Carlo control with random noise

For the second part of the performance analysis, we will analyze the results of the policies when we add random noise. We will first look at the results of the policies with random noise against random players and then the results when the policies play against each other.

For the analysis of the policies with random noise versus a random player, we analyzed the result with a percentage of random noise added to the policies, that ranges from 10% to 90% of the policies. We performed that analysis for the two scenarios of interest, the one when the agent starts and when the random player starts. We should expect the winning rate to converge to values close to the values in table 6.5.1 when more and more random noise is added.




(a) (b)

Figure 6.5.1: Figure (a) displays the winning rate of the different policies, and their 95% confidence interval, for different levels of random noise added to the policy when a trained using one of the policies starts the game. Figure (b) displays the winning rate of the different policies, and their 95% confidence interval, for different levels of noise added to the policy when the random player starts the game.

From figures 6.5.1 (a) and (b), we can see that for the three policies, we have the expected behavior. Moreover, when the agent starts, we can see in figure 6.5.1 (a) that Monte Carlo

ES has a higher winning rate than the two other policies for every level of random noise added. Thus, against random players, the Monte Carlo ES policy recovers better from random noise than Value Iteration and Policy Iteration. It also appears that Policy Iteration is the policy that recovers less well from random noise compared to Value Iteration and Monte Carlo ES when the trained agent starts. For the case when the random player starts, we can see in figure 6.5.1 (b) that policy iteration, value iteration and Monte Carlo ES have almost the same results. However, we can also see that for low amounts of noise (from 10% to 30%), Value Iteration and Policy Iteration are slightly better than Monte Carlo ES but after there is no clear difference between the three policies.

For the analysis when policies play against each other, we did it differently than table 6.5.3. Since the policies are deterministic, if we simulate 10.000 games of policy iteration versus value iteration, the same states and outcome will occur for the ten thousands simulations. Therefore we decided the add random noise to the policies only for the set of states that occur during an opposition between two policies. This ensure that perturbation is added when two policies plays against each other. We simulated 10.000 games for each opposition, using the set up explained and we added random noise to the policies that ranges from 10% to 90% of the set of states expected to occur during an opposition.

First, we will inspect the results when the policies play against themselves. In Figures 6.5.2 (a) and (b), we can see that when the Policy Iteration and Value Iteration Policy play against their own policy, there is an advantage for the second player. However, for the Monte Carlo Policy, we can see in Figure 6.5.2 (c) that the advantage of being the first player or the second player depends on the percentage of random noise added, although it is preferable to be the first for most the different levels of noise added.

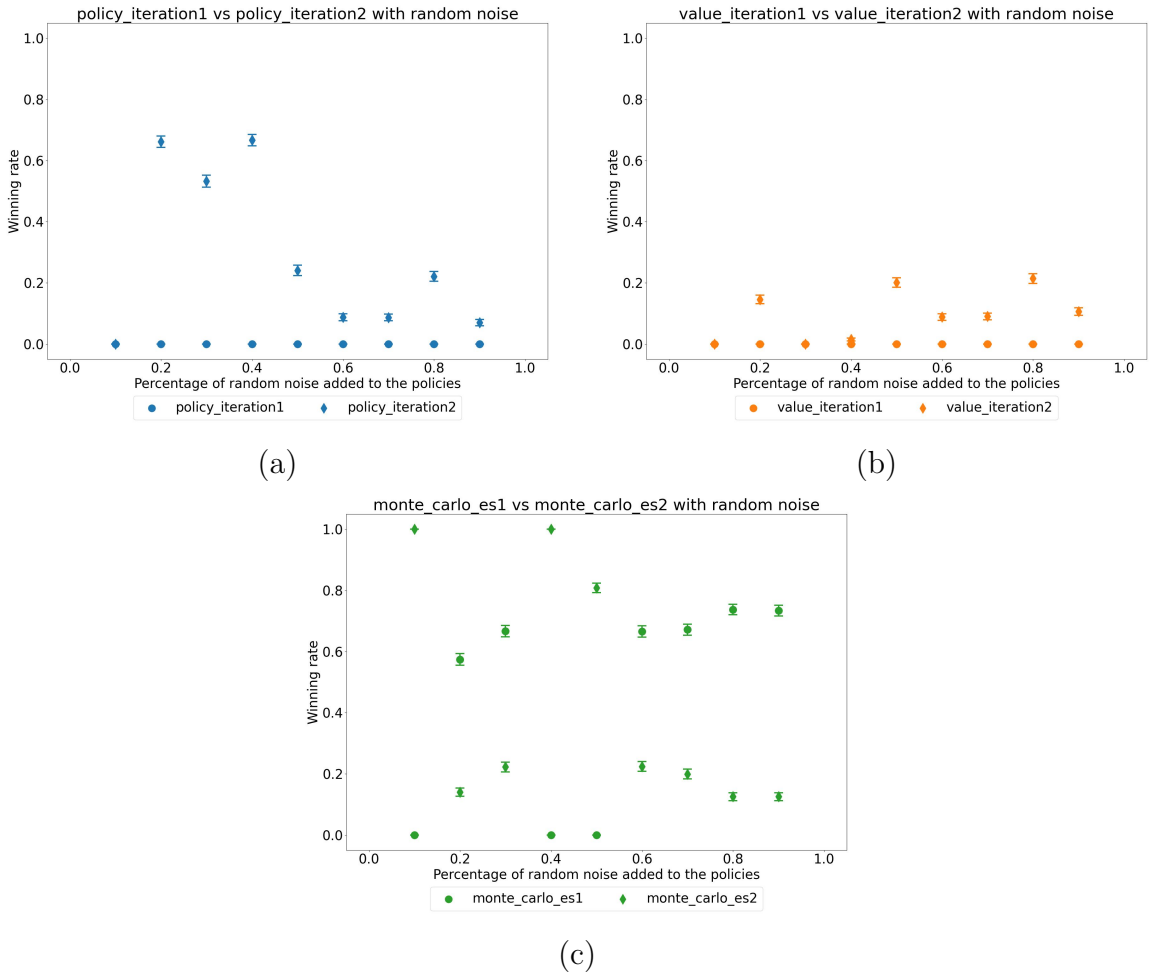



Figure 6.5.2: Winning rate and 95% confidence interval of the trained policies when they play against themselves and for different levels of random noise added to the policies. On figure (a) we have the results for Policy Iteration, on figure (b) the results for Value Iteration and on figure (c) the results for Monte Carlo ES.

Now, let us look at when policies play against other policies. For Policy Iteration versus Value Iteration, in figures 6.5.3 (a) and (b), one can see that when random noise is added, there is an advantage for the second player. It also appears that Policy Iteration has a small advantage because it has, in general, higher winning rate on figure 6.5.3 (a) than Value Iteration on figure 6.5.3 (b).

Now, let us look at the Policy Iteration versus Monte Carlo ES opposition. For that opposition, we can see in figures 6.5.3 (c) and (d), that Policy Iteration has a strategic advantage over Monte Carlo ES because it has better winning rates in either scenarios, when Policy Iteration starts and when Value Iteration starts. It still appears that playing second gives some advantage. For the Monte Carlo ES policy, one can see that when it plays second (see figure 6.5.3 (c)), it manages to have a positive winning rate and when it starts, its winning rate equals 0 for all levels of noise added (see figure 6.5.3 (d)).

Finally, let us analyze the opposition of Value Iteration versus Monte Carlo ES. In figures 6.5.3 (e) and (f), we can see that Value Iteration also has a strategic advantage over Monte Carlo ES. Value Iteration has a better winning rate in every opposition and either scenario.

Based on the analysis of policy opposition with random noise, we can say that playing second can give an advantage when random noise is added. Moreover, when we add random noise to the policies, Policy Iteration has a strategic advantage over the two other policies, even though it is a small advantage over Value Iteration.

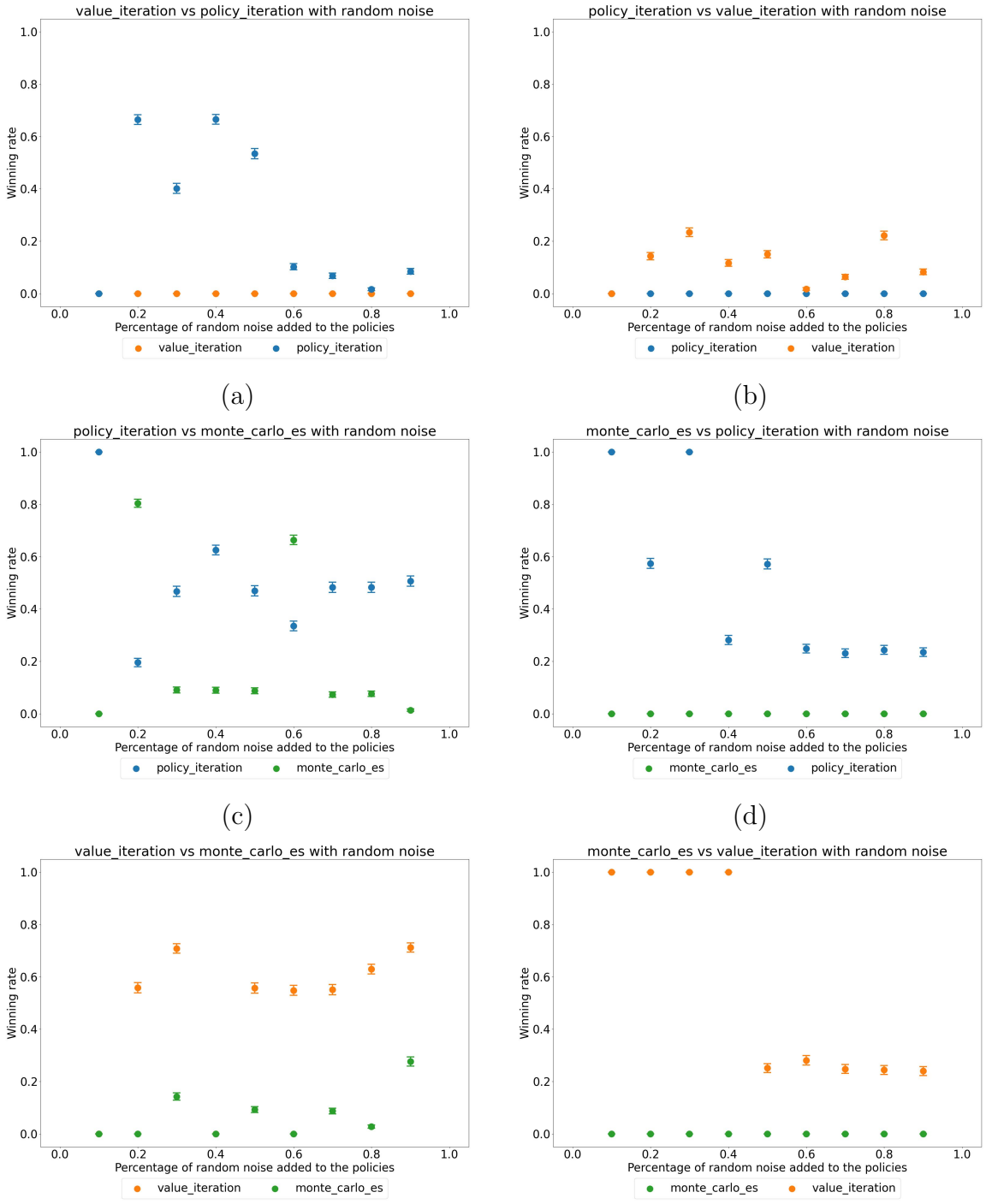



Figure 6.5.3: Winning rate and 95% confidence interval of the trained policies when they play against other trained policies and for different levels of random noise added to the policies. On figure (a) one can see the result of Value Iteration versus Policy Iteration when Value Iteration starts and On figure (b) the results when Policy Iteration starts. On figure (c) one can see the result of Policy Iteration versus Monte Carlo ES when Policy Iteration starts and On figure (d) the results when Monte Carlo ES starts. On figure (e) one can see the result of Value Iteration versus Monte Carlo ES when Value Iteration starts and On figure (f) the results when Monte Carlo ES starts.

# Chapter 7

# Reinforcement Learning in Finance: The QLBS Model

## 7.1 Useful finance vocabulary

Before going further into the chapter let us define some keywords.

- **Option**: An option is a financial derivative that gives its owner the right, but not the obligation, to buy or sell an underlying asset at a fixed price before a specified date for American-style option or at a specified date for European-style option.
- **Underlying Asset**: Real financial asset that a financial derivative is based on.
- **Option Strike (strike)**: The strike of an option is the fixed price at which the holder of the option can buy or sell the underlying asset.
- **Option Maturity (maturity)**: The last day on which the option holder may exercise its right for an American-style option or the day on which it may exercise its right for a European-style option.
- **Option Price**: The price of an option is the price that someone must pay to buy the option and get the right that comes with it.
- **Put option**: A put option is a financial derivative that gives its owner the right, but not the obligation, to sell an underlying asset at a fixed price (the strike) before a specified date (maturity) for American-style option or at a specified date for European-style option.
- **Call option**: A put option is a financial derivative that gives its owner the right, but not the obligation, to buy an underlying asset at a fixed price (the strike) before a specified date (maturity) for American-style option or at a specified date for European-style option.

- **A long position**: A long position in a financial asset consists of buying the asset and holding it with the expectation that its price will rise.
- **A short position**: A short position in a financial asset consists of selling the asset and buy it later it with the expectation that its price will fall. In general an investor will borrow the financial asset from someone, then sell it and buy it later. If the price fell, then the investor can reimburse the borrower with a benefit. If the price went up then the investor has to reimburse the borrower and pay for the price difference.
- **To hedge**: To hedge a primary position is an offsetting strategy that involves taking an opposite position in a related asset, which will gain (or lose) value if the primary position loses (or gains) value. Hedging strategies are used to manage risk.
- **To re-hedge**: To re-hedge a position consists of adjusting its original hedging position in response to the market conditions to maintain the same level of risk.
- **Optimal Hedging**: Optimal hedging consists of implementing a hedging strategy that minimises risk or maximises profits for a given level of risk.
- **Portfolio**: A portfolio is a combination financial assets (stocks, bonds, options,...).
- **Portfolio Valuation**: Portfolio valuation is the process of determining the value of a given portfolio.
- **Hedge Portfolio**: To hedge a portfolio means adapting a hedging strategy for a given portfolio that will reduce the risk of unexpected price movements on the market.
- **Risk Free rate**: Rate of return of an investment that has zero risk. As The risk free rate is a theoretical rate, it is generally represented by the rate of return of a government bond, as it is considered almost impossible for a government to go bankrupt.
- **Risk Neutral environment**: In option pricing, it represents an environment where the underlying asset's growth rate is equal to the risk-free rate instead of its real rate of return.
- $\Delta t$: Time step according to the re-hedging frequency.

## 7.2 Literature Review: Application of Reinforcement Learning in Finance

Some authors have shown that option pricing can be seen as a Markov Decision Process. Therefore, we will see on how Reinforcement learning can be used in option pricing and present the different results obtained in the literature. The price of an option depends on various things such as the price of the underlying asset, the expiration date of the option

and the strike. Even financial and economic factors such as interest rates can be considered as dependent variables for the option price. Li [2019] explains that to consider option pricing as a reinforcement learning problem, every factor cited before should be part of the state. Moreover, every uncertainty factor on the option price can be part of the state. Regarding the possible actions, there are two choices: exercise the right (to buy or sell) and continue (keep the option) [Li, 2019]. If one decides to exercise its option, the state action-value is the actual value of the option. That value can be calculated. If one decides not to exercise its option and continue, then there are no immediate rewards, or the reward is 0 [Li, 2019].

The work of Li et al. [2009] implements a Policy Iteration and Value Iteration algorithm to learn an exercise policy for American stock Options. In that work, they implemented the Least Square Policy Iteration (LSPI) from Lagoudakis and Parr [2003] and Fitted Value Iteration (FQI) from Tsitsiklis and Van Roy [2001] and they compare the performance against Least Square Monte Carlo (LSM) of Carriere [1996], a popular technique in the finance community. In that work, they analyzed the performance on American put options. They focused their work on at-the-money options, when the strike equals the initial stock price. They assume that the risk-free interest rate is constant, that the stock is non-dividend-paying and they considered 252 trading days each year. They only studied options with quarterly, semi-annual and annual expiration dates. They used five years of daily stock prices from January 2002 to December 2006 for Dow Jones 30 companies that they obtained from Wharton Research Data Services.

To perform the analysis, they use different simulation methods. They used two of the most popular models to model stock price variation, which is the Geometric Brownian Motion (GBM) and Generalized Auto Regressive Conditional Heteroskedasticity (GARCH). They use the GBM and GARCH models with parameters estimated from real data to learn an exercise policy. They also learn an exercise policy from sample paths obtained from real data directly. For this, they constructed multiple trajectories following a windowing technique due to the scarcity of real data and as there is only one stock price time series path for each company (Li et al. [2009]). From that technique, they could have, respectively 600, 500, and 500 training paths for quarterly, semi-annual, and annual maturity.

| Maturity | LSPI | | | FQI | | | LSM | | |
|---|---|---|---|---|---|---|---|---|---|
| | gbm | garch | data | gbm | garch | data | gbm | garch | data |
| quarterly | 1.310 | 1.333 | 1.339 | 1.321 | 1.341 | 1.331 | 0.573 | 0.572 | 0.719 |
| semi-annual | 1.681 | 1.663 | 1.739 | 1.718 | 1.749 | 1.797 | 0.693 | 0.687 | 0.887 |
| annual | 1.559 | 1.496 | 1.677 | 1.832 | 1.797 | 2.015 | 0.717 | 0.685 | 0.860 |

Table 7.2.1: Average payoffs of LSPI, FQI and LSM on real data for Dow Jones 30 companies. Source: Li et al. [2009]

In table 7.2.1, we can see the average payoffs of LSPI, FQI and LSM on real data

for Dow Jones 30. For each method, we can see the results on the data from the GBM model, GARCH model, and real data. That figure shows that LSPI and FQI outperform the famous LSM model. Their analysis shows that reinforcement learning techniques can produce better results than State-Of-The-Art techniques. This shows that finance can be an interesting field for reinforcement learning.

The Q-Learner in Black-Scholes (QLBS) from Halperin [2019], is a discrete-time option pricing model based on Reinforcement Learning and the famous Black-Scholes (B&S) model. To summarise, the QLBS model is a model that implements the concept of pricing and hedging in a consistent way for a discrete-time version of the (B&S) model [Halperin, 2019]. In the QLBS model, the rewards are the negative of the one-step variances of the hedge portfolios multiplied by risk aversion, plus a drift term [Halperin, 2019]. Actions are the possible hedge positions. The states are defined as a change of variable of a stock dynamic, such that the change of variable is stationary. This ensures that the new state variable is a standard Brownian motion [Halperin, 2019].
In the QLBS model, the rewards are the negative of the one-step variances of the hedge portfolios multiplied by risk aversion, plus a drift term.

An extension of the QLBS model for reinforcement learning-based, data-driven, and model-independent option pricing with experimentation was provided in Halperin [2018]. With his work, Halperin [2018] showed that the FQI offers a reasonable level of noise tolerance with regard to the possible sub-optimality of the actions observed in the QLBS model. This renders the QLBS model capable of hedging and pricing even when actions are sub-optimal or not mutually consistent for different time steps [Halperin, 2018].

To summarize, some great work has already been done on the topic of reinforcement learning in finance. Therefore, finance can be a great field in which to apply and develop reinforcement learning since there is still room for improvement. It is important to note that financial markets are complex and dynamic, and applying Reinforcement Learning techniques requires careful consideration of the specific challenges and risks involved. Additionally, issues related to interpretability and explicability are crucial in finance, where regulatory and compliance requirements are often strict.

## 7.3 The Black Scholes Merton model

The Black-Scholes-Merton model is a mathematical model that provides a theoretical estimate for European-style option prices. Under the B&S model for option pricing and hedging, the stock price dynamic is driven by a Geometric Brownian Motion (GBM) that takes a drift parameter $\mu$ and a volatility parameter $\sigma$. The stock price dynamic can be expressed using the following formula:

$$dS_t = \mu S_t dt + \sigma S_t dW_t \tag{7.3.1}$$

where $S_t$ is the stock price at moment $t$ and $W_t$ is a standard Brownian Motion.
Suppose that $C(S_t, t)$ is the price at moment $t$ of any derivative on the underlying asset $S_t$. From Itô's lemma we can show that $C(S_t, t)$ has the following dynamic

$$dC(S_t,t) = \Big(\frac{\partial C(S_t,t)}{\partial S_t}\mu S_t + \frac{\partial C(S_t,t)}{\partial t} + \frac{1}{2}\frac{\partial^2 C(S_t,t)}{\partial S_t^2}\sigma^2 S_t^2\Big)dt + \frac{\partial C(S_t,t)}{\partial S_t}\sigma S_t dW_t. \quad (7.3.2)$$

To eliminate the standard Brownian Motion one can build a portfolio that contains the derivative and the stock. That portfolio must short one derivative and long $\frac{\partial C(S_t,t)}{\partial S_t}$ of the stock. Let us define $\Psi$ as the value of that portfolio, so we have that

$$\Psi = -C(S_t,t) + \frac{\partial C(S_t,t)}{\partial S_t}S_t. \quad (7.3.3)$$

From equation (7.3.3), we have that the change rate of $\Psi$, $d\Psi$, is given by

$$d\Psi = -dC(S_t,t) + \frac{\partial C(S_t,t)}{\partial S_t}dS_t. \quad (7.3.4)$$

Substituting equations (7.3.1) and (7.3.2) into equations (7.3.4) gives

$$d\Psi = \Big(-\frac{\partial C(S_t,t)}{\partial t} - \frac{1}{2}\frac{\partial^2 C(S_t,t)}{\partial S_t^2}\sigma^2 S_t^2\Big)dt. \quad (7.3.5)$$

We can see that since equation (7.3.5) does not depend on $dW_t$, it must be risk less and must have the same rate of return as other short-term risk-free securities. So,

$$d\Psi = r\Psi dt \quad (7.3.6)$$

where $r$ is the risk-free rate.
Introducing equations (7.3.3) and (7.3.5) into equation (7.3.6) gives

$$\frac{\partial C(S_t,t)}{\partial t} + rS_t\frac{\partial C(S_t,t)}{\partial S_t} + \frac{1}{2}\frac{\partial^2 C(S_t,t)}{\partial S_t^2}\sigma^2 S_t^2 = rC(S_t,t) \quad (7.3.7)$$

Equation (7.3.7) is known as the Black-Scholes-Merton differential equation. Depending on the derivative that can be defined with $S_t$, that equation has different solutions. The derivative that can be computed depends on boundary conditions that are used to solve the equation. For a European Call option, the boundary condition is

$$C(S_t,t) = \max(S_t - K, 0) \; when \; t = T \quad (7.3.8)$$

and for a European Put option, the boundary condition is

$$C(S_t,t) = \max(K - S_t, 0) \; when \; t = T \quad (7.3.9)$$

where $K$ is the option strike price.
From equation (7.3.5) the most famous solutions are the Black-Scholes-Merton formulas to price European Call options and European Put options. Fot the call option we have

$$c(S_t, K, r, T, \sigma) = S_t N(d_1) - K e^{-r(T-t)} N(d_2) \tag{7.3.10}$$

and for the put option we have

$$p(S_t, K, r, T, \sigma) = K e^{-r(T-t)} N(-d_2) - S_t N(-d_1) \tag{7.3.11}$$

where $N(\cdot)$ is the cumulative distribution function of the standard normal distribution and

$$d_1 = \frac{\ln\left(\frac{S_t}{K}\right) + (r + \frac{\sigma^2}{2})T}{\sigma\sqrt{T}} \tag{7.3.12}$$

$$d_2 = \frac{\ln\left(\frac{S_t}{K}\right) + (r - \frac{\sigma^2}{2})T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T}. \tag{7.3.13}$$

## 7.4 The QLBS Model setup

Let us take the view of the seller of a European Put option, with terminal payoff $H_T(S_T) = \max(K - S_T, 0)$. To hedge its position, the seller can replicate a portfolio $\Psi^{qlbs}$ composed of $S_t$ and a risk-free bank deposit $B_t$. The value of $\Psi^{qlbs}$ at $t \leq T$ is

$$\Psi_t^{qlbs}(S_t) = a_t S_t + B_t \tag{7.4.1}$$

where $a_t$ is here a position in the stock at moment $t$, to hedge risk in the option.
At maturity, the stock position in the portfolio should be closed. Since we are ignoring transaction costs, we can do this by simply setting $a_T = 0$ which will convert the hedge portfolio into cash that is equal to the option's payoff at maturity. So we have that

$$\Psi_T^{qlbs}(S_T) = B_T = \max(K - S_T, 0) \tag{7.4.2}$$

which is the terminal condition for $B_t$ when $t = T$.

The QLBS model has a self-financing constraint which requires that all future changes in the hedge portfolio should be financed from an initially set bank account, without cash deposits or withdrawals over the option lifetime [Halperin, 2019]. This implies that

$$a_t S_{t+1} + e^{r\Delta t} B_t = a_{t+1} S_{t+1} + B_{t+1} \tag{7.4.3}$$

which ensures the conservation of the portfolio's value by a re-hedge at time $t+1$ [Halperin, 2019]. From equation (7.4.3) we can show that

$$B_t = e^{-r\Delta t}[B_{t+1} + (a_{t+1} - a_t)S_{t+1}],\ t = T-1, ..., 0. \tag{7.4.4}$$

Then plugging this last equation into equation (7.4.1) gives

$$\Psi_t^{qlbs}(S_t) = e^{-r\Delta t}[\Psi_{t+1}^{qlbs} - a_t \Delta S_t] \tag{7.4.5}$$

where $\Delta S_t = S_{t+1} - e^{r\Delta t} S_t$ and $t = T-1, ..., 0$.
When inspecting equations (7.4.4) and (7.4.5) we can see that it is not possible to compute $B_t$ and $\Psi_t^{qlbs}(S_t)$ since they depend on the values at $t+1$ which are unknown. Since $B_t$ and $\Psi_t^{qlbs}(S_t)$ also depend on $S_t$ we also need the paths of $S_t$. To get $S_t$ paths, the model uses Monte Carlo simulation to generate $N_{mc}$ paths $S_1 \to S_2 \to ... \to S_{N_{mc}}$ that are used to evaluate $\Psi_t^{qlbs}(S_t)$ backwards since we know that $\Psi_T^{qlbs}(S_T) = \max(K - S_T, 0)$.

We saw that under the BS model, the stock price dynamic can expressed as in equation (7.3.1). Using Itô's lemma to solve equation (7.3.1) we get that

$$d(\ln(S_t)) = \left(\mu - \frac{\sigma^2}{2}\right)dt + \sigma dW_t \ . \tag{7.4.6}$$

Since stock prices are non-stationary, we will consider a change of state variable in order to use a time-homogeneous variable $X_t$, where the relationship between $X_t$ and $S_t$ is as follows

$$X_t = -\left(\mu - \frac{\sigma^2}{2}\right)t + \ln(S_t) \ . \tag{7.4.7}$$

This relationship implies that

$$dX_t = -\left(\mu - \frac{\sigma^2}{2}\right)dt + d(\ln(S_t)) = \sigma dW_t \ , \tag{7.4.8}$$

therefore we can say that $X_t$ is a standard Brownian motion scaled by volatility $\sigma$. In some cases $X_t$, will be referred to using the notation $X_t(S_t)$. It can also be shown that

$$S_t = e^{X_t + \left(\mu - \frac{\sigma^2}{2}\right)t}. \tag{7.4.9}$$

Let us recall that in the QLBS model, $a_t$ refers to the position on the stock at moment $t$ and can also be written as $a_t(S_t)$. Since we are using a new variable $X_t$, $a_t$ will now be referring to the position in $X_t$ and can be written as $a_t(X_t)$. In the QLBS setup, the action in terms of stock price is simply $a_t(S_t) = a_t(X_t(S_t))$.

### 7.4.1 Optimal Hedging in the QLBS setup

As for portfolio valuation (see equation 7.4.5), the optimal hedging position $a_t^*$ can be computed backward, starting from $t = T$. Since it is not possible to know the future when we want to compute a hedge at a moment $t$, the computation of the optimal hedge position is conditional on the information available at time $t$ which we will refer to as $F_t$.

In the QLBS model, the optimal hedge position $a_t^*$ is obtained from the requirement to minimize the variance of $\Psi_t^{qlbs}(S_t)$, obtained from Monte Carlo simulations, conditional on the cross-sectional information available $F_t$ [Halperin, 2019]. So we have that

$$\begin{aligned} a_t^*(S_t) &= \arg\min_{a_t} Var[\Psi_t^{qlbs}(S_t)|F_t] \\ &= \arg\min_{a_t} Var[\Psi_{t+1}^{qlbs} - a_t \Delta S_t|F_t]. \end{aligned} \tag{7.4.10}$$

### 7.4.2 Option Pricing in QLBS

In the work of Halperin [2019] they defined the fair option $\hat{C}(S_t, t)$ as the expected value of $\Psi_t^{qlbs}(S_t)$ conditional on $F_t$:

$$\hat{C}(S_t, t) = \mathbb{E}_t[\Psi_t^{qlbs}(S_t)|F_t]. \tag{7.4.11}$$

However the trader cannot simply ask for the fair price $\hat{C}(S_t, t)$ when he sells the option because it needs to account for the risk that the bank account $B_t$ may not cover the needs of the portfolio. To compensate that risk, a risk premium must be added to the fair price. In QLBS, one possible specification of the risk premium is the cumulative expected discounted variance of the hedge portfolio for all $t = 0, 1, ..., T$ with a risk aversion parameter $\lambda$ [Halperin, 2019]. So we have that

$$\hat{C}^{(ask)}(S_t, t) = \mathbb{E}_t\Big[\Psi_t^{qlbs}(S_t) + \lambda \sum_{t=0}^{T} e^{-rt} Var[\Psi_t^{qlbs}(S_t)|F_t]|S_t = S, a_t = a\Big]. \tag{7.4.12}$$

So now the goal is to minimize the fair option price $\hat{C}^{(ask)}(S_t, t)$, which can equivalently be expressed as maximizing the value function in the QLBS setup $V_t(S_t) = -\hat{C}^{(ask)}(S_t, t)$, and we have that

$$V_t(S_t) = \mathbb{E}_t\Big[-\Psi_t^{qlbs}(S_t) - \lambda \sum_{t'=t}^{T} e^{-r(t'-t)} Var[\Psi_{t'}^{qlbs}(S_t')|F_{t'}]|F_t\Big]. \tag{7.4.13}$$

### 7.4.3 Value function and Bellman equation in the QLBS setup

Now let us see how value functions and Bellman equations are used in the QLBS setup. To start we will use the new state variable $X_t$ from equation (7.4.7), and we write the value function of a policy $\pi$, in the QLBS setup, as follows

$$\begin{aligned} V_t^{\pi}(X_t) &= \mathbb{E}_t\Big[-\Psi_t^{qlbs}(X_t) - \lambda \sum_{t'=t}^{T} e^{-r(t'-t)} Var[\Psi_{t'}^{qlbs}|F_{t'}]|F_t\Big] \\ &= \mathbb{E}_t\Big[-\Psi_t^{qlbs}(X_t) - \lambda Var[\Psi_t^{qlbs}(X_t)] - \lambda \sum_{t'=t+1}^{T} e^{-r(t'-t)} Var[\Psi_{t'}^{qlbs}(X_t')|F_{t'}]|F_t\Big] \end{aligned} \tag{7.4.14}$$

where $\pi$ is a policy such that $\pi(t, X_t) = a_t(X_t)$. The last term of equation (7.4.14) can be expressed in terms of $V_{t+1}$ using the definition of value function with a shifted time argument [Halperin, 2019]:

$$-\lambda \mathbb{E}_{t+1}\Big[\sum_{t'=t+1}^{T} e^{-r(t'-t)} Var[\Psi_{t'}^{qlbs}(X_t')|F_{t'}]\Big] = \gamma\Big(V_{t+1}(X_{t+1}) + \mathbb{E}_{t+1}[\Psi_{t+1}^{qlbs}(X_{t+1})]\Big) \quad (7.4.15)$$

where $\gamma = e^{-r\Delta t}$, is a discount factor.
Plugging equation (7.4.15) into equation (7.4.14) gives the Bellman equation for the QLBS model

$$V_t^{\pi}(X_t) = \mathbb{E}_t\Big[-\Psi_t^{qlbs}(X_t) - \lambda Var(\Psi_t(X_t)) + \gamma\Big(V_{t+1}(X_{t+1}) + \mathbb{E}_{t+1}[\Psi_{t+1}^{qlbs}(X_{t+1})]\Big)|F_t\Big] \quad (7.4.16)$$

then plugging equation (7.4.5) into equation (7.4.16) gives

$$\begin{aligned} V_t^{\pi}(X_t) &= \mathbb{E}_t\Big[-e^{-r\Delta t}[\Psi_{t+1}^{qlbs}(X_{t+1}) - a_t\Delta X_t] - \lambda Var(\Psi_t(X_t)) + \gamma V_{t+1}(X_{t+1}) + \gamma \mathbb{E}_{t+1}[\Psi_{t+1}^{qlbs}(X_{t+1})]|F_t\Big] \\ &= \mathbb{E}_t\Big[-\gamma\Psi_{t+1}^{qlbs}(X_{t+1}) + \gamma a_t\Delta X_t - \lambda Var(\Psi_t(X_t)) + \gamma V_{t+1}(X_{t+1}) + \gamma \mathbb{E}_{t+1}[\Psi_{t+1}^{qlbs}(X_{t+1})]|F_t\Big] \\ &= \mathbb{E}_t\Big[R(X_t, a_t, X_{t+1}) + \gamma V_{t+1}(X_{t+1})\Big] \end{aligned} \quad (7.4.17)$$

where $R(X_t, a_t, X_{t+1})$ is the one-step time-dependent random reward and is defined as

$$R(X_t, a_t, X_{t+1}) = \gamma a_t(X_t)\Delta X_t - \lambda Var(\Psi_t(X_t)|F_t),\ t = T-1, ..., 0 \quad (7.4.18)$$

where $\Delta X_t$ refers to $X_{t+1} - e^{r\Delta t}X_t$ and the variance in the equation (7.4.18) can be expressed as

$$\begin{aligned} Var(\Psi_t(X_t)|F_t) &= Var[e^{-r\Delta t}(\Psi_{t+1}^{qlbs}(X_{t+1}) - a_t\Delta X_t)|F_t] \\ &= \gamma^2 Var[\Psi_{t+1}^{qlbs}(X_{t+1}) - a_t\Delta X_t|F_t] \\ &= \gamma^2 \mathbb{E}_t\Big[\Big((\Psi_{t+1}^{qlbs}(X_{t+1}) - \mathbb{E}_t[\Psi_{t+1}^{qlbs}(X_{t+1})]) - (a_t\Delta X_t - \mathbb{E}_t[a_t\Delta X_t])\Big)^2\Big] \\ &\approx \gamma^2 \mathbb{E}_t\Big[(\hat{\Psi}_{t+1}^{qlbs}(X_{t+1}) - a_t\Delta\hat{X}_t)^2\Big] \\ &= \gamma^2 \mathbb{E}_t\Big[(\hat{\Psi}_{t+1}^{qlbs}(X_{t+1}))^2 - 2a_t\Delta\hat{X}_t\hat{\Psi}_{t+1}^{qlbs}(X_{t+1}) + a_t^2(\Delta\hat{X}_t)^2\Big] \end{aligned} \quad (7.4.19)$$

with $\hat{\Psi}_{t+1}^{qlbs}(X_{t+1}) \equiv \Psi_{t+1}^{qlbs}(X_{t+1}) - \bar{\Psi}_{t+1}^{qlbs}(X_{t+1})$ and $\bar{\Psi}_{t+1}^{qlbs}(X_{t+1})$ the sample mean of $\Psi_{t+1}^{qlbs}(X_{t+1})$ on all Monte Carlo paths generated. Similarly, $\Delta\hat{X}_t = \Delta X_t - \Delta\bar{X}_t$.
So if we plug equation (7.4.19) into equation (7.4.18), we have that

$$R(X_t, a_t, X_{t+1}) = \gamma a_t(X_t)\Delta X_t - \lambda\gamma^2 \mathbb{E}_t\Big[(\hat{\Psi}_{t+1}^{qlbs}(X_{t+1})^2 - 2a_t\Delta\hat{X}_t\hat{\Psi}_{t+1}^{qlbs}(X_{t+1}) + a_t^2(\Delta\hat{X}_t)^2\Big]$$

(7.4.20)

and it follows that the expected reward is

$$\mathbb{E}_t\Big[R(X_t, a_t, X_{t+1})\Big] = \gamma a_t(X_t)\mathbb{E}_t[\Delta X_t] - \lambda\gamma^2\mathbb{E}_t\Big[(\hat{\Psi}^{qlbs}_{t+1}(X_{t+1}))^2 - 2a_t\Delta\hat{X}_t\hat{\Psi}^{qlbs}_{t+1}(X_{t+1}) + a_t^2(\Delta\hat{X}_t)^2\Big]. \tag{7.4.21}$$

The $Q$-function or action-value function in the QLBS setup, is simply defined as in equation (7.4.14) but conditional on $X_t$ and $a_t$ following a given policy $\pi$, so we have that

$$Q^\pi_t(x,a) = \mathbb{E}_t\Big[-\Psi(X_t)|X_t = x, a_t = a\Big] - \lambda\mathbb{E}^\pi_t\Big[\sum_{t'=t}^{T} e^{-r(t'-t)}Var\Big[\Psi^{qlbs}_{t'}(X_{t'})|F_{t'}|X_t = x, a_t = a\Big]\Big]. \tag{7.4.22}$$

In the QLBS setup the goal is to find the optimal policy $\pi^*$ that maximizes the value function $V^\pi_t(X_t)$ or the $Q$-function $Q^\pi_t(X_t, a_t)$, so we have that

$$\pi^*_t(X_t) = \arg\max_\pi V^\pi_t(X_t) = \arg\max_\pi Q^\pi_t(X_t, a_t) \tag{7.4.23}$$

The optimal value function satisfies the Bellman optimality equation and is given by

$$V^*_t(X_t) = \mathbb{E}_t\Big[R(X_t, a_t = \pi^*_t(X_t), X_{t+1}) + \gamma V^*_{t+1}(X_{t+1})\Big] \tag{7.4.24}$$

and the optimal $Q$-function is given by

$$Q^*_t(X_t, a_t) = \mathbb{E}_t\Big[R(X_t, a_t, X_{t+1}) + \gamma \max_{a_{t+1}\in A} Q^*_{t+1}(X_{t+1}, a_{t+1})|X_t = x, a_t = a\Big], t = 0, ..., T-1 \tag{7.4.25}$$

where a terminal condition, when $t = T$, is given by

$$Q^*_T(X_T, a_T = 0) = -\Psi^{qlbs}_T(X_T) - \lambda Var\Big[\Psi_T(X_T)\Big]. \tag{7.4.26}$$

### 7.4.4 Dynamic programming solution in the QLBS setup

The goal of Dynamic Programming in the QLBS setup is to use backward recursion to solve the Bellman optimality equation (7.4.25) jointly with the optimal policy from equation (7.4.23), starting from $t = T-1$ and the terminal condition from equation (7.4.26). To achieve this, we first need to plug equation (7.4.21) into equation (7.4.25) which gives

$$\begin{aligned} Q^*_t(X_t, a_t) = \gamma\mathbb{E}_t\Big[Q^*_{t+1}(X_{t+1}, a^*_{t+1}) + a_t\Delta X_t\Big] - \\ \lambda\gamma^2\mathbb{E}_t\Big[(\hat{\Psi}^{qlbs}_{t+1}(X_{t+1}))^2 - 2a_t\Delta\hat{X}_t\hat{\Psi}^{qlbs}_{t+1}(X_{t+1}) + a_t^2(\Delta\hat{X}_t)^2\Big], \end{aligned} \tag{7.4.27}$$

for $t = 0, ..., T-1$.
We can see that $Q_t^*(X_t, a_t)$ is a quadratic function of $a_t$. Therefore, finding the optimal action $a_t^*$ that maximizes $Q_t^*(X_t, a_t)$ can be done by setting its first derivative $\frac{\partial Q_t^*(X_t, a_t)}{\partial a_t}$ to zero which gives

$$a_t^*(X_t) = \frac{\mathbb{E}_t\Big[\Delta\hat{X}_t\hat{\Psi}_{t+1}^{qlbs} + \frac{1}{2\gamma\lambda}\Delta X_t\Big]}{\mathbb{E}_t\Big[\Big(\Delta\hat{X}_t\Big)^2\Big]}. \tag{7.4.28}$$

Plugging back equation (7.4.28) into equation (7.4.27) gives an explicit recursive formula for the optimal $Q$-function:

$$Q_t^*(X_t, a_t^*) = \gamma\mathbb{E}_t\Big[Q_{t+1}^*(X_{t+1}, a_{t+1}^*) - \lambda\gamma(\hat{\Psi}_{t+1}^{qlbs}(X_{t+1}))^2 + \lambda\gamma\Big(a_t^*\Big)^2\Big(\Delta\hat{X}_t\Big)^2\Big] \tag{7.4.29}$$

for $t = 0, ..., T-1$.
So using equation (7.4.28), equation (7.4.29), the terminal condition (7.4.26) and starting from $t = T-1$ to the present, we can recursively find the solution of $Q_t^*(X_t, a_t^*)$ so that the QLBS option price expressed in terms of $Q_t^*(X_t, a_t^*)$ is given by

$$\hat{C}^{(ask)}(S_t, t) = -Q_t^*(X_t, a_t^*). \tag{7.4.30}$$

To implement dynamic programming solution, we have access to $N_{mc}$ simulated paths for the state variable $X_t$ using Monte Carlo Simulations. The QLBS model reproduces the portfolio simultaneously at time $t$ and $t+1$ using all the paths simulated with Monte Carlo method. Since we are in RL setting this is equivalent to asynchronous updates of the policy and the $Q$-function, which significantly slows down the learning process [Halperin, 2019]. So the model also uses a set of basis functions $\{\Phi_n(x)\}$ and expand the optimal action and optimal $Q$-function in basis functions, time-depend coefficients [Halperin, 2018]:

$$a_t^*(X_t) = \sum_{n=1}^{M}\phi_{nt}\Phi_n(X_t), \quad Q_t^*(X_T, a_t^*) = \sum_{n=1}^{M}\omega_{nt}\Phi_n(X_t). \tag{7.4.31}$$

The coefficients $\phi_{nt}$ and $\omega_{nt}$ are computed recursively backward in time for $t = T-1, ..., 0$, which gives

$$\phi_t^* = \mathbf{A}_t^{-1}\mathbf{B}_t \tag{7.4.32}$$

where

$$A_{nm}^{(t)} = \sum_{k=1}^{N_{mc}}\Phi_n(X_t^k)\Phi_m(X_t^k)(\Delta\hat{S}_t^k)^2$$

$$B_{nm}^{(t)} = \sum_{k=1}^{N_{mc}}\Phi_n(X_t^k)\Big[\hat{\Psi}_{t+1}^{qlbs}\Delta\hat{S}_t^k + \frac{1}{2\gamma\lambda}\Delta S_t^k\Big]$$

and

$$\omega_t^* = \mathbf{C}_t^{-1}\mathbf{D}_t \tag{7.4.33}$$

where

$$C_{nm}^{(t)} = \sum_{k=1}^{N_{mc}} \Phi_n(X_t^k)\Phi_m(X_t^k)$$

$$D_n^{(t)} = \sum_{k=1}^{N_{mc}} \Phi_n(X_t^k)\Big(R_t(X_t^k, a_t^{k*}, X_{t+1}^k) + \gamma \max_{a_{t+1}\in\mathcal{A}} Q_{t+1}^*(X_{t+1}^k, a_{t+1})\Big)$$

Equations (7.4.32) and (7.4.33) are computed recursively for $t = T-1, ..., 0$ and give a practical implementation of the dynamic programming based solution to the QLBS model using expansions in basis functions [Halperin, 2018].

## 7.5 QLBS Model Application on Simulated Data

In this section, we generated a sample of 100 option prices using the QLBS model and 10000 Monte Carlo simulations to generate stock prices. The results were compared with the B&S model. For the numerical example, the initial stock price $S_0 = 100$, the drift $\mu = 0.05$ and volatility $\sigma = 0.15$. The option's maturity is $T = 1$ year, the risk free rate is set to $r = 0.03$. For the experiment we considered a European put option with strike $K = 100$. Re-hedges are done every two weeks (i.e $T = 24$ with time interval $\Delta t = \frac{1}{24}$). We also used 12 basis functions chosen to be cubic B-splines on a range of value that go from the smallest to the largest values of $X_t$. Finally we used the Markowitz risk aversion parameter $\lambda = 0.00$. It as been set to 0 because option pricing, in the B&S setup, is done in a risk neutral environment and a risk aversion parameter of 0.00 represents a risk neutral environment in the QLBS setup.

| B&S Put Price | | |
|---|---|---|
| 4.53 | | |
| QLBS Put Price | 95% CI lower bound | 95% CI upper bound |
| 4.43 | 4.42 | 4.43 |

Table 7.5.1: Table to compare the QLBS model and B&S model Put Option Price. For QLBS price there is the 95% confidence interval and not for the B&S price because the model is deterministic.

In Table 7.5.1 we can see the results for the QLBS model and the B&S model for Put Option are close with the numerical setup we used. Although we have a sample with only 100 prices, the 95% confidence interval for the QLBS put option price is small which means that model is precise and the variability is low in the set up we used.

Since a precise model does not mean an accurate model, in the next section we will evaluate the accuracy of the model using real option prices.

## 7.6 QLBS Model Application on Real Financial Data: Option Pricing On LVMH Stock

Now we will apply QLBS on LVMH (Louis Vuitton Moet Hennessy) stock. LVMH is a French company and one of the biggest companies traded on the Euronext. It operates in the luxury goods market and is a global market leader. We will use LVMH stock prices from August 2019 to August 2024 to fit an option pricing model using QLBS and compare the results with the with real option prices.

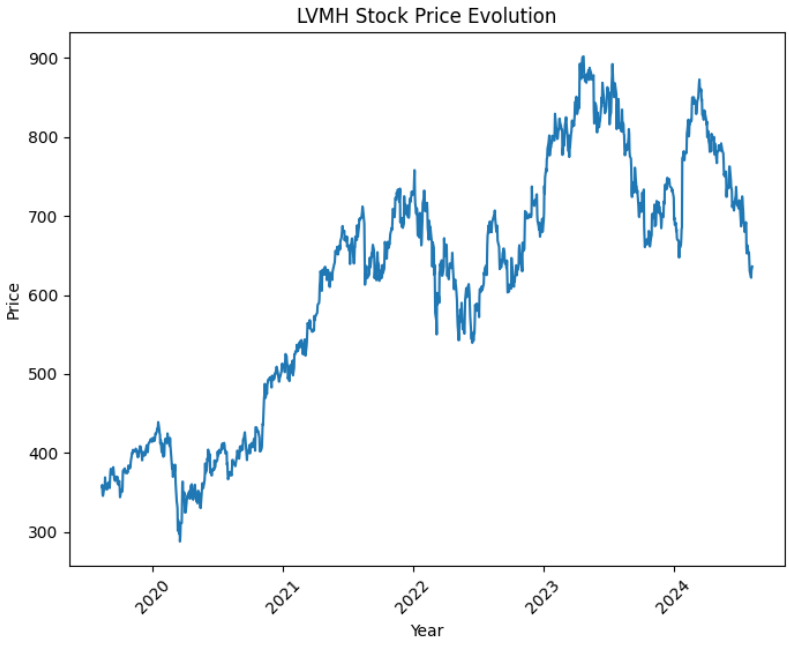



Figure 7.6.1: Evolution of LVMH stock Price from August 2019 to August 2024

From figure 7.6.1 we can see that LVMH stock price rose between August 2019 and August 2024, the stock price has almost doubled. To compute the drift and volatility parameters needed to use for QLBS and B&S model we analyzed the log returns. From the log return that you can see on figure A.1.1 we computed the annual drift for LVMH that is equal to 0.11 and the annual volatility that is equal to 0.29.

Now that we have the volatility and the drift for the LVMH, let us compute the put option price using QLBS and B&S model. To do it we will use the same approach that for the example on simulated data, except that some parameters will be fixed using the results

from LVMH stock price analysis. Based on the analysis of LVMH stock, we have that the drift parameter $\mu$ is equal to 0.11 and the volatility parameter $\sigma$ is equal to 0.29. The initial stock price is 636.00 and the strike used for the analysis is 700.00. As risk free rate, we will use the one year French government bond rate of return that is equal to 0.03083[1]. The risk aversion parameter $\lambda$ is set to 0.00. Finally for the maturity we will evaluate the model when the maturity is set to 4 months, 10 months, 16 months, 22 months and 28 months.

For the QLBS model we obtained the following results

| Maturity | QLBS Put Price | 95% CI Lower Bound | 95% CI Upper Bound |
|---|---|---|---|
| 4 months | 77.60 | 77.56 | 77.65 |
| 10 months | 93.98 | 93.91 | 94.04 |
| 16 months | 105.59 | 105.49 | 105.69 |
| 22 months | 115.06 | 114.91 | 115.21 |
| 28 months | 123.46 | 123.28 | 123.63 |

Table 7.6.1: QLBS Put option price for Put option on LVMH stock

For the B&S model we obtained the following results

| Maturity | QLBS Put Price |
|---|---|
| 4 months | 78.40 |
| 10 months | 94.94 |
| 16 months | 106.00 |
| 22 months | 114.25 |
| 28 months | 120.70 |

Table 7.6.2: B&S Put option price for Put option on LVMH stock

To compare the results of the QLBS model in table 7.6.1 with the results from B&S model in table 7.6.2, we compared the absolute error of both models using real put option prices on the LVMH stock that can been seen in the appendices A.1.2, A.1.3, A.1.4, A.1.5 and A.1.6. On figure 7.6.2 we can see that the absolute error of QLBS model and B&S are small which indicates that both model are good predictors for the options of interest. Moreover, both absolute erros are close to each other and the absolute error of QLBS is smaller than the absolute error of B&S for options with a maturity of 4 months, 10 months and 16 months. For options with maturity in 22 months and 28 months B&S has a smaller absolute error.

Based on our analysis we cannot easily determine whether the QLBS model is better than the B&S model and vice versa. However we can say that, the QLBS model achieves

[1] `https://fr.investing.com/rates-bonds/france-1-year-bond-yield`

good performance despite that we generated only 100 options prices in the experiment. Moreover, we fixed the risk aversion parameter to 0.00 to have a risk neutral option price but market conditions are not risk neutral. Therefore, adapting the risk aversion parameter may help to have better option price predictions.

To conclude, we can say that even though we cannot easily determine which model is better, we can say that QLBS model has promising results.

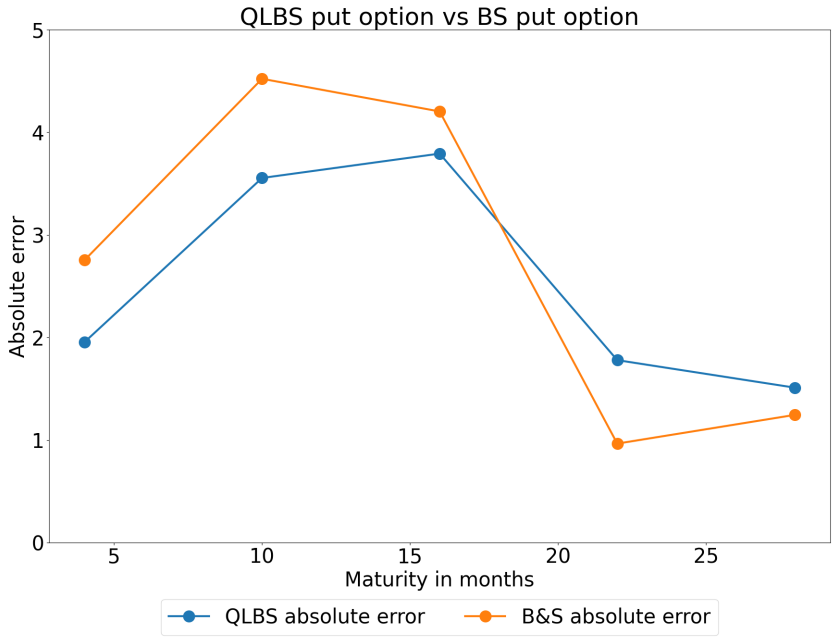



Figure 7.6.2: Absolute error of QLBS and B&S with respect to option maturity

# Chapter 8

# Conclusions

The goal of the thesis was to give an introduction to Markov Decision Processes and reinforcement learning and how they can be applied to real use cases. Before taking on the thesis, I knew very little, if anything, about the subject. Therefore, the work provides the reader with the necessary foundations to understand and deepen their knowledge of Markov processes and their implication and use in reinforcement learning.

In the second chapter of the thesis, we presented the basics of reinforcement learning. We presented the bandit problem to introduce a decision-making problem. We also briefly introduce quantities that are used frequently in reinforcement learning, which are cumulative reward and $q$-value. In that section, we explained how when an agent faces a bandit problem, it can use these quantities to maximize its chance to accomplish its goal. After we covered the effect of nonstationarity on decision-making problems and more particularly in the context of reinforcement learning. At the end of the chapter, we covered an important aspect of reinforcement learning: exploration versus exploitation. Indeed, the exploration versus exploitation trade-off greatly impacts the performance of the trained model. Therefore, establishing a strategy to manage exploration and exploitation when training a reinforcement learning algorithm is crucial to the model performances.

In the third chapter, we presented Markov Decision Processes. In that chapter, we covered how MDPs can be used to model decision-making problems and the different properties of MDPs. We also studied the relationship between MDPs and value functions. At the end of the chapter, we used an example to explain how MDPs can be used to solve decision-making problems.

Then, in the fourth chapter, we introduced dynamic programming and its use with Markov Decision Process and Reinforcement learning. Dynamic Programming is one of the foundational concepts in reinforcement learning. It provides a principled approach to solve Markov Decision Processes and finding optimal policies. It is important to note that it may not always be practical in large-scale and real-world problems due to the "curse of dimensionality". We presented two of the most popular reinforcement learning

algorithms based on Dynamic Programming: Policy Iteration and Value Iteration. That chapter presented the algorithms and their properties.

In the fifth chapter, we covered Monte Carlo Methods to solve reinforcement learning problems. We presented the Monte Carlo ES Algorithm and its properties. Monte Carlo ES is also one of the most famous algorithms in reinforcement learning. One of the greatest advantages of Monte Carlo methods and Monte Carlo ES is that they do not require knowledge of the transition probabilities and rewards. Instead, they learn directly from sampled experiences. Monte Carlo ES is a foundational algorithm to understand how to use Monte Carlo methods in the context of reinforcement learning.

In the sixth chapter, we applied Policy Iteration, Value Iteration and Monte Carlo ES on the famous board game tic-tac-toe. Based on our analysis, we can say that the three algorithms have shown promising results, especially the Policy Iteration algorithm. Indeed, among the three algorithms used, it was the one that had the best results. On the contrary, despite the promising results obtained with Monte Carlo ES, we could see room for improvement. Indeed, the policy obtained with the algorithm was not optimal for every move. Improvement can be made by simulating more games from which the agent can learn. Unfortunately, due to the limitations of our computing infrastructure and the computing resources required to run a large enough number of simulations to improve the results, we were unable to do so. Finally, results obtained from Value Iteration were also promising but not as good as the ones from Policy Iteration. Indeed, when we ran simulations of the three different policies against one another, we noticed that Policy Iteration has an advantage over Monte Carlo ES in every circumstance. For Value Iteration, that advantage was influenced by the first player of the game when it played against Monte Carlo ES.

In the seventh chapter, we started with a literature review on the application of reinforcement learning in finance. we presented the different work done in the literature on the topic and covered some results obtained. We could see in that part that promising work has already been done on the subject. Then, we covered the QLBS model and how it is used in finance. We had an introduction of the famous B&S model. Then we presented the QLBS model and how it uses value function and Bellman equations for option pricing. We also used a numerical example, with simulated data, to apply the QLBS model and the B&S model to price a put option. With that example we could compare QLBS put option price with the B&S put option price. We saw that QLBS put price and B&S put price were close. Based on the 95 % confidence interval of the QLBS put price could also conclude that the model had precise predictions. Finally, we had a final example using historical data from LVMH stock price evolution from August 2019 to August 2024. From that example we could not determine if the QLBS model were significantly better than B&S model but the results of QLBS model were promising.

Based on our work, one should now be able to understand Markov Decision Processes and how they are used in reinforcement learning. Moreover, we also saw that reinforcement

learning can be applied in finance, but it is important to know that in can also be applied in many other fields.

# Bibliography


K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

D. P. Bertsekas. *Dynamic Programming And Optimal Control*, volume 2. Athena Scientific, 2007.

D. P. Bertsekas and J. N Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996.

J. F. Carriere. Valuation of the early-exercise price for options using simulations and nonparametric regression. *Insurance: mathematics and Economics*, 19(1):19–30, 1996.

T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *introduction to algorithms*. MIT Press, 2nd edition, 2009.

T. G Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of artificial intelligence research*, 13:227–303, 2000.

I. Halperin. The qlbs q-learner goes nuqlear: fitted q iteration, inverse rl, and option portfolios, 2018. URL `https://arxiv.org/abs/1801.06077`.

I. Halperin. Qlbs: Q-learner in the black-scholes(-merton) worlds, 2019. URL `https://arxiv.org/abs/1712.04609`.

H. Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, volume 23, pages 2613 – 2621. Curran Associates, Inc., 2010. URL `https://proceedings.neurips.cc/paper_files/paper/2010/file/091d584fced301b442654dd8c23b3fc9-Paper.pdf`.

M. Hauskrecht. Value-function approximations for partially observable Markov decision processes. *Journal of artificial intelligence research*, 13:33–94, 2000.

J. K. Hunter and B. Nachtergaele. *Applied analysis*. World Scientific Publishing Company, 2001.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

M. G. Lagoudakis and R. Parr. Least-squares policy iteration. *The Journal of Machine Learning Research*, 4:1107–1149, 2003.

A. Lew and H. Mauch. *Dynamic programming: A computational tool*, volume 38. Springer, 2006.

F. L. Lewis, D. Vrabie, and V. L. Syrmos. *Optimal control*. John Wiley & Sons, 2012.

Y. Li. Reinforcement learning applications. *CoRR*, 2019. URL `http://arxiv.org/abs/1908.06973`.

Y. Li, C. Szepesvari, and D. Schuurmans. Learning exercise policies for american options. In *Artificial intelligence and statistics*, pages 352–359. PMLR, 2009.

R. A. Lockhart. Stat 870: Applied probability - course notes, September 2006.

James R Norris. *Markov chains*, volume 2. Cambridge university press, 1998.

M. L. Putterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley Sons, INC., 1994.

Leopold Schmetterer. Stochastic approximation. In *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics*, volume 4, pages 587–610. University of California Press, 1961.

S. Singh, R. L. Lewis, and Andrew G. Barto. Where do rewards come from. In *Proceedings of the annual conference of the cognitive science society*, pages 2601–2606. Cognitive Science Society, 2009.

S. P. Singh and R. S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine learning*, 22(1-3):123–158, 1996.

R. S. Sutton. On the significance of Markov decision processes. In *Artificial Neural Networks — ICANN'97*, pages 273–282, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

M. Tokic and G. Palm. Value-difference based exploration: adaptive control between epsilon-greedy and softmax. In *Annual conference on artificial intelligence*, pages 335–346. Springer, 2011.

J. N. Tsitsiklis and B. Van Roy. Regression methods for pricing complex American-style options. *IEEE Transactions on Neural Networks*, 12(4):694–703, 2001.

M. Van Otterlo and M. Wiering. Reinforcement learning and Markov decision processes. In *Reinforcement learning: State-of-the-art*, pages 3–42. Springer, 2012.

C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Oxford, 1989.

C. J. C H. Watkins and P. Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

R. J. Williams and L. C. Baird III. Analysis of some incremental variants of policy iteration: First steps toward understanding actor-critic learning systems. Technical report, Citeseer, 1993. URL `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=a4d262acad49ff3302a8a666da81088450769914`.

# Appendix A

# Appendix

## A.1 QLBS Model Application: supplementary figures

Log returns of LVMH stock

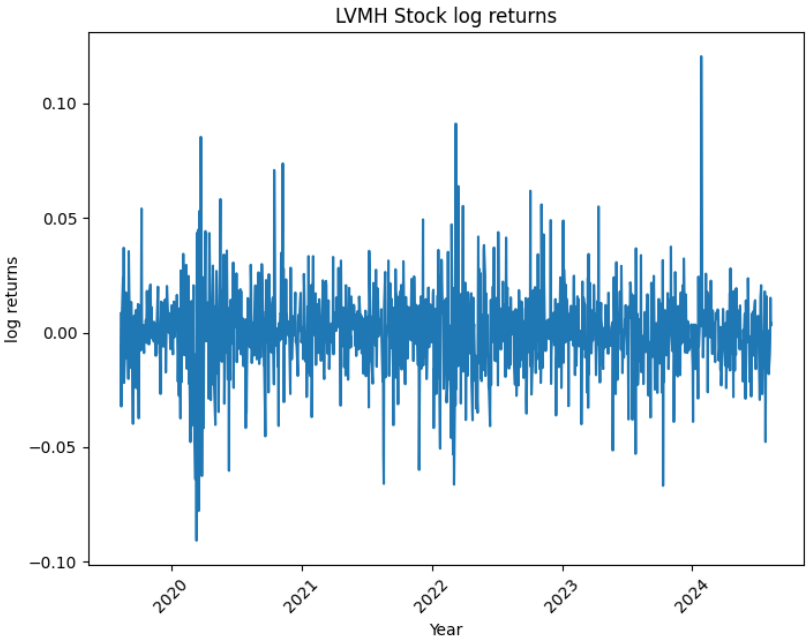
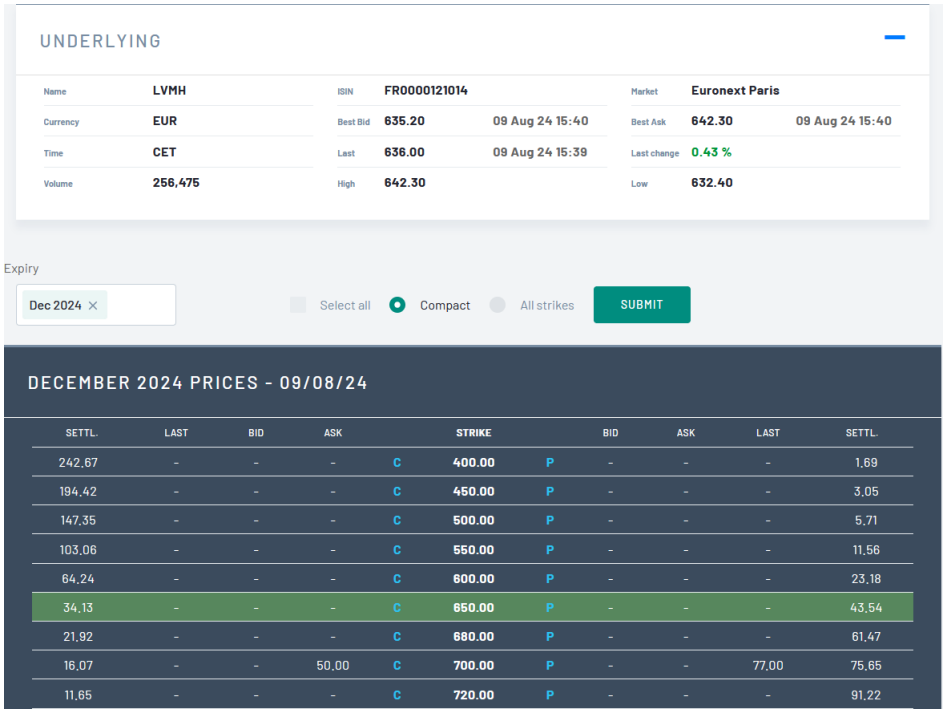



Figure A.1.1: Log return of LVMH stock from August 2019 to August 2024

LVMH option prices for option that expires in 4, 10, 16, 22, 28 months



UNDERLYING

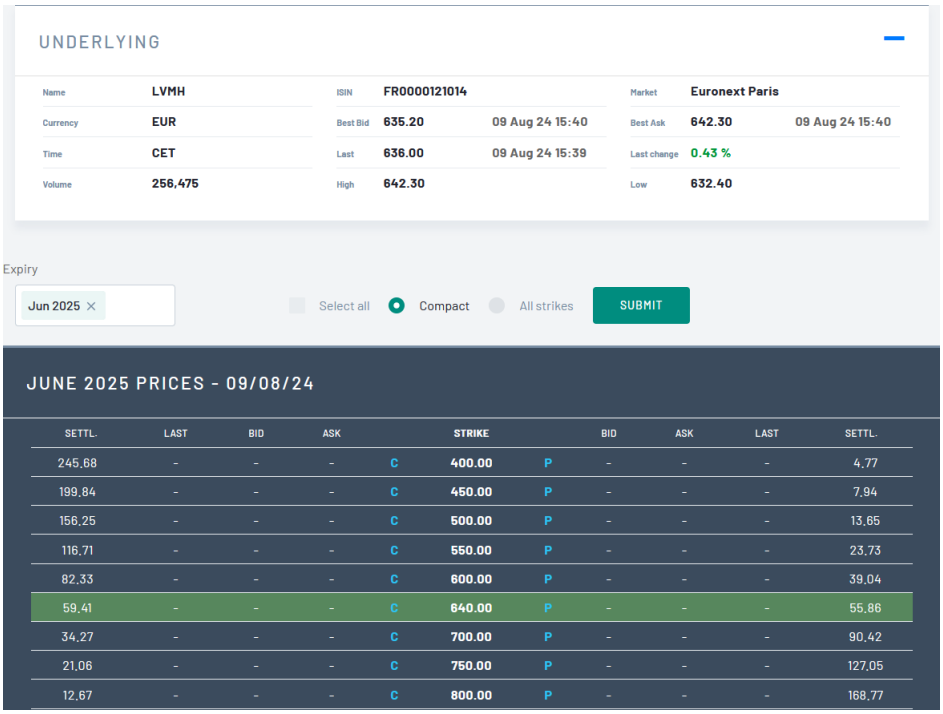
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Name | LVMH | ISIN | FR0000121014 | | Market | Euronext Paris | |
| Currency | EUR | Best Bid | 635.20 | 09 Aug 24 15:40 | Best Ask | 642.30 | 09 Aug 24 15:40 |
| Time | CET | Last | 636.00 | 09 Aug 24 15:39 | Last change | 0.43 % | |
| Volume | 256,475 | High | 642.30 | | Low | 632.40 | |

Expiry: Dec 2024 ×  Select all  Compact  All strikes  SUBMIT

DECEMBER 2024 PRICES - 09/08/24

| SETTL. | LAST | BID | ASK | | STRIKE | | BID | ASK | LAST | SETTL. |
|---|---|---|---|---|---|---|---|---|---|---|
| 242,67 | - | - | - | C | 400.00 | P | - | - | - | 1,69 |
| 194,42 | - | - | - | C | 450.00 | P | - | - | - | 3,05 |
| 147,35 | - | - | - | C | 500.00 | P | - | - | - | 5,71 |
| 103,06 | - | - | - | C | 550.00 | P | - | - | - | 11,56 |
| 64,24 | - | - | - | C | 600.00 | P | - | - | - | 23,18 |
| 34,13 | - | - | - | C | 650.00 | P | - | - | - | 43,54 |
| 21,92 | - | - | - | C | 680.00 | P | - | - | - | 61,47 |
| 16,07 | - | - | 50,00 | C | 700.00 | P | - | - | 77,00 | 75,65 |
| 11,65 | - | - | - | C | 720.00 | P | - | - | - | 91,22 |

Figure A.1.2: LVMH option prices for option with maturity in 4 months



UNDERLYING

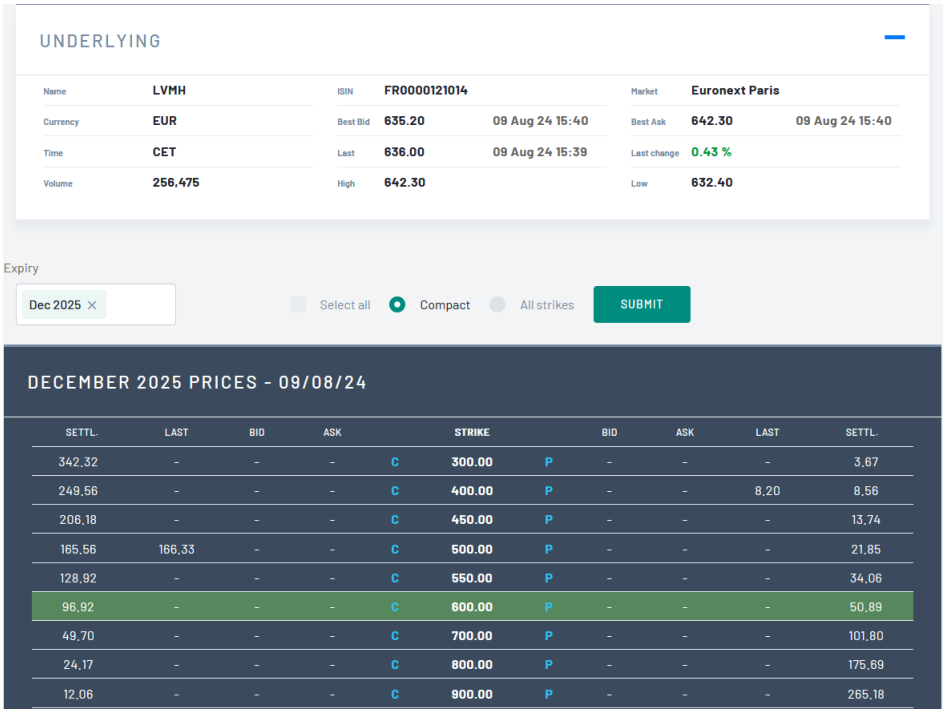
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Name | LVMH | ISIN | FR0000121014 | | Market | Euronext Paris | |
| Currency | EUR | Best Bid | 635.20 | 09 Aug 24 15:40 | Best Ask | 642.30 | 09 Aug 24 15:40 |
| Time | CET | Last | 636.00 | 09 Aug 24 15:39 | Last change | 0.43 % | |
| Volume | 256,475 | High | 642.30 | | Low | 632.40 | |

Expiry: Jun 2025 ×  Select all  Compact  All strikes  SUBMIT

JUNE 2025 PRICES - 09/08/24

| SETTL. | LAST | BID | ASK | | STRIKE | | BID | ASK | LAST | SETTL. |
|---|---|---|---|---|---|---|---|---|---|---|
| 245,68 | - | - | - | C | 400.00 | P | - | - | - | 4,77 |
| 199,84 | - | - | - | C | 450.00 | P | - | - | - | 7,94 |
| 156,25 | - | - | - | C | 500.00 | P | - | - | - | 13,65 |
| 116,71 | - | - | - | C | 550.00 | P | - | - | - | 23,73 |
| 82,33 | - | - | - | C | 600.00 | P | - | - | - | 39,04 |
| 59,41 | - | - | - | C | 640.00 | P | - | - | - | 55,86 |
| 34,27 | - | - | - | C | 700.00 | P | - | - | - | 90,42 |
| 21,06 | - | - | - | C | 750.00 | P | - | - | - | 127,05 |
| 12,67 | - | - | - | C | 800.00 | P | - | - | - | 168,77 |

Figure A.1.3: LVMH option prices for option with maturity in 4 months



UNDERLYING

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Name | LVMH | ISIN | FR0000121014 | | Market | Euronext Paris | |
| Currency | EUR | Best Bid | 635.20 | 09 Aug 24 15:40 | Best Ask | 642.30 | 09 Aug 24 15:40 |
| Time | CET | Last | 636.00 | 09 Aug 24 15:39 | Last change | 0.43 % | |
| Volume | 256,475 | High | 642.30 | | Low | 632.40 | |

Expiry: Dec 2025 ×  Select all  Compact  All strikes  SUBMIT

DECEMBER 2025 PRICES - 09/08/24

| SETTL. | LAST | BID | ASK | | STRIKE | | BID | ASK | LAST | SETTL. |
|---|---|---|---|---|---|---|---|---|---|---|
| 342,32 | - | - | - | C | 300.00 | P | - | - | - | 3,67 |
| 249,56 | - | - | - | C | 400.00 | P | - | - | 8,20 | 8,56 |
| 206,18 | - | - | - | C | 450.00 | P | - | - | - | 13,74 |
| 165,56 | 166,33 | - | - | C | 500.00 | P | - | - | - | 21,85 |
| 128,92 | - | - | - | C | 550.00 | P | - | - | - | 34,06 |
| 96,92 | - | - | - | C | 600.00 | P | - | - | - | 50,89 |
| 49,70 | - | - | - | C | 700.00 | P | - | - | - | 101,80 |
| 24,17 | - | - | - | C | 800.00 | P | - | - | - | 175,69 |
| 12,06 | - | - | - | C | 900.00 | P | - | - | - | 265,18 |

Figure A.1.4: LVMH option prices for option with maturity in 4 months

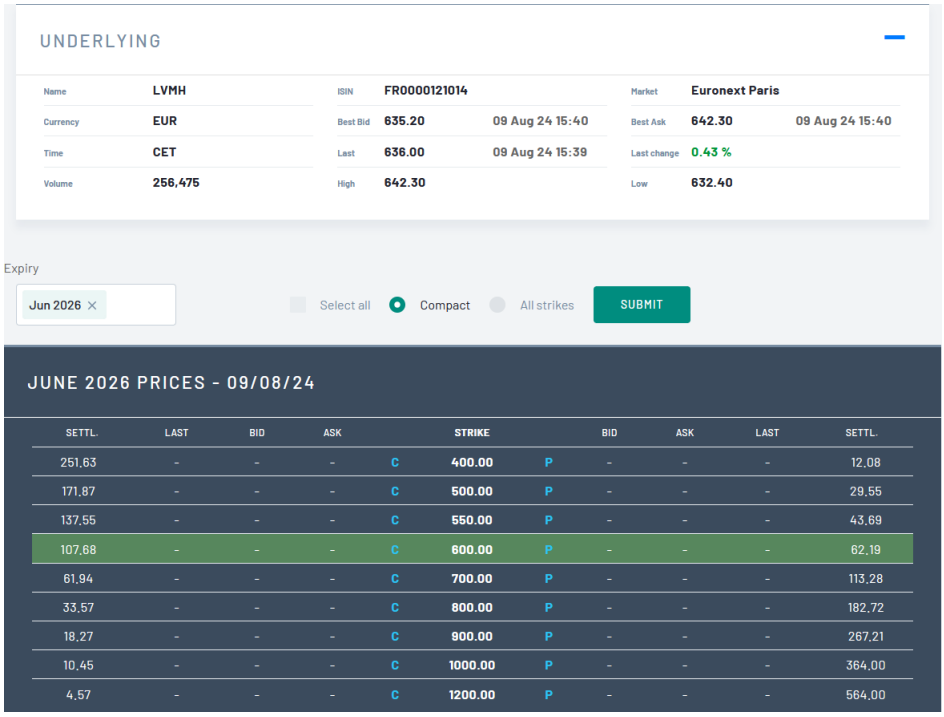



Figure A.1.5: LVMH option prices for option with maturity in 4 months

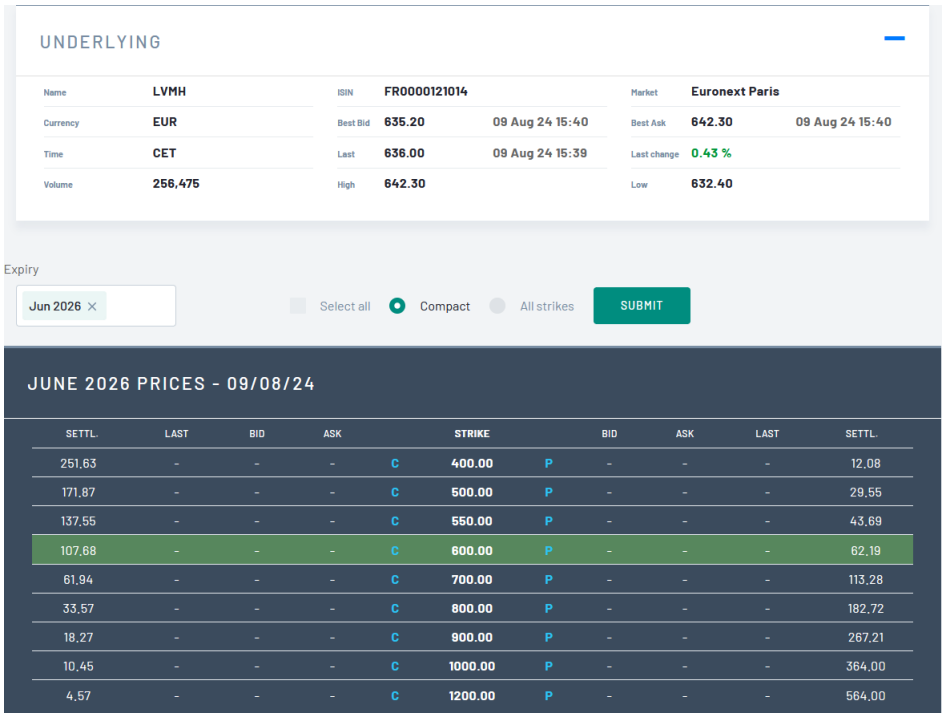



Figure A.1.6: LVMH option prices for option with maturity in 4 months

**UNIVERSITÉ CATHOLIQUE DE LOUVAIN**
**Faculté des sciences**
Place des Sciences, 2 bte L6.06.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/sc