Università di Genova | **DIBRIS** DIPARTIMENTO DI INFORMATICA, BIOINGEGNERIA, ROBOTICA E INGEGNERIA DEI SISTEMI

# From Dynamic Programming to Reinforcement Learning

**Computational Aspects**

Andrea Guarnore

Master Thesis

Università di Genova, DIBRIS Via Opera Pia, 13 16145 Genova, Italy
https://www.dibris.unige.it/

Università
di Genova

MSc Computer Science
Data Science and Engineering Curriculum

# From Dynamic Programming to Reinforcement Learning
Computational Aspects

Andrea Guarnore

Advisors: Alessandro Verri
Agnese Seminara

Examiner: Francesca Odone

September, 2022

# Abstract

Reinforcement learning theory ensures convergence in environments with finite state and action spaces. However, real-life environments often involve large and continuous state and action spaces, which require approximate methods able to learn states not yet visited. Unfortunately the theoretical guarantees for such methods are not as robust.

In this work we review the theory of the fundamental reinforcement learning methods. Then, we show how in practice, in the finite case, various methods and parameters can affect convergence rate. In the continuous case, we first present results obtained by discretizing the state space, then subsequently show how better results can be achieved with approximate methods, while also considering once again the role of the parameters while learning.

# Table of Contents

**1 Introduction** **1**

**2 Optimal control** **3**

2.1 Markov decision processes . . . 3

2.1.1 The agent–environment interaction . . . 3

2.1.2 Return and value . . . 4

2.1.3 Optimality criterion . . . 5

2.2 Bellman theory . . . 6

2.2.1 Bellman expectation equations . . . 6

2.2.2 Bellman expectation backup . . . 7

2.2.3 Bellman's principle of optimality . . . 8

2.3 Dynamic programming . . . 9

2.3.1 Policy evaluation . . . 9

2.3.2 Policy improvement . . . 10

2.3.3 Policy iteration . . . 11

2.3.4 Value iteration . . . 12

2.3.5 Efficiency of dynamic programming methods . . . 12

**3 Value-based reinforcement learning** **14**

3.1 Preliminary remarks . . . 14

3.1.1 Taxonomy of reinforcement learning algorithms . . . 14

3.1.2 Control in value-based reinforcement learning . . . . . . . . . . . . 15
3.2 Monte Carlo methods . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2.1 Monte Carlo prediction . . . . . . . . . . . . . . . . . . . . . . . 15
3.2.2 Monte Carlo control . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.3 Temporal difference learning . . . . . . . . . . . . . . . . . . . . . . . 18
3.3.1 Temporal difference prediction . . . . . . . . . . . . . . . . . . . 18
3.3.2 TD($\lambda$) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.3.3 Temporal difference control . . . . . . . . . . . . . . . . . . . . . 21
3.4 Value function approximation . . . . . . . . . . . . . . . . . . . . . . . 22
3.4.1 Stochastic gradient methods . . . . . . . . . . . . . . . . . . . . 23
3.4.2 Linear value function approximation . . . . . . . . . . . . . . . . 24
3.4.3 Radial basis functions . . . . . . . . . . . . . . . . . . . . . . . . 25
3.4.4 Prediction with function approximation . . . . . . . . . . . . . . 25
3.4.5 Control with function approximation . . . . . . . . . . . . . . . . 26

**4 Policy-based reinforcement learning** **28**
4.1 Policy gradient . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.1.1 Analytic computation . . . . . . . . . . . . . . . . . . . . . . . . 29
4.1.2 Policy gradient theorem . . . . . . . . . . . . . . . . . . . . . . . 30
4.2 Differentiable policies . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.2.1 Softmax policy . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.2.2 Gaussian policy . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.3 Basic policy gradient algorithms . . . . . . . . . . . . . . . . . . . . . 32
4.3.1 REINFORCE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.3.2 Actor-critic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.3.3 Comparison of value- and policy-based methods . . . . . . . . . . 35

**5 Empirical results** **36**

5.1 Domains . . . 36
5.1.1 Cliff walking . . . 36
5.1.2 Frozen lake . . . 37
5.1.3 Mountain car . . . 37
5.2 Finite state space experiments . . . 38
5.2.1 Policy and value iteration . . . 38
5.2.2 Scheduling strategies for $\varepsilon$ . . . 39
5.2.3 On- and off-policy learning . . . 41
5.2.4 The choice of $\alpha$ . . . 42
5.3 Continuous state space experiments . . . 42
5.3.1 Discretizing the state space . . . 43
5.3.2 Tuning the features . . . 44
5.3.3 The $\lambda$ parameter . . . 45
5.3.4 Comparison of approximate methods . . . 46
5.3.5 Continuous action space . . . 47

**6 Conclusions 50**

**Appendix A Implementation details 52**
A.1 Structure of the implementation . . . 52
A.2 OpenAI Gym . . . 53

**Bibliography 54**

# Chapter 1

# Introduction

Reinforcement learning (RL), along with supervised and unsupervised learning, is one of the three main machine learning paradigms. In RL an agent situated in an environment has to learn to behave optimally according to some notion of cumulative numerical reward. It differs from supervised learning, since learning is not guided by a set of examples labeled by a knowledgeable external supervisor, and it also differs from unsupervised learning since the agent does not seek hidden patterns in unlabeled data.

Part of RL can be seen as directly inspired from behaviorism, which attempts to understand human and animal behavior with respect to their response to stimuli that the environment provides. Likewise, RL concerns itself with how agents learn by iteratively interacting with an environment. In particular: the agent takes actions environment, then, in response the said action, the environment sends back to the agent a new representation of the state after the action, and a numerical reward associated with the action the agent took. Furthermore, one of the distinguishing characteristics of RL is the dilemma between exploration and exploitation which, in a sense, emulates learn by trial and error. In order to obtain a high reward, the agent has to choose actions which it has found to be effective in the past. However, in order to find such actions, it also has to try actions that it has not attempted previously.

More formally, the objective of the agent is that of finding an policy — a mapping from states to actions — which maximizes the expected cumulative reward, called value. In the case of finite state and action spaces, RL theory clearly establishes the conditions needed for each method to converge, while in the case of large or continuous state spaces, where approximate methods that update the value to states not yet visited are needed, only weaker convergence guarantees are given. In fact, when representing the value as linear combination of weights and features, the two most prominent value-based methods, SARSA and Q-learning, respectively converge near to the optimal solution and can exhibit instability. Alternatively, using a nonlinear representation, such as the more powerful neural networks, no convergence guarantees have been proven. In recent years interest has shifted towards a second class of methods, which instead directly optimize the policy. In general such policy-based methods benefit from better convergence properties, as well as the possibility to be employed with environments with continuous action spaces.

The aim of this work is to review the main RL methods, and compare the aforementioned theoretical results with empirical ones, for various classic environments. In Chapter 2 we introduce the theory of optimal control, upon which the theory of reinforcement learning is built. Then, by removing the assumption of perfect knowledge of the environment, we start exploring reinforcement learning theory. Chapters 3 and 4 respectively illustrate the most important model-free value- and policy-based reinforcement learning methods. In Chapter 5 we report and discuss some empirical results obtained by running some experiments on the methods presented in the previous chapters. In particular, in the case of finite state spaces, we show how the agent's farsightedness and level of exploration over time, but also the environment and the method itself can affect learning, both in terms of convergence rate and variance. In the case of continuous state spaces we first present results obtained by discretizing the state space, and then compare them with agents trained on featurized states, with different parameters. Ultimately, we also consider the more complex case in which both the state and the action space are continuous. Chapter 6 concludes the thesis by providing some remarks regarding possible future developments and other relevant aspects that were not discussed in this work. The code with which the experiments have been conducted has been published in a GitHub repository[1].

[1] `https://github.com/andreaguarnore/reinforcement-learning`

# Chapter 2

# Optimal control

Much of reinforcement can be seen as optimal control, albeit with less assumptions about the environment, and less computation. Because of this, we will start this chapter by briefly introducing the fundamental aspects of optimal control theory.

This chapter and the following two are loosely based on the second edition of Sutton and Barto's *Reinforcement Learning* [1] and Silver's lectures [2].

## 2.1 Markov decision processes

An optimal control agent is modeled to perform sequential decision-making by interacting with the environment. The environment is usually defined as an infinite-horizon Markov decision process (MDP), referred to as Markov decision process from now on. Note that are there several extensions and generalizations, accounting for continuous time, or partial observability of the state. For ease of exposition we will just restrict ourselves to the most common case.

### 2.1.1 The agent–environment interaction

A Markov decision process is defined as a tuple $(\mathcal{S}, \mathcal{A}, p, p_0, r, \gamma)$, where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $p$ is a transition function, $p_0$ is a probability distribution of initial states, $r$ is a reward function, and $\gamma \in [0, 1]$ is a discount factor, whose role will be explained later. For now, we will only consider finite MDPs, i.e., MDPs with a finite set of states and actions.

Markov decision processes satisfy the Markov property, which assumes that future states and rewards depend only on the current state and action. Because of this, we can define the transition function as

$$p\left(s'|s,a\right) \doteq \Pr\left(s_{t+1}=s'|s_t=s,a_t=a\right), \tag{2.1}$$

where $t \in \mathbb{N}_0$ denotes the time step. The initial distribution $p_0$ is defined as

$$p_0\left(s\right) = \Pr\left(s_0=s\right). \tag{2.2}$$

The reward function defines the expected immediate reward when action $a$ is taken in state $s$, i.e.,

$$r\left(s,a\right) = \mathbb{E}\left[r_{t+1}|s_t=s,a_t=a\right]. \tag{2.3}$$

The behavior of the agent is described by its policy $\pi$, a mapping from states to actions, which defines the probability of taking action $a$ in a given state $s$:

$$\pi\left(a|s\right) \doteq \Pr\left(a_t=a|s_t=s\right). \tag{2.4}$$

The policy can be either deterministic or stochastic. Sometimes, we will use $\pi\left(s\right)$ as a shorthand notation for deterministic policies.

At each of a sequence of time steps $t$, the environment is in some state $s_t$. The agent chooses and performs an action, $a_t \sim \pi\left(\cdot|s_t\right)$. Thus, the agent receives the new state of the environment, $s_{t+1} \sim p\left(\cdot|s_t,a_t\right)$, and the reward associated to the state transition, $r_{t+1}$, given by $r\left(s_t,a_t\right)$.

MDPs are called episodic if there is at least one state in which the process terminates. In this case, we denote the last time step as $T$. Once a process terminates a new episode starts. We call a trajectory the sequence of random variables until a terminal state is reached.

### 2.1.2 Return and value

The return $G_t$ is defined as the discounted sum of rewards starting at time step $t$,

$$G_t \doteq r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{2.5}$$

As $\gamma$ gets smaller, the agent ignores farther-away rewards, and vice versa. Additionally, we can split the return in two parts: the immediate reward, and the discounted return at the next time step, as follows:

$$\begin{aligned} G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \\ &= r_{t+1} + \gamma\left(r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + \ldots\right) \\ &= r_{t+1} + \gamma G_{t+1}. \end{aligned} \tag{2.6}$$

If $\gamma < 1$, the infinite sum has a finite value only if the sequence of rewards $\{r_k\}$ is bounded. We admit $\gamma = 1$ only for episodic MDPs.

We define the state-value function, $V^{\pi}(s)$, as the expected discounted return when starting in state $s$, and following policy $\pi$ thereafter,

$$V^{\pi}(s) \doteq \mathbb{E}_{\pi}\left[G_t | s_t = s\right], \tag{2.7}$$

where $\mathbb{E}_{\pi}[\cdot]$ indicates the expected value of a random variable given that the agent follows policy $\pi$ at each time step. We will however drop the subscript $\pi$ to simplify notation going forward. Similarly, we define the value of being in state $s$, taking action $a$, and following policy $\pi$ thereafter, as follows:

$$Q^{\pi}(s, a) \doteq \mathbb{E}_{\pi}\left[G_t | s_t = s, a_t = a\right]. \tag{2.8}$$

We call $Q^{\pi}(s, a)$ the action-value function.

### 2.1.3 Optimality criterion

We define a partial ordering over policies:

$$\pi \geq \pi' \iff V^{\pi}(s) \geq V^{\pi'}(s), \text{ for all } s. \tag{2.9}$$

A policy which is better than or equal to all other policies is called an optimal policy. All optimal policies are denoted as $\pi^*$. Following an optimal policy yields both the optimal state-value, $V^*$,

$$V^{\pi^*}(s) = V^*(s) \doteq \max_{\pi} V^{\pi}(s), \text{ for all } s, \tag{2.10}$$

and the optimal action-value, $Q^*$,

$$Q^{\pi^*}(s, a) = Q^*(s, a) \doteq \max_{\pi} Q^{\pi}(s, a), \text{ for all } s, a. \tag{2.11}$$

Both the optimal state- and optimal action-value functions specify the best possible performance obtainable in an MDP. Hence, they are the solution to the MDP. The problem of finding the optimal value function, and a corresponding optimal policy, is known as the control problem. Over the course of the remaining sections of the chapter, we will a see way in which the control problem can be solved when having perfect knowledge of the environment.

## 2.2 Bellman theory

In this section we introduce the Bellman equations [3], which express a relationship between the value of the current state (or state-action pair) and the value of the following state (or state-action pair).

### 2.2.1 Bellman expectation equations

The state-value function can be rewritten in terms of the action-value function, as follows:

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}\left[G_t | s_t = s\right] \\ &\overset{(a)}{=} \sum_a \mathbb{E}\left[G_t | s, a\right] \Pr(a|s) \\ &\overset{(b)}{=} \sum_a \pi(a|s) Q^{\pi}(s,a), \end{aligned} \tag{2.12}$$

where (a) follows by the law of total expectation, and (b) follows from (2.4) and (2.8). Similarly, the action-value function can be rewritten in terms of the state-value function,

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}\left[G_t | s, a\right] \\ &\overset{(a)}{=} \mathbb{E}\left[r_{t+1} | s, a\right] + \gamma \mathbb{E}\left[G_{t+1} | s, a\right] \\ &\overset{(b)}{=} r(s,a) + \gamma \sum_{s'} \mathbb{E}\left[G_{t+1} | s', s, a\right] \Pr(s'|s,a) \\ &\overset{(c)}{=} r(s,a) + \gamma \sum_{s'} \mathbb{E}\left[G_{t+1} | s'\right] \Pr(s'|s,a) \\ &\overset{(d)}{=} r(s,a) + \gamma \sum_{s'} p(s'|s,a) V^{\pi}(s') \end{aligned} \tag{2.13}$$

where (a) follows from (2.6), (b) follows from (2.3) and the law of total expectation, (c) follows by the Markov property, and (d) follows follows from (2.1) and (2.7). Ultimately, by plugging (2.13) into (2.12), we obtain a definition of the state-value function in a state in terms of the state-value function in the next state:

$$V^{\pi}(s) = \sum_a \pi(a|s) \left( r(s,a) + \gamma \sum_{s'} p(s'|s,a) V^{\pi}(s') \right). \tag{2.14}$$

Likewise, if we plug (2.12) into (2.13), we obtain a definition of the action-value function in a state-action pair in terms of the next state-action pair:

$$Q^{\pi}(s,a) = r(s,a) + \gamma \sum_{s'} p(s'|s,a) \pi(a'|s') Q^{\pi}(s',a'). \tag{2.15}$$

Equations (2.14) and (2.15) are the Bellman expectation equations for $V^\pi$ and $Q^\pi$. In both cases, we can interpret the value as the immediate reward plus the discounted value of the following state (or state-action pair), averaged over all possibilities, which depend on both the dynamics of the environment and the policy that the agent is following.

### 2.2.2 Bellman expectation backup

We define the operator $B^\pi : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$ for the policy $\pi$ as

$$\begin{aligned}(B^\pi U)(s) &\doteq \sum_a \pi(a|s) \left( r(s,a) + \gamma \sum_{s'} p(s'|s,a)\, U(s') \right) \\ &= r^\pi(s,a) + \gamma \sum_{s'} p^\pi(s'|s,a)\, U(s'), \end{aligned} \tag{2.16}$$

where $r^\pi$ and $p^\pi$ are given by

$$\begin{aligned} r^\pi(s) &\doteq \sum_a \pi(a|s)\, r(s,a), \\ p^\pi(s'|s) &\doteq \sum_a \pi(a|s)\, p(s'|s,a). \end{aligned} \tag{2.17}$$

We call $B^\pi$ the Bellman expectation backup operator. The name derives from the fact that the operator transfers value information back to a state from its successor states.

We now show that applying the Bellman expectation backup operator brings value functions closer by at least $\gamma$:

$$\begin{aligned} |(B^\pi U_1)(s) - (B^\pi U_2)(s)| &= \gamma \left| \sum_{s'} p^\pi(s'|s)\,(U_1(s') - U_2(s')) \right| \\ &\overset{(a)}{\leq} \gamma \sum_{s'} p^\pi(s'|s)\, |U_1(s') - U_2(s')| \\ &\leq \gamma \sum_{s'} p^\pi(s'|s) \max_{s''} |U_1(s'') - U_2(s'')| \\ &\overset{(b)}{=} \gamma \sum_{s'} p^\pi(s'|s)\, ||U_1 - U_2||_\infty \\ &\overset{(c)}{=} \gamma\, ||U_1 - U_2||_\infty . \end{aligned} \tag{2.18}$$

where (a) follows from $p^\pi(s'|s) > 0$, for all $s$, (b) follows by the definition of uniform norm, and (c) follows from $\sum_{s'} p^\pi(s'|s) = 1$, for all $s$.

As (2.18) is true for all $s$ we can conclude that

$$||B^{\pi}U_1 - B^{\pi}U_2||_{\infty} \leq \gamma \, ||U_1 - U_2||_{\infty} \, . \tag{2.19}$$

Hence, $B^{\pi}$ is a contraction map and it has a unique fixed point. Also, from (2.16) it follows that $\left(B^{\pi}V^{\pi}\right)(s) = V^{\pi}(s)$ for all $s$. Thus, $V^{\pi}$ is a fixed point of $B^{\pi}$, which implies that $V^{\pi}$ is the unique fixed point of $B^{\pi}$.

### 2.2.3 Bellman's principle of optimality

Bellman's principle of optimality [3, Chap. III.3] states that:

> "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."

This allows us to write the optimal state-value function as

$$\begin{aligned} V^{*}(s) &\doteq \max_{\pi} V^{\pi}(s) \\ &= \max_{a} \left( r(s,a) + \gamma \sum_{s'} p(s'|s,a) \, V^{*}(s') \right), \end{aligned} \tag{2.20}$$

and the action-value function as

$$\begin{aligned} Q^{*}(s,a) &\doteq \max_{\pi} Q^{\pi}(s,a) \\ &= r(s,a) + \gamma \sum_{s'} p(s'|s,a) \max_{a'} Q^{*}(s',a') \, . \end{aligned} \tag{2.21}$$

Equations (2.20) and (2.21) are the Bellman optimality equations for $V^{*}$ and $Q^{*}$. Often they are simply referred to as the Bellman equations. We will not go through the full derivation, nor the proof of their existence and uniqueness, as they require a fair amount of work. As before, we can express the state-value function with respect to the action-value function, and vice versa, as follows:

$$\begin{aligned} V^{*}(s) &= \max_{a} Q^{*}(s,a) \, , \\ Q^{*}(s,a) &= r(s,a) + \gamma \sum_{s'} p(s'|s,a) \, V^{*}(s') \, . \end{aligned} \tag{2.22}$$

We also define the Bellman optimality backup operator $B^{*}$ as

$$\left(B^{*}U\right)(s) \doteq \max_{a} \left( r(s,a) + \gamma \sum_{s'} p(s'|s,a) \, U(s') \right). \tag{2.23}$$

The Bellman optimality backup operator has analogous properties to those of the Bellman expectation backup operator. In fact, it is a contraction map, and its unique fixed point is $V^*$.

## 2.3 Dynamic programming

In this section we will see how MDPs can be solved by using the Bellman equations defined in the previous section as update rules. This class of methods are known as dynamic programming (DP) [3] methods.

### 2.3.1 Policy evaluation

We will first consider the problem of computing the value of a given policy $\pi$, $V^\pi$. This problem is known as policy evaluation, since the objective is to evaluate the policy. It is also sometimes called the prediction problem. Since the Bellman expectation equation for $V^\pi$ (2.14) is linear, and we assume to be in the case of finite MDPs, the prediction problem can be solved directly through matrix inversion. Unfortunately, matrix inversion has a complexity of $O\left(|\mathcal{S}|^3\right)$. Thus, it is too costly for large state spaces.

As mentioned already, we can instead turn the Bellman expectation equation for $V^\pi$ (2.14) into an iterative update rule, as follows:

$$\begin{aligned} V_0\left(s\right) &\leftarrow 0 \\ V_{k+1}\left(s\right) &\leftarrow \sum_a \pi\left(a|s\right)\left(r\left(s,a\right) + \gamma \sum_{s'} p\left(s'|s,a\right) V_k\left(s'\right)\right). \end{aligned} \tag{2.24}$$

Since update rule for $V_{k+1}$ is the application of the Bellman expectation backup operator, which we know to be a contraction map with $V^\pi$ as its unique fixed point, this method converges to $V^\pi$ as $k \to \infty$. This method is called iterative policy evaluation. The pseudo-code is presented in Algorithm 1.

**Algorithm 1:** Iterative policy evaluation, estimating $V \approx V^{\pi}$

```
function PolicyEvaluation $(\pi, \theta)$
    $V(s) \leftarrow 0$, $V'(s) \leftarrow \infty$, for all $s$
    while $||V - V'||_{\infty} > \theta$ do
        $V' \leftarrow V$
        $V(s) \leftarrow \sum_a \pi(a|s)\left(r(s,a) + \gamma \sum_{s'} p(s'|s) V'(s')\right)$, for all $s$
    return $V$
```

### 2.3.2 Policy improvement

Once a policy has been evaluated, we can generate a better (deterministic) policy, $\pi'$, by acting greedily with respect to the computed value $V^{\pi}$,

$$\begin{aligned} \pi'(s) &\doteq \arg\max_a Q^{\pi}(s,a) \\ &= \arg\max_a \left( r(s,a) + \gamma \sum_{s'} p(s'|s,a) V^{\pi}(s') \right). \end{aligned} \tag{2.25}$$

Clearly, $\pi'$ improves the value from any state over one step. We can formalize the improvement as the following inequality:

$$Q^{\pi}(s, \pi'(s)) = \max_a Q^{\pi}(s,a) \geq Q^{\pi}(s, \pi(s)) = V^{\pi}(s). \tag{2.26}$$

We will now prove that the value also improves over all future steps, by applying (2.26) to each one the future steps, as follows:

$$\begin{aligned} V^{\pi}(s) &\leq Q^{\pi}(s, \pi'(s)) \\ &= \mathbb{E}\left[r_{t+1} + \gamma V^{\pi}(s_{t+1}) | s_t = s, a_t = \pi'(s)\right] \\ &= \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma V^{\pi}(s_{t+1}) | s_t = s\right] \\ &\leq \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma Q^{\pi}(s_{t+1}, \pi'(s_{t+1})) | s_t = s\right] \\ &= \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma \mathbb{E}\left[r_{t+2} + \gamma V^{\pi}(s_{t+2}) | s_{t+1}, a_{t+1} = \pi'(s_{t+1})\right] | s_t = s\right] \\ &= \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 V^{\pi}(s_{t+2}) \middle| s_t = s\right] \\ &\leq \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 V^{\pi}(s_{t+3}) \middle| s_t = s\right] \\ &\vdots \\ &\leq \mathbb{E}_{\pi'}\left[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \middle| s_t = s\right] \\ &= V^{\pi'}(s). \end{aligned} \tag{2.27}$$

If improvement stops, we have that

$$Q^{\pi}\left(s, \pi'\left(s\right)\right) = \max_{a} Q^{\pi}\left(s, a\right) = Q^{\pi}\left(s, \pi\left(s\right)\right) = V^{\pi}\left(s\right), \tag{2.28}$$

which satisfies the Bellman optimality equation, since

$$V^{\pi}\left(s\right) = \max_{a} Q^{\pi}\left(s, a\right). \tag{2.29}$$

Hence, improvement stops only when $V^{\pi} = V^{*}$. This method is called policy improvement. The pseudo-code is outlined in Algorithm 2.

**Algorithm 2:** Policy improvement, computing $\pi$ by acting greedily w.r.t. $V$

```
function PolicyImprovement ($V$)
    $\pi(s) \leftarrow \arg\max_a \left(r(s,a) + \gamma \sum_{s'} p(s'|s,a) V(s')\right)$, for all $s$
    return $\pi$
```

## 2.3.3 Policy iteration

Policy iteration [4] is an iterative algorithm which interleaves policy evaluation with policy improvement, until convergence to an optimal policy is reached. At each step, the value is approximated to the true value of the current policy, while the policy is improved towards optimality. The pseudo-code of this algorithm is shown in Algorithm 3.

**Algorithm 3:** Policy iteration, estimating $\pi \approx \pi^{*}$

```
function PolicyIteration ($\theta$)
    $\pi \leftarrow$ initialized randomly
    while true do
        $V \leftarrow$ PolicyEvaluation($\pi, \theta$)
        $\pi' \leftarrow$ PolicyImprovement($V$)
        if $\pi'(s) = \pi(s)$ for all $s$ then
            return $\pi$                                        (reached optimal policy)
        $\pi \leftarrow \pi'$
```

The sequence of steps in policy iteration can be represented in the following way:

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi^{*} \xrightarrow{E} V^{\pi^{*}},$$

where $\xrightarrow{E}$ denotes a step of policy evaluation, and $\xrightarrow{I}$ denotes a step of policy iteration. Almost all reinforcement learning methods can be described through this framework, which takes the name generalized policy iteration (GPI).

### 2.3.4 Value iteration

The control problem can alternatively be solved by using the Bellman optimality equation for $V^*$ (2.20), instead of the Bellman expectation equation for $V^\pi$ (2.14). The equation is non-linear, meaning that it cannot be solved directly through matrix inversion. However, we can once again turn it into an iterative update rule, as follows:

$$\begin{aligned} V_0(s) &\leftarrow 0 \\ V_{k+1}(s) &\leftarrow \max_a \left( r(s,a) + \gamma \sum_{s'} p(s'|s,a) V_k(s') \right). \end{aligned} \tag{2.30}$$

This method is called value iteration [3]. Intuitively, this update rule can be seen as both policy evaluation and policy improvement in one. Once again, since the update rule for $V_{k+1}$ is the application of the Bellman optimality backup operator, we know that it will converge to $V^*$. Once the optimal value has been estimated, the optimal policy can be found by acting greedily with respect to it, as per (2.25). The pseudo-code is given in Algorithm 4.

**Algorithm 4:** Value iteration, estimating $\pi \approx \pi^*$

```
function ValueIteration ($\theta$)
    $V(s) \leftarrow 0$, $V'(s) \leftarrow \infty$, for all $s$
    while $||V - V'||_\infty > \theta$ do
        $V' \leftarrow V$
        $V(s) \leftarrow \max_a (r(s,a) + \gamma \sum_{s'} p(s'|s,a) V'(s'))$, for all $s$
    $\pi(s) \leftarrow \arg\max_a (r(s,a) + \gamma \sum_{s'} p(s'|s,a) V(s'))$, for all $s$
    return $\pi$
```

### 2.3.5 Efficiency of dynamic programming methods

Up until now, we based our algorithms on state-value functions, which have complexity $O\left(|\mathcal{S}|^2 |\mathcal{A}|\right)$ per iteration, for both policy and value iteration. The same ideas can also be easily applied to action-value functions, which instead result in a complexity of $O\left(|\mathcal{S}|^2 |\mathcal{A}|^2\right)$ per iteration.

The aptly named curse of dimensionality [3] describes one of the main problems which limits the applicability of dynamic programming methods: the number of states grows exponentially with the number of state variables, possibly rendering even a single Bellman update too expensive. Nevertheless, thanks to the capabilities of modern computers, dynamic programming can still be used effectively in problems with millions of states.

Dynamic programming methods are still very efficient when compared to other methods for solving MDPs. In fact, they are guaranteed to find an optimal policy in polynomial time even though the space of all possible deterministic policies has dimension $|\mathcal{S}|^{|\mathcal{A}|}$. In practice, both policy iteration and value iteration are used, with no particular evidence regarding which one is best.

Starting from the next chapter, we will see reinforcement learning methods. Reinforcement learning methods also attempt to solve MDPs, but do so without relying on the unrealistic assumption of perfect knowledge of the model of the environment. Either because it is actually not known, or because it would be too expensive to do full-width backups.

# Chapter 3

# Value-based reinforcement learning

Starting from this chapter, we will remove the assumption of perfect knowledge of the dynamics of the environment, i.e., the transition function $p$ and the reward function $r$, although we will still be able to sample from these functions. By doing so, we start to delve into reinforcement learning.

## 3.1 Preliminary remarks

Prior to introducing any reinforcement learning methods, we will make some remarks regarding the different approaches that can be taken to solve the control problem in the reinforcement learning framework, and how removing the knowledge of both $p$ and $r$ forces us to use action-value functions.

### 3.1.1 Taxonomy of reinforcement learning algorithms

Devising a complete and orderly taxonomy of reinforcement learning methods is not easy, but we can roughly divide them into three categories: 1) model-free value-based methods, that estimate the optimal action-value function, from which an optimal policy is implicitly derived; 2) model-free policy-based methods, which directly estimate an optimal policy; 3) model-based value-based methods, which learn the model, then apply methods such as the ones presented in the previous chapter.

We will focus exclusively on model-free methods, mainly because such methods require much less computation.

### 3.1.2 Control in value-based reinforcement learning

In this chapter, we will introduce the fundamental model-free value-based methods. Once the optimal value has been estimated, a deterministic optimal policy can be found by simply maximizing over it, in the same way we did for value iteration (Section 2.3.4).

Another important point to note regarding this class of methods is the use of the action-value function for control, rather than the state-value function. This is not by choice. In fact, a model is required to improve the policy with respect to the state-value function,

$$\pi'(s) = \arg\max_a \left( r(s,a) + \gamma \sum_{s'} p(s'|s,a) V^\pi(s') \right), \tag{3.1}$$

whereas a policy improvement step over the action-value function is the following:

$$\pi'(s) = \arg\max_a Q^\pi(s,a), \tag{3.2}$$

making it model-free.

## 3.2 Monte Carlo methods

All reinforcement learning problems deal with the absence of a model by learning from experience, rather than exploring all possibilities. In particular, Monte Carlo (MC) methods use the empirical average returns, each computed over a sampled episode, as an estimate of the expected return, which is the value function (2.7). Note that in order to compute the average return, episodes must terminate. Because of this, MC methods can only be applied to episodic MDPs.

### 3.2.1 Monte Carlo prediction

We will start by solving the prediction problem. There are two approaches to do so: first-visit MC and every-visit MC [5]. First-visit MC estimates the value of a state as the average of the returns following the first visit to the state. So, when a state is first visited in an episode, a single empirical return is computed, regardless if the state appears in the episode again. Conversely, every-visit MC estimates the value of a state as the average of the returns following all visits to the state. In other words, in this case, an empirical return is computed for each of the visits to the state in the episode.

Due to the law of large numbers, the averages of the estimates converge to their expected value, $V^\pi$. The two methods have similar theoretical properties, but we will focus only on first-visit MC. Algorithm 5 shows the pseudo-code for this case, while also using a moving average.

**Algorithm 5:** First-visit Monte Carlo prediction, estimating $V \approx V^{\pi}$

```
function FirstVisitMCPrediction ($\pi$)
    $N(s) \leftarrow 0$, $V(s) \leftarrow 0$, for all $s$
    loop
        Sample an episode following $\pi$: $(s_0, a_0, r_1, \dots, s_{T-1}, a_{T-1}, r_T)$
        foreach $s_t$ do
            if $s_t \notin \{s_0, s_1, \dots, s_{t-1}\}$ then
                $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
                $N(s_t) \leftarrow N(s_t) + 1$
                $V(s_t) \leftarrow V(s_t) + \frac{G_t - V(s_t)}{N(s_t)}$
    return $V$
```

### 3.2.2 Monte Carlo control

We can now apply the concept of generalized policy iteration by combining Monte Carlo prediction with policy improvement. Recall that however reinforcement learning methods do not have perfect knowledge of the dynamics of the system. Thus, it is not clear how improving the policy on an estimate of the value function would monotonically improve the policy as in the model-based case. In other words, we have to ensure exploration of all state-action pairs, even if suboptimal with respect to the current estimate of the value function.

One of the simplest, yet very effective, approaches to solve this problem is that of $\varepsilon$-greedy policies. Perhaps counter-intuitively given the name, $\varepsilon$-greedy policies choose the greedy action with probability $1 - \varepsilon$, and with probability $\varepsilon$ they choose an action at random. Formally,

$$\pi(a|s) = \begin{cases} \varepsilon/|\mathcal{A}| + 1 - \varepsilon & \text{if } a = \arg\max_b Q^{\pi}(s, b) \\ \varepsilon/|\mathcal{A}| & \text{otherwise.} \end{cases} \tag{3.3}$$

As in the case of greedy policy improvement, we can prove that for any $\varepsilon$-greedy policy $\pi$, the $\varepsilon$-greedy policy $\pi'$ improved with respect to $Q^{\pi}$ assures that $V^{\pi'}(s) \geq V^{\pi}(s)$, for all $s$.

In fact, we have that

$$\begin{aligned}
Q^{\pi}\left(s, \pi'\left(s\right)\right) &= \sum_{a} \pi'\left(a|s\right) Q^{\pi}\left(s,a\right) \\
&\overset{(a)}{=} \frac{\varepsilon}{|\mathcal{A}|} \sum_{a} Q^{\pi}\left(s,a\right) + \left(1-\varepsilon\right) \max_{a} Q^{\pi}\left(s,a\right) \\
&= \frac{\varepsilon}{|\mathcal{A}|} \sum_{a} Q^{\pi}\left(s,a\right) + \left(1-\varepsilon\right) \max_{a} Q^{\pi}\left(s,a\right) \frac{1-\varepsilon}{1-\varepsilon} \\
&\overset{(b)}{=} \frac{\varepsilon}{|\mathcal{A}|} \sum_{a} Q^{\pi}\left(s,a\right) + \left(1-\varepsilon\right) \max_{a} Q^{\pi}\left(s,a\right) \sum_{a} \frac{\pi\left(a|s\right) - \frac{\varepsilon}{|\mathcal{A}|}}{1-\varepsilon} \\
&\geq \frac{\varepsilon}{|\mathcal{A}|} \sum_{a} Q^{\pi}\left(s,a\right) + \sum_{a} \left(\pi\left(a|s\right) - \frac{\varepsilon}{|\mathcal{A}|}\right) Q^{\pi}\left(s,a\right) \\
&= \frac{\varepsilon}{|\mathcal{A}|} \sum_{a} Q^{\pi}\left(s,a\right) - \frac{\varepsilon}{|\mathcal{A}|} \sum_{a} Q^{\pi}\left(s,a\right) + \sum_{a} \pi\left(a|s\right) Q^{\pi}\left(s,a\right) \\
&= \sum_{a} \pi\left(a|s\right) Q^{\pi}\left(s,a\right) \\
&\overset{(c)}{=} V^{\pi}\left(s\right). \qquad (3.4)
\end{aligned}$$

where (a) follows from the fact that with probability $\varepsilon$ we choose an action randomly, and with probability $(1-\varepsilon)$ we choose the greedy action with respect to $Q^{\pi}$, (b) follows from $1-\varepsilon = \sum_a \left(\pi\left(a|s\right) - \frac{\epsilon}{|\mathcal{A}|}\right)$, and (c) follows from (2.12). Algorithm 6 gives a formulation of first-visit Monte Carlo control in pseudo-code.

**Algorithm 6:** First-visit Monte Carlo control, estimating $\pi \approx \pi^*$

```
function FirstVisitMC (·)
    $N(s,a) \leftarrow 0$, $Q(s,a) \leftarrow 0$, for all $s$, $a$
    loop
        Sample an episode following $\pi$: $(s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T)$      ($\varepsilon$-greedy)
        foreach $(s_t, a_t)$ do
            if $(s_t, a_t) \notin \{(s_0,a_0), (s_1,a_1), \ldots, (s_{t-1},a_{t-1})\}$ then
                $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
                $N(s_t,a_t) \leftarrow N(s_t,a_t) + 1$
                $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \frac{G_t - Q(s_t,a_t)}{N(s_t,a_t)}$
    return $\pi$
```

A formal guideline to schedule the value of $\varepsilon$ for $\varepsilon$-greedy policies is given by the greedy in the limit with infinite exploration (GLIE) theorem [6]. A sequence of policies $\{\pi_k\}$ is called GLIE if it satisfies the following conditions:

1. All state-action pairs should be explored infinitely many times,

   $$\lim_{k\to\infty} N_k(s,a) = \infty, \tag{3.5}$$

   where $N_k(s,a)$ denotes the times the state-action pair $(s,a)$ has been visited up to and including episode $k$.

2. The policy has to converge to a greedy policy as the number of episodes goes to infinity,

   $$\lim_{k\to\infty} \pi_k(a|s) = \mathbf{1}\left(a = \arg\max_b Q_k(s,b)\right), \tag{3.6}$$

   where $\pi_k$ and $Q_k$ denote the policy and the action-value function at episode $k$, respectively, and $\mathbf{1}$ is the indicator function.

A simple rule is to have, for example, $\varepsilon$ reduce to zero using an hyperbolic schedule, i.e., $\varepsilon_k = \frac{1}{k}$. Monte Carlo control converges to the optimal state-action value function (and, consequently, to an optimal policy) with a GLIE sequence of policies $\{\pi_k\}$.

## 3.3 Temporal difference learning

Temporal difference (TD) learning is one of the core ideas of reinforcement learning. TD methods learn from samples generated by the environment, like Monte Carlo methods, by bootstrapping from the current estimate of the value function, like dynamic programming methods.

In this section we will see how MC and TD methods are related, as well as two TD control algorithms.

### 3.3.1 Temporal difference prediction

The prediction step is similar to the Monte Carlo prediction step in the case of a non-stationary MDP,

$$V(s_t) \leftarrow V(s_t) + \alpha\left(G_t - V(s_t)\right), \tag{3.7}$$

where $\alpha$ is a step size parameter, such that $0 < \alpha \leq 1$. Instead of updating towards the empirical return obtained during an episode, $G_t$, we update from incomplete episodes, by bootstrapping, as in dynamic programming. When updating at each time step, we have the following update rule:

$$V(s_t) \leftarrow V(s_t) + \alpha \left( r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right). \tag{3.8}$$

This method is called TD(0) [7], due to the fact that the estimate is updated at every time step. The sampled reward plus the estimate of the value of the next state, $r_{t+1} + \gamma V(s_{t+1})$, is referred to as TD target. While $r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$, is called TD error, and is denoted as $\delta_t$. The TD prediction algorithm is given in pseudo-code in Algorithm 7. TD(0)

**Algorithm 7:** Temporal difference prediction, estimating $V \approx V^\pi$

```
function TDPrediction $(\pi, \alpha)$
    $V(s) \leftarrow 0$, for all $s$
    loop
        $s_t \leftarrow s_0 \sim p_0(\cdot)$
        loop for each step of the episode
            $a_t \sim \pi(\cdot|s_t)$
            $s_{t+1} \sim p(\cdot|s_t, a_t)$, $r_{t+1} \leftarrow r(s_t, a_t)$
            $V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t))$
            $s_t \leftarrow s_{t+1}$
    return $V$
```

can be proven to converge to $V^\pi$ with probability 1 [8].

In the previous chapter we saw that

$$\begin{aligned} V^\pi(s) &\doteq \mathbb{E}_\pi[G_t|s_t = s] && (3.9)\\ &= \mathbb{E}_\pi[r_{t+1} + \gamma G_{t+1}|s_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma \mathbb{E}_\pi[G_{t+1}|s_{t+1} = s']|s_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma V^\pi(s_{t+1})|s_t = s], && (3.10) \end{aligned}$$

where the second to last equality follows by the law of total expectation and the Markov property. Monte Carlo methods sample empirical returns to estimate (3.9). Dynamic programming methods use (3.10) as a target, by estimating $V^\pi(s_{t+1})$ (i.e., by doing prediction) and then computing the expected value via the model of the MDP. Temporal difference methods also use (3.10) as a target, but also have to estimate the expected value, due to the fact that they are model-free.

Clearly, the return, $G_t$, is an unbiased estimate of the value, $V^{\pi}(s_t)$, whereas the TD target, $r_{t+1} + \gamma V(s_{t+1})$, is not, since the estimate is done using the current estimate in the next state. Conversely, the MC target uses the complete return as estimate, which depends on many random actions, transitions, and rewards. Because of this, it is high variance. The TD target, however, depends solely on one random action, one random transition, and one random reward, meaning that it is much lower variance.

### 3.3.2 TD($\lambda$)

Monte Carlo and TD(0) methods can be seen as extremes on a spectrum. The methods in-between are represented by $n$-step methods. The $n$ indicates how many steps should be taken before updating the estimate of the value function. In this case, the target for an update is the $n$-step return [9],

$$G_{t:t+n} \doteq r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^{n} V(s_{t+n}), \tag{3.11}$$

where $G_{t:t+n}$ denotes the return from time step $t$ to time step $t+n$. The corresponding update will be

$$V(s_t) \leftarrow V(s_t) + \alpha \left(G_{t:t+n} - V(s_t)\right). \tag{3.12}$$

It can be empirically shown that in general intermediate values of $n$ achieve better results than TD(0) or MC methods. It is however not possible to know in advance which value of $n$ will be better for the current task. A simple idea is to instead not only update towards any $n$-step return, but an average of $n$-step returns for different values of $n$. We can elegantly combine all $n$-step updates by updating towards a return, called $\lambda$-return [9], which is a geometrically weighted average of all $n$-step returns, defined as:

$$G_t^{\lambda} \doteq (1-\lambda) \sum_{n=1}^{T} \lambda^{n-1} G_{t:t+n}, \tag{3.13}$$

where $\lambda \in [0,1]$ is a constant which determines how much each of the following $n$-step returns is decayed. In particular: the one-step return will have weight $1-\lambda$, and at each additional step the weight is decayed by a factor of $\lambda$. The $(1-\lambda)$ is a normalizing factor, which simply makes all weights sum to 1. This method is known as TD($\lambda$) [7] and, like TD(0), it converges to $V^{\pi}$ with probability 1 [10].

This approach requires to look forward into the future in order to compute the $\lambda$-return at each time step. For this reason it is more specifically referred to as forward TD($\lambda$). A better alternative, called backward TD($\lambda$) [11], allows us instead to compute the $\lambda$-return online, at each step, from incomplete episodes, while also being equivalent to forward

TD($\lambda$). Backward TD($\lambda$) makes use of eligibility traces, which we will now introduce. Eligibility traces are defined as follows:

$$\begin{aligned} &E_0(s) = 0 \\ &E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbf{1}(s_t = s) \,. \end{aligned} \tag{3.14}$$

When a state is visited the eligibility trace of that state is increased by 1. Then, the eligibility trace is decreased exponentially for each time step in which the state is not visited. Intuitively, they can be seen as a metric to decide how much each state has contributed to the reward obtained, depending on both the frequency and the recency with which the state has been visited. We can now update the estimate of the value function at each time step with respect to the TD error, $\delta_t$, and also the eligibility trace, $E_t(s)$, as follows:

$$V(s) \leftarrow V(s) + \alpha\delta_t E_t(s)\,, \text{ for all } s. \tag{3.15}$$

When $\lambda = 0$, it is easy to see that TD($\lambda$) and TD(0) are equivalent, since in this case only the current state is updated. Conversely, when $\lambda = 1$, the credit given to previous states decreases by $\gamma$ at each step, thus behaving like a Monte Carlo method, albeit online and with incremental updates.

### 3.3.3 Temporal difference control

Up to now we have only seen methods which do policy improvement on the same policy used for exploration. These methods are called on-policy methods. Off-policy methods instead have a target policy, which is the policy being evaluated and improved, and a behavior policy, from which the actions are sampled. We will now introduce SARSA [12] and Q-learning [9], which are respectively on- and off-policy TD control algorithms.

The SARSA update is done using the quintuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, which gives the name to the algorithm. Thus, the update rule is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right). \tag{3.16}$$

As in policy iteration, we continually estimate $Q^\pi$, while improving $\pi$ towards greediness with respect to $Q$. SARSA can be shown to converge [6] with a GLIE sequence of policies $\{\pi_k\}$, and a Robbins-Monro sequence of step sizes $\{\alpha_k\}$ such that

$$\sum_{k=0}^{\infty} \alpha_k = \infty \quad \text{and} \quad \sum_{k=0}^{\infty} \alpha_k^2 < \infty. \tag{3.17}$$

These conditions for $\alpha$ are a well-known result in stochastic approximation which give us the necessary condition to ensure convergence. The pseudo-code for SARSA with

TD(0) prediction is presented in Algorithm 8. We will call this version one-step SARSA. Additionally, the implementation can be easily modified to support TD($\lambda$) for the prediction step by using update rule (3.15) instead, as well as saving eligibility traces for state-action pairs, rather than only states. SARSA with a TD($\lambda$) prediction step is called SARSA($\lambda$) [12].

**Algorithm 8:** One-step SARSA, estimating $\pi \approx \pi^*$

```
function OneStepSARSA ($\alpha$)
    loop
        $s_t \leftarrow s_0 \sim p_0(\cdot)$
        loop for each step of the episode
            $a_t \sim \pi(\cdot|s_t)$                                        ($\varepsilon$-greedy)
            $s_{t+1} \sim p(\cdot|s_t, a_t)$, $r_{t+1} \leftarrow r(s_t, a_t)$
            $a_{t+1} \sim \pi(\cdot|s_{t+1})$                                  ($\varepsilon$-greedy)
            $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$
            $s_t \leftarrow s_{t+1}$
    return $\pi$
```

The update step for Q-learning is very similar, except that now the target policy is greedy with respect to $Q$, giving us the following update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left( r_{t+1} + \gamma \max_b Q(s_{t+1}, b) - Q(s_t, a_t) \right). \tag{3.18}$$

Due to the greediness of the target policy, the estimated action-value function $Q$ directly approximates the optimal action-value function $Q^*$, regardless of the policy used for exploration. In particular, Q-learning can be shown to converge [13, 10, 14] to the optimal action-value function under the assumption that all state-action pairs continue to be visited, and a Robbins-Monro sequence of step sizes, as in (3.17). Q-learning with TD(0) prediction is detailed in Algorithm 9. We also mention that, similarly to SARSA, two methods have been proposed to combine eligibility traces and Q-learning: Watkins's Q($\lambda$) [9] and Peng's Q($\lambda$) [15].

## 3.4 Value function approximation

So far we have always represented value functions by a lookup table. However, most interesting problems have large or continuous state spaces, which make the computation of the value function for all states not feasible. We can approximate the value function

**Algorithm 9:** Q-learning, estimating $\pi \approx \pi^*$

```
function Q-LEARNING $(\alpha)$
    loop
        $s_t \leftarrow s_0 \sim p_0(\cdot)$
        loop for each step of the episode
            $a_t \sim \pi(\cdot|s_t)$
            $s_{t+1} \sim p(\cdot|s_t, a_t)$, $r_{t+1} \leftarrow r(s_t, a_t)$
            $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_b Q(s_{t+1}, b) - Q(s_t, a_t))$
            $s_t \leftarrow s_{t+1}$
    return $\pi$
```

via function approximation (FA), in order to generalize to states, or state-action pairs, which are yet to be visited, as follows:

$$\begin{aligned} \hat{V}(s; \mathbf{w}) &\approx V^{\pi}(s), \text{ or} \\ \hat{Q}(s, a; \mathbf{w}) &\approx Q^{\pi}(s, a), \end{aligned} \tag{3.19}$$

where $\mathbf{w}$ is a weight vector. There are many function approximation methods, but stochastic gradient descent (SGD) methods are particularly well suited for online reinforcement learning, so we will consider this specific case.

### 3.4.1 Stochastic gradient methods

Let $J(\mathbf{w})$ be a differential function of parameter vector $\mathbf{w}$, where $\mathbf{w}$ is a column vector of real valued components of length $d$, $\mathbf{w} \doteq (w_1, w_2, \ldots, w_d)^{\top}$. We define the gradient of $J(\mathbf{w})$ as

$$\nabla_{\mathbf{w}} J(\mathbf{w}) \doteq \left( \frac{\partial J(\mathbf{w})}{\partial w_1}, \ldots, \frac{\partial J(\mathbf{w})}{\partial w_d} \right)^{\top}. \tag{3.20}$$

This vector is the direction of steepest ascent. By following the negative direction of the gradient vector, we will find a local minimum of the objective function $J(\mathbf{w})$. Formally, the update rule will be

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{1}{2}\eta \nabla_{\mathbf{w}} J(\mathbf{w}), \tag{3.21}$$

where $\eta$ is a step size parameter. We will also assume the sequence of step sizes $\{\eta_k\}$ to be a Robbins-Monro sequence, as in (3.17).

We define our objective function as the mean squared error between the approximate value function $\hat{V}(s;\mathbf{w})$ and the true value function $V^{\pi}(s)$:

$$J(\mathbf{w}) \doteq \mathbb{E}_{\pi}\left[\left(V^{\pi}(s) - \hat{V}(s;\mathbf{w})\right)^2\right]. \tag{3.22}$$

Thus, the update rule becomes

$$\begin{aligned}
\mathbf{w} &\leftarrow \mathbf{w} - \frac{1}{2}\eta\nabla_{\mathbf{w}}J(\mathbf{w}) \\
&= \mathbf{w} - \frac{1}{2}\eta\nabla_{\mathbf{w}}\mathbb{E}_{\pi}\left[\left(V^{\pi}(s) - \hat{V}(s;\mathbf{w})\right)^2\right] \\
&= \mathbf{w} - \eta\mathbb{E}_{\pi}\left[\left(V^{\pi}(s) - \hat{V}(s;\mathbf{w})\right)\nabla_{\mathbf{w}}\hat{V}(s;\mathbf{w})\right],
\end{aligned} \tag{3.23}$$

where the last equality follows by the chain rule. Furthermore, since we are only considering incremental methods (although, of course, there are also batch methods), we can simply apply the update after each sample, as follows:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta\left(V^{\pi}(s_t) - \hat{V}(s_t;\mathbf{w})\right)\nabla_{\mathbf{w}}\hat{V}(s_t;\mathbf{w}). \tag{3.24}$$

### 3.4.2 Linear value function approximation

We will now consider the simple yet effective case in which the value function is represented by a linear combination of features, i.e.,

$$\hat{V}(s;\mathbf{w}) \doteq \mathbf{w}^{\top}\phi(s) \doteq \sum_{k=1}^{d} w_k\phi_k(s), \tag{3.25}$$

where vector $\phi(s)$ is a feature vector representing state $s$. Each of its components $\phi_k(s)$ is the value of a function $\phi_k : \mathcal{S} \to \mathbb{R}$, which can be nonlinear.

In this case, the objective function is quadratic in the parameter vector $\mathbf{w}$, meaning that stochastic gradient descent will converge to the global minimum of the objective function $J(\mathbf{w})$. Additionally, the update rule becomes remarkably simple:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta\left(V^{\pi}(s_t) - \hat{V}(s_t;\mathbf{w})\right)\phi(s_t), \tag{3.26}$$

since $\nabla_{\mathbf{w}}\hat{V}(s;\mathbf{w}) = \phi(s)$.

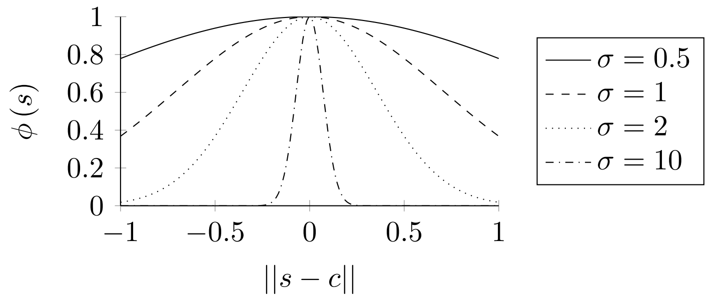

Figure 3.1: Response for various choices of $\sigma$.

### 3.4.3 Radial basis functions

When we have no domain knowledge regarding the problem at hand, an option is that of using kernel feature maps, such as radial basis functions (RBFs). Each feature, $\phi_k$, takes a value in the interval $[0, 1]$. The response, $\phi_k(s)$, indicates how distant the state is to the feature's center, $c_k$, where the distance also depends on the width of the feature, $\sigma_k$. A common choice for the type of radial basis function is the Gaussian, defined as

$$\phi_k(s) \doteq \exp\left(-\left(\sigma_k \, ||s - c_k||\right)^2\right). \tag{3.27}$$

Figure 3.1 shows the response in the one-dimensional setting for various choices of $\sigma$. As we will see later, such features require manual tuning in order to learn efficiently. Because of this, if possible, it is generally advisable to construct problem-specific features.

### 3.4.4 Prediction with function approximation

Update rule (3.26) cannot be performed. In fact, we do not have the true value function, $V^\pi(s)$, but we can substitute it with some approximation. For MC methods, the target is the return, $G_t$. For TD(0), the target is the TD target. For TD($\lambda$), the target is the $\lambda$-return, $G_t^\lambda$. We will see how prediction can be performed only in the last two cases, since MC methods do not extend as well to value function approximation.

Assuming linear value function approximation, the update rule for TD(0) becomes

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} - \eta\left(r_{t+1} + \gamma \hat{V}(s_{t+1}; \mathbf{w}) - \hat{V}(s_t; \mathbf{w})\right) \nabla_{\mathbf{w}} \hat{V}(s_t; \mathbf{w}) \\ &= \mathbf{w} - \eta \delta_t \phi(s_t).\end{aligned} \tag{3.28}$$

Instead, in the case of backward TD($\lambda$), we have,

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} - \eta \left( G_t^\lambda - \hat{V}\left(s_t; \mathbf{w}\right) \right) \nabla_{\mathbf{w}} \hat{V}\left(s_t; \mathbf{w}\right) \\ &= \mathbf{w} - \eta \left( G_t^\lambda - \hat{V}\left(s_t; \mathbf{w}\right) \right) \phi\left(s_t\right),\end{aligned} \tag{3.29}$$

and in the case of forward TD($\lambda$) [7],

$$\begin{aligned}\delta_t &= r_{t+1} + \gamma \hat{V}\left(s_{t+1}; \mathbf{w}\right) - \hat{V}\left(s_t; \mathbf{w}\right) \\ E_t &= \gamma\lambda E_{t-1} + \nabla_{\mathbf{w}} \hat{V}\left(s_t; \mathbf{w}\right) = \gamma\lambda E_{t-1} + \phi\left(s_t\right) \\ \mathbf{w} &\leftarrow \mathbf{w} - \eta \delta_t E_t.\end{aligned} \tag{3.30}$$

Unfortunately, however, in the case of linear value function approximation, we do not have the same convergence guarantees that we have in the tabular case. In fact, TD prediction has an asymptotic error which depends on the discount factor $\gamma$ [16].

### 3.4.5 Control with function approximation

Recall that to do control in the model-free case we need to use the action-value function instead of the state-value function. This implies that rather than representing a state by a feature vector, we will now represent a state-action pair, as follows:

$$\phi\left(s, a\right) \doteq \left(\phi_1\left(s, a\right), \ldots, \phi_d\left(s, a\right)\right)^\top. \tag{3.31}$$

This also causes the vector $\mathbf{w}$ to be of length $d \cdot |\mathcal{A}|$. We can see this as having a (state-value) feature vector for each of the possible actions. In the linear case, a simple idea is that of appending the feature vector $|\mathcal{A}|$ times and then set to zero the feature values of all actions except the currently selected one [17], as follows:

$$\phi\left(s, a\right) \doteq \begin{pmatrix} \phi\left(s\right) \cdot \mathbf{1}\left(a_t = a_1\right) \\ \vdots \\ \phi\left(s\right) \cdot \mathbf{1}\left(a_t = a_{|\mathcal{A}|}\right) \end{pmatrix}, \tag{3.32}$$

and then updating the whole weight vector by stochastic gradient descent,

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \delta_t \phi\left(s_t, a_t\right). \tag{3.33}$$

Of course, copying the features many times is computationally very expensive. A better alternative is to leverage the sparsity of the vector and update only the relevant action.

To do so, we reshape the value vector $\mathbf{w}$ into a matrix of size $d \times |\mathcal{A}|$, and then update only the column corresponding to the selected action,

$$\mathbf{w}_{k,a} \leftarrow \mathbf{w}_{k,a} + \eta \delta_t \phi\left(s_t\right) \text{ for } k \in \{1, \ldots, d\} \text{ and } a = a_t. \tag{3.34}$$

Once again, in order to do control we apply the concept of generalized policy iteration, by interleaving (approximate) policy evaluation steps and policy improvement steps. However, in the linear case, SARSA ends up wandering around a bounded region near the optimal solution [18, 19], whereas Q-learning can even exhibit instability [20]. Additionally, when doing control with nonlinear function approximators no theoretical guarantees have been proven.

# Chapter 4

# Policy-based reinforcement learning

We will now introduce policy-based methods. Up to now, we presented only reinforcement learning methods that learned an action-value function, from which the target policy was implicitly derived. Policy-based methods directly optimize a stochastic policy instead.

The policy is generally modeled with a parameterized function with respect to a weight vector $\theta$, and is denoted $\pi_\theta(a|s)$. The goal is to find $\theta$ that maximizes an objective function $J(\theta)$. Policy-based reinforcement learning is an optimization problem. We will focus on gradient methods, even though many other approaches can be used.

## 4.1 Policy gradient

Policy gradient methods search for a local maximum in $J(\theta)$ by ascending the gradient of the policy, with respect to the parameter vector $\theta$,

$$\theta \leftarrow \theta + \eta \nabla_\theta J(\theta), \tag{4.1}$$

where $\eta$ is a step size parameter. We will also assume the sequence of step sizes $\{\eta_k\}$ to be a Robbins-Monro sequence, as in (3.17). The gradient $\nabla_\theta J(\theta)$ is called the policy gradient, and is defined as

$$\nabla_\theta J(\theta) \doteq \left(\frac{\partial J(\theta)}{\partial \theta_1}, \ldots, \frac{\partial J(\theta)}{\partial \theta_d}\right)^\top. \tag{4.2}$$

### 4.1.1 Analytic computation

We will consider the simple case in which the MDP is episodic and $\gamma = 1$, although the following results can be extended to non-episodic MDPs and $\gamma < 1$. Let the objective function $J(\theta)$ be the expected rewards for an episode,

$$\begin{aligned} J(\theta) &\doteq \mathbb{E}_{(s_t,a_t)\sim\pi_\theta}\left[\sum_{t=1}^{T} r_t\right] \\ &= \mathbb{E}_{\tau\sim\pi_\theta}\left[r(\tau)\right] \\ &= \sum_{\tau} p_\theta(\tau)\, r(\tau), \end{aligned} \tag{4.3}$$

where $\tau$ is a trajectory, $(s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T)$, $p_\theta(\tau)$ denotes the probability of sampling a trajectory $\tau$ when following policy $\pi_\theta$, i.e., $\tau \sim p_\theta(\tau)$, and $r(\tau)$ is the sum of rewards for a trajectory $\tau$.

Assuming that we can compute the policy gradient $\nabla_\theta \pi_\theta(a|s)$, then we can compute the policy gradient, as follows:

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \sum_{\tau} p_\theta(\tau)\, r(\tau) \\ &= \sum_{\tau} \nabla_\theta p_\theta(\tau)\, r(\tau) \\ &= \sum_{\tau} \frac{p_\theta(\tau)}{p_\theta(\tau)} \nabla_\theta p_\theta(\tau)\, r(\tau) \\ &= \sum_{\tau} p_\theta(\tau)\, r(\tau)\, \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} \\ &= \sum_{\tau} p_\theta(\tau)\, r(\tau)\, \nabla_\theta \log p_\theta(\tau) \\ &= \mathbb{E}_{\tau\sim\pi_\theta}\left[r(\tau)\, \nabla_\theta \log p_\theta(\tau)\right], \end{aligned} \tag{4.4}$$

where the second to last equality follows from $\nabla \log f(x) = \frac{\nabla f(x)}{f(x)}$ which, in turn, follows by the chain rule. This is also known as the REINFORCE trick [21].

Since the expectation $\mathbb{E}_{\tau\sim\pi_\theta}[\cdot]$ can be estimated by sample averages, we only need the derivative $\nabla_\theta \log p_\theta(\tau)$. Importantly, this derivative can be computed without knowledge of the initial state distribution, or the dynamics of the MDP. In fact, we have that

$$\begin{aligned}\nabla_\theta \log p_\theta(\tau) &= \nabla_\theta \log \left( p_0(s_0) \prod_{t=0}^{T-1} p(s_{t+1}|s_t, a_t)\, \pi_\theta(a_t|s_t) \right) \\ &\overset{(a)}{=} \nabla_\theta \left( \log p_0(s_0) + \sum_{t=0}^{T-1} \log p(s_{t+1}|s_t, a_t) + \sum_{t=0}^{T-1} \log \pi_\theta(a_t|s_t) \right) \\ &\overset{(b)}{=} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t), \qquad (4.5)\end{aligned}$$

where (a) follows from $\log(ab) = \log a + \log b$, and (b) follows from the fact that only the policy depends on $\theta$. The gradient $\nabla_\theta \log \pi_\theta(a|s)$ is known as the score function. Thus, by combining (4.4) and (4.5), we have that

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta} \left[ r(\tau) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \right]. \qquad (4.6)$$

### 4.1.2 Policy gradient theorem

A more general theorem, known as the policy gradient theorem [22, 23], allows us to substitute the sum of rewards in a trajectory, $r(\tau)$, with the action-value function, $Q^\pi(s, a)$. Formally, the policy gradient, becomes

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[ \sum_{t=0}^{\infty} Q^{\pi_\theta}(s_t, a_t)\, \nabla_\theta \log \pi_\theta(a_t|s_t) \right]. \qquad (4.7)$$

This update has no bias but high variance. Many variants have been proposed in order to reduce the variance while keeping the bias unchanged [24]. In general, they take the form

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[ \sum_{t=0}^{\infty} \Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t) \right]. \qquad (4.8)$$

We will see the cases in which $\Psi_t$ is the return starting at time step $t$, $G_t$, and the TD target, $r_{t+1} + \gamma V(s_{t+1})$.

## 4.2 Differentiable policies

Prior to introducing any policy gradient method, we will see two kinds of policies commonly used that are differentiable with respect to the parameter vector $\theta$.

### 4.2.1 Softmax policy

If the action space is discrete and not too large, the softmax function is generally used to parameterize the policy. It is defined as:

$$\pi_\theta (a|s) \doteq \frac{e^{\phi(s,a)^\top \theta}}{\sum_b e^{\phi(s,b)^\top \theta}}. \tag{4.9}$$

The score function can be derived as follows:

$$\begin{aligned}\nabla_\theta \log \pi_\theta (a|s) &= \nabla_\theta \log \frac{e^{\phi(s,a)^\top \theta}}{\sum_b e^{\phi(s,b)^\top \theta}} \\ &= \nabla_\theta \log e^{\phi(s,a)^\top} \theta - \nabla_\theta \log \sum_b e^{\phi(s,b)^\top \theta},\end{aligned} \tag{4.10}$$

where the equality follows from $\log \frac{a}{b} = \log a - \log b$. The first term can be derived easily,

$$\nabla_\theta \log e^{\phi(s,a)^\top} \theta = \nabla_\theta \phi (s,a)^\top \theta = \phi (s,a), \tag{4.11}$$

while the second one requires a bit more work,

$$\begin{aligned}\nabla_\theta \log \sum_b e^{\phi(s,b)^\top \theta} &\overset{(a)}{=} \frac{\nabla_\theta \sum_b e^{\phi(s,b)^\top \theta}}{\sum_b e^{\phi(s,b)^\top \theta}} \\ &\overset{(b)}{=} \frac{\sum_b \phi (s,b) e^{\phi(s,b)^\top \theta}}{\sum_b e^{\phi(s,b)^\top \theta}} \\ &\overset{(c)}{=} \sum_b \phi (s,b) \pi_\theta (b|s) \\ &= \mathbb{E}_{\pi_\theta} [\phi (s, \cdot)],\end{aligned} \tag{4.12}$$

where (a) follows by the chain rule, (b) follows again by the chain rule, and (c) follows by the definition of $\pi_\theta$ (4.9). Putting everything together, we have that the score function is

$$\nabla_\theta \log \pi_\theta (a|s) = \phi (s,a) - \mathbb{E}_{\pi_\theta} [\phi (s, \cdot)]. \tag{4.13}$$

### 4.2.2 Gaussian policy

A typical choice in the case of continuous actions spaces is a Gaussian policy [21]. Both the mean and the variance can be parameterized (in which case the parameter vector would be $\theta = (\theta_\mu, \theta_\sigma)^\top$), but we will assume for the variance to be fixed.

Actions are sampled from a normal probability distribution, $a \sim \mathcal{N}\left(\mu_\theta(s), \sigma^2\right)$, that is

$$\pi_\theta(a|s) \doteq \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{\left(a-\mu_\theta(s)\right)^2}{2\sigma^2}\right), \tag{4.14}$$

where the mean is a linear combination of state features, $\mu_\theta(s) = \phi(s)^\top \theta$. Thus, the score function is:

$$\begin{aligned}
\nabla_\theta \log \pi_\theta(a|s) &\overset{(a)}{=} \nabla_\theta \log \exp\left(-\frac{\left(a-\mu_\theta(s)\right)^2}{2\sigma^2}\right) - \nabla_\theta \log \frac{1}{\sigma\sqrt{2\pi}} \\
&\overset{(b)}{=} -\frac{1}{2\sigma^2} \nabla_\theta \left(a-\mu_\theta(s)\right)^2 \\
&\overset{(c)}{=} -\frac{1}{2\sigma^2} \left(-2\left(a-\mu_\theta(s)\right) \nabla_\theta \mu_\theta(s)\right) \\
&\overset{(d)}{=} \frac{\left(a-\mu_\theta(s)\right)\phi(s)}{\sigma^2}.
\end{aligned} \tag{4.15}$$

where (a) follows from $\log \frac{a}{b} = \log a - \log b$, (b) follows from the fact that the second term does not depend on $\theta$, (c) follows by the chain rule, and (d) follows from $\nabla_\theta \mu_\theta(s) = \phi(s)$.

## 4.3 Basic policy gradient algorithms

We are now ready to introduce some basic policy gradient algorithms. First, we will see REINFORCE [21], which can be simply described as Monte Carlo policy gradient. Then, we will add a critic as a simple way to reduce the variance.

### 4.3.1 REINFORCE

The REINFORCE algorithm uses the return $G_t$ as an unbiased sample of $Q^\pi(s_t, a_t)$, as in value-based Monte Carlo. In particular, we will sample an episode following $\pi_\theta$, then, for each time step $t$ of the episode, we estimate the policy gradient (4.7), and update the parameter vector $\theta$ according to stochastic gradient ascent,

$$\theta \leftarrow \theta + \eta G_t \nabla_\theta \log \pi_\theta(a_t|s_t). \tag{4.16}$$

The pseudo-code for the REINFORCE algorithm is given in Algorithm 10.

Of course, as a Monte Carlo method, REINFORCE has high variance. A common way to reduce variance is to introduce a baseline, $b(s)$, into the expectation. The baseline can

**Algorithm 10:** REINFORCE, estimating $\pi_\theta \approx \pi^*$

```
function REINFORCE ($\eta$)
    $\theta_k \leftarrow 0$, for $k \in \{1, \ldots, d\}$
    loop
        Sample an episode following $\pi_\theta$: $(s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T)$
        foreach $(s_t, a_t)$ do
            $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
            $\theta \leftarrow \theta + \eta G_t \nabla \log \pi_\theta (a_t|s_t)$
    return $\pi_\theta$
```

be any function, as long as it does not vary with $a$. The gradient now takes the form

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} \left(G_t - b(s_t)\right) \nabla_\theta \log \pi_\theta (a_t|s_t)\right]. \tag{4.17}$$

This can be done as subtracting a baseline is unbiased in expectation, in fact we have that

$$\begin{aligned}\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log p_\theta(\tau)\, b\right] &= \sum_\tau p_\theta(\tau) \nabla_\theta \log p_\theta(\tau)\, b \\ &= \sum_\tau \nabla_\theta p_\theta(\tau)\, b \\ &= b \nabla_\theta \sum_\tau p_\theta(\tau) \\ &= b \nabla_\theta 1 = 0. \end{aligned} \tag{4.18}$$

A typical choice for the baseline is the state-value function, $b(s) = V^\pi(s)$. This allows us to rewrite the policy gradient using the so-called advantage function, $A^\pi$,

$$\begin{aligned} A^\pi(s,a) &\doteq Q^\pi(s,a) - V^\pi(s) \\ \nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T-1} A^\pi(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)\right], \end{aligned} \tag{4.19}$$

since the return is an unbiased estimate of $Q^\pi(s_t, a_t)$. Intuitively, the advantage function measures how much better or worse it is to take action $a$ in state $s$, rather than randomly sampling an action according to $\pi_\theta(\cdot|s)$. Algorithm 10 can be modified to use a baseline by simply replacing the return with the advantage function in the update step defined in (4.16), where the advantage function is approximated by $G_t - \hat{V}(s_t; \mathbf{w})$, for some weight vector $\mathbf{w}$.

### 4.3.2 Actor-critic

In actor-critic methods [25] we have a critic, which estimates the value function, and an actor, which updates the policy in the direction suggested by the critic.

The REINFORCE algorithm with a baseline can indeed be seen as a case of actor-critic. In fact, in the case presented the baseline is an estimate of the state-value function. Alternatively, we can reduce variance and accelerate learning by using bootstrapping methods for prediction, such as the ones introduced in sections 3.3.1 and 3.3.2. Of course, this is done at the cost of introducing some bias, since bootstrapping methods update their estimate based on the current estimate of subsequent states. We will only see the case for TD(0), as generalizing the concepts to TD($\lambda$) is straightforward. In this case the algorithm is called one-step actor-critic. We replace the full return, $G_t$, with the TD target in the gradient update, as follows:

$$\begin{aligned} \theta &\leftarrow \theta + \eta \left( G_t - \hat{V}\left(s_t; \mathbf{w}\right) \right) \nabla_\theta \log \pi_\theta \left(a_t | s_t\right) \\ &= \theta + \eta \left( r_{t+1} + \gamma \hat{V}\left(s_{t+1}; \mathbf{w}\right) - \hat{V}\left(s_t; \mathbf{w}\right) \right) \nabla_\theta \log \pi_\theta \left(a_t | s_t\right) \\ &= \theta + \eta \delta_t \nabla_\theta \log \pi_\theta \left(a_t | s_t\right) \end{aligned} \tag{4.20}$$

Algorithm 11 shows the pseudo-code for the one-step actor-critic algorithm.

**Algorithm 11:** One step actor-critic, estimating $\pi_\theta \approx \pi^*$

```
function OneStepActorCritic ($\eta_\theta, \eta_\mathbf{w}$)
    $\theta_k \leftarrow 0$, for $k \in \{1, \dots, d_\theta\}$
    $\mathbf{w}_k \leftarrow 0$, for $k \in \{1, \dots, d_\mathbf{w}\}$
    loop
        $s_0 \sim p_0(\cdot)$
        loop for each step of the episode
            $a_t \sim \pi_\theta(\cdot | s_t)$
            $s_{t+1} \sim p(\cdot | s_t, a_t)$, $r_{t+1} \leftarrow r(s_t, a_t)$
            $\delta_t \leftarrow r_{t+1} + \gamma \hat{V}(s_{t+1}, a_{t+1}; \mathbf{w}) - \hat{V}(s_t, a_t; \mathbf{w})$
            $\theta \leftarrow \theta + \eta_\theta \delta_t \nabla_\theta \log \pi_\theta(a_t | s_t)$
            $\mathbf{w} \leftarrow \mathbf{w} - \eta_\mathbf{w} \delta_t \phi(s_t, a_t)$
            $s_t \leftarrow s_{t+1}$
    return $\pi_\theta$
```

### 4.3.3 Comparison of value- and policy-based methods

Policy-based methods were the earliest to be studied in reinforcement learning, but they were neglected in favor of value-based methods starting from the introduction of Q-learning [9]. In recent years, however, several developments [26, 27, 28] have shifted once again the focus on policy-gradient methods.

Although policy-gradient methods generally converge locally, they still have better convergence properties than value-based methods with function approximation. Additionally, they can learn stochastic policies, which are beneficial in the case of games, since repetitive actions can be exploited by the opponents, and environments in which the observations of the state are not fully representative, as in this case some states share the same observation, thus possibly causing deterministic policies to always choose non-optimal actions. Ultimately, as we have seen, they can even be employed with both high-dimensional and continuous action spaces.

# Chapter 5

# Empirical results

We now move on from theory by running some experiments using the methods presented in the previous chapters. We start by presenting the environments used, then, just as in the first part, we will first present results in the case in which both the state and the action space are finite, and then the case in which the state space is continuous.

## 5.1 Domains

In this section we present the environments on which we will run the experiments. The first two environments have a finite state space, while the third one has a continuous state space. Additionally, in the third environment we have the possibility of either choosing the variant with a finite action space, or the variant with a continuous action space.

### 5.1.1 Cliff walking

The cliff walking environment [29, Example 6.6] is a $4 \times 12$ grid in which the agent has to travel from a starting point (S) to a goal (G), as shown in Figure 5.1. At each time step the agent receives a reward of -1, and a reward of -100 if it falls into the cliff, which resets the agent back to the starting point. The goal is a terminal state, meaning that any action taken in that state will transition to itself, with reward 0.

Since the agent receives a reward of -1 for each time step, it is incentivized to reach the goal as fast as possible. Clearly, this implies that the optimal policy is the one that walks closest to the cliff, obtaining a cumulative reward of -13 for each episode.

|  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |
| S | Cliff |  |  |  |  |  |  |  |  |  | G |

Figure 5.1: Cliff walking environment.

|  |  |  |  |
|---|---|---|---|
| S |  |  |  |
|  | H |  | H |
|  |  |  | H |
| H |  |  | G |

(a) Environment.

|  |  |  |  |
|---|---|---|---|
| ← | ↑ | ← | ↑ |
| ← | H | ← | H |
| ↑ | ↓ | ← | H |
| H | → | ↓ | G |

(b) Optimal policy.

Figure 5.2: Frozen lake environment.

### 5.1.2 Frozen lake

The frozen lake environment is a $4 \times 4$ grid in which the agent has to cross the frozen lake from a starting point (S) to a goal (G) while avoiding holes (H). The default configuration of the grid and the according optimal policy are shown in Figure 5.2. Due to the slippery nature of the lake, the agent will move towards its intended direction with probability 1/3, else it will move in either of the perpendicular directions with an equal probability of 1/3. The reward is 1 if the agent reaches the goal, and 0 otherwise.

Intuitively, the solution can be explained as follows: the best policy is to move in such a way that both the intended direction, as well as its perpendicular directions, do not result in falling into a hole. Thus, here the objective is to reach the goal as safely as possible, rather than as fast as possible.

### 5.1.3 Mountain car

In the mountain car environment [30] a car placed at the bottom of a valley has to reach a goal state on top of a hill. Due to the steepness of the valley, the car cannot simply accelerate forwards and reach the goal. Instead, it has to sway from one hill to the other until it has gained enough momentum to get to the top. Figure 5.3 is a visual representation of the environment.

The state is a tuple containing the position of the car along the x-axis and the velocity

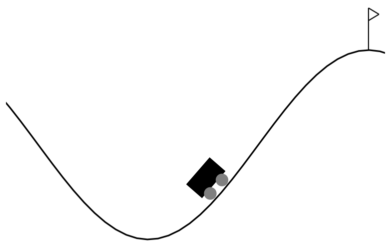
Figure 5.3: Mountain car environment.

of the car. There are two variants of the environment, the first one has 3 discrete actions (accelerate to the left, do not accelerate, accelerate to the right), whereas the second one takes as an action a value between $[-1, 1]$ representing the directional force applied to the car. Additionally, in the continuous action space case, the car is penalized for taking actions of large magnitude. In both cases the maximum number of steps per episode has been set to 10000.

## 5.2 Finite state space experiments

We will now compare the theoretical results presented in the first two chapters, with empirical results obtained in these environments. We will also observe that even in such simple problems, both the choice of the method to be used, as well as its parameters, is not straightforward. In order to assess how well the reinforcement learning agents have trained, we will use the mean squared error between the optimal value, computed via dynamic programming, and the estimate value.

### 5.2.1 Policy and value iteration

We first present an example comparing policy and value iteration. We will consider the cliff walking problem. Since the the environment has a terminal state, we can also consider the case in which $\gamma = 1$.

Figure 5.4 presents the maximum change in the value function for each full sweep of the state space, for the first prediction step of policy iteration, with different discount values. The learned policy is the same for each of the optimal value functions obtained with the various discount factors, yet the number of sweeps needed for the first prediction step varies widely. Table 5.1 illustrates this. Furthermore, as the discount factor gets smaller, each of the prediction steps is less precise, thus requiring more iterations in order to converge to the optimal policy.

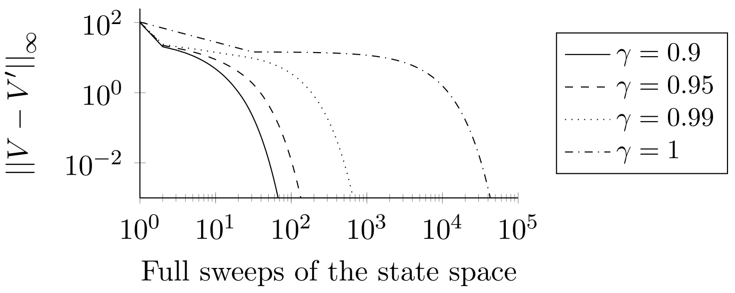

Figure 5.4: Convergence rate for the first prediction step of policy iteration.

| $\gamma$ | Avg. number of sweeps | Iterations |
|---|---|---|
| 0.9 | 63 | 13 |
| 0.95 | 127 | 12 |
| 0.99 | 282 | 9 |
| 1 | 4298 | 9 |

Table 5.1: Average number of sweeps in the prediction step of policy iteration and total number of iterations needed to converge to the optimal policy.

Interestingly, policy iteration still converges to the optimal value function with $\gamma = 1$ when stopping the prediction step to a maximum upper bound of sweeps of the state space (e.g., 100), clearly indicating that convergence in the prediction step is not always necessary to converge to the optimal policy.

In this problem, value iteration simply requires one iteration to converge to the optimal value. Of course, the initial value function and policy influence significantly the convergence rate of both methods.

### 5.2.2 Scheduling strategies for $\varepsilon$

We now move onto reinforcement learning methods. We will start by comparing different scheduling strategies for $\varepsilon$ with various methods. In figure 5.5 the schedules taken into consideration are shown.

Figure 5.6 presents the mean squared error (MSE) between the optimal value and the estimate value at some episode for three reinforcement learning methods with the various scheduling strategies. The environment used here is the frozen lake problem. The results are averaged over 100 runs.

The first two cases, first-visit Monte Carlo and SARSA, require a GLIE sequence of policies

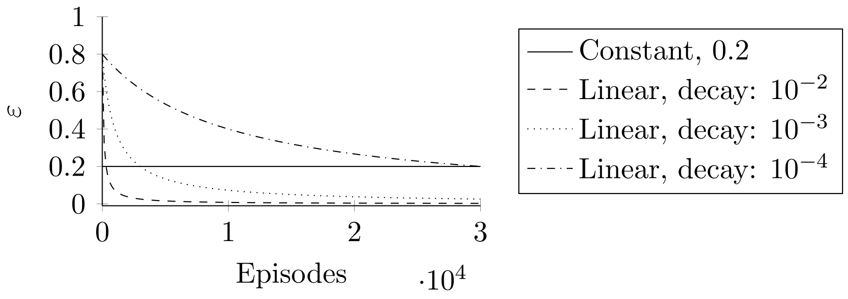

Figure 5.5: $\varepsilon$ scheduling strategies taken into consideration.

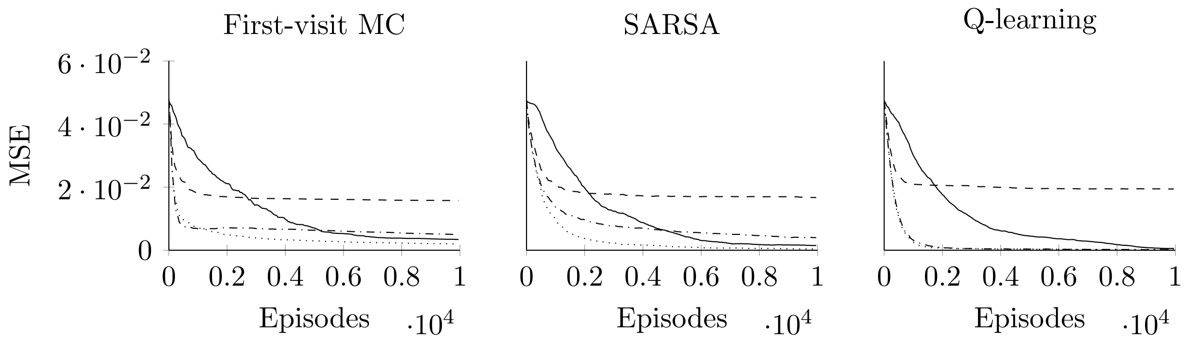

Figure 5.6: MSE for various reinforcement learning methods with different scheduling strategies for $\varepsilon$.

in order to converge to the optimal value. Still, in this simple example even a constant $\varepsilon$ works adequately, even if the training the error does not decrease as quickly as in the case of the decaying schedules. Among GLIE policies certain decay values work better than others. Although this might also depend on the problem itself, it is mainly a byproduct of the chosen number of training episodes. A policy whose $\varepsilon$ decays too rapidly, such as the one with a decay of $10^{-2}$, will result in low exploration and, consequently, worse results. Vice versa, a policy which remains exploratory for too long does not allow to the agent to exploit what it has learned. Ultimately, we have Q-learning, which only needs that the state-action pairs continue to be visited to converge. As a result of this, the policy with the worst performance is once again the one with highest decay.

Figure 5.7 displays the MSE $\pm$ the standard deviation across 10 runs, in the case of the policy with a decay rate of $10^{-3}$. First-visit Monte Carlo, being a Monte Carlo method, is high variance due to the fact that the precision of its estimate depends on all the states, actions, and rewards in each episode. Here, since the episodes can be rather short, especially at the start of the training process, where the agent often falls in the hole, the difference in variance between the methods is not as remarkable.

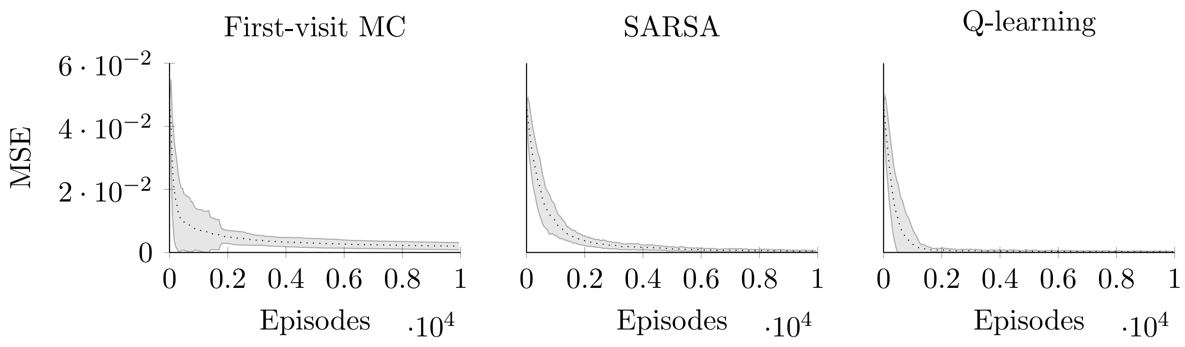

Figure 5.7: MSE ± the standard deviation across 10 runs for various reinforcement learning methods with $\varepsilon$ decaying at a rate of $10^{-3}$.

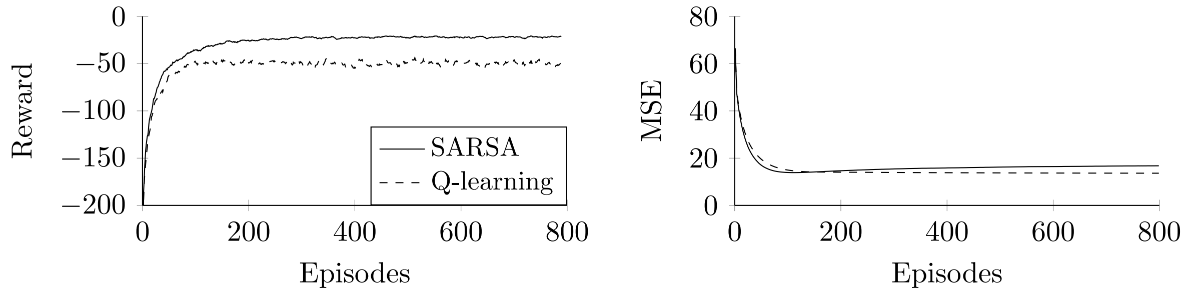

Figure 5.8: Reward (averaged over 100 runs and then smoothed with a moving average of 10 episodes) and MSE for each episode (averaged over 100 runs).

Even in such a simple environment we can observe how scheduling strategies that should theoretically converge, may fail to do so in practice. Vice versa, with a constant value of $\varepsilon$ we obtained satisfactory results even though reinforcement learning theory says otherwise. Furthermore, such observations are obviously only possible where the optimal value can be computed, but in real-world scenarios, where the mean squared error cannot be estimated, the trained agents need to be evaluated separately.

### 5.2.3 On- and off-policy learning

We now present a further example in order to compare SARSA and Q-learning, on the cliff walking environment. Both methods are trained with a constant $\varepsilon$ of 0.1. The left part of Figure 5.8 shows the reward obtained for each of the 800 episodes of training.

The rewards obtained during training also include rewards obtained after exploratory moves. These moves can sometimes also make the agent fall off the cliff when walking on its edge. Q-learning falls off more frequently since it directly learns the values of the optimal greedy policy, whereas SARSA takes a safer route by learning the values of the (approximately) optimal $\varepsilon$-greedy policy. This causes Q-learning to obtain less reward during training, even though it has learned to behave optimally, while SARSA obtains larger rewards as it walks farther away from the cliff.

Even though this example was specifically designed for this comparison, it makes apparent that in general the reward obtained during training is not a good proxy for how well the agent will do in the real environment. Furthermore, as exhibited in the right part of Figure 5.8, the error of the two methods is almost identical, yet, due to the differences between the two detailed in the previous paragraph, Q-learning will always behave optimally, whereas SARSA will behave sub-optimally by being safer, obtaining a total reward of either -15 or -17.

### 5.2.4 The choice of $\alpha$

The TD update (3.8) can be rearranged as follows:

$$\begin{aligned} V(s_t) &\leftarrow V(s_t) + \alpha\left(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right) \\ &= V(s_t) - \alpha V(s_t) + \alpha\left(r_{t+1} + \gamma V(s_{t+1})\right) \\ &= (1-\alpha) V(s_t) + \alpha\left(r_{t+1} + \gamma V(s_{t+1})\right). \end{aligned} \tag{5.1}$$

When $0 < \alpha \leq 1$ we can interpret the step size parameter as the weight given between the old estimate of the value, $V(s_t)$, and the new estimate of the value, $r_{t+1} + \gamma V(s_{t+1})$. Clearly, the value of $\alpha$, alongside its decay schedule, takes the role of the learning rate, as in supervised learning.

Figure 5.9 presents the MSE $\pm$ the standard deviation obtained training a SARSA agent on the frozen lake environment with different values of $\alpha$. The policy is linearly decaying with a decay of $10^{-3}$, starting from 0.8. Predictably, smaller values of $\alpha$ convergence steadily but slowly. Vice versa, larger values of $\alpha$ converge quickly, but with oscillations which might compromise the precision of the estimate. As usual, the parameter has to be manually tuned in order to achieve a good trade-off between the two.

## 5.3 Continuous state space experiments

We now proceed to the case of continuous state spaces, thus losing the possibility of using the mean squared error as a metric for how well the agent has trained. Instead, depending on the task, we will either use the steps or the reward per episode.

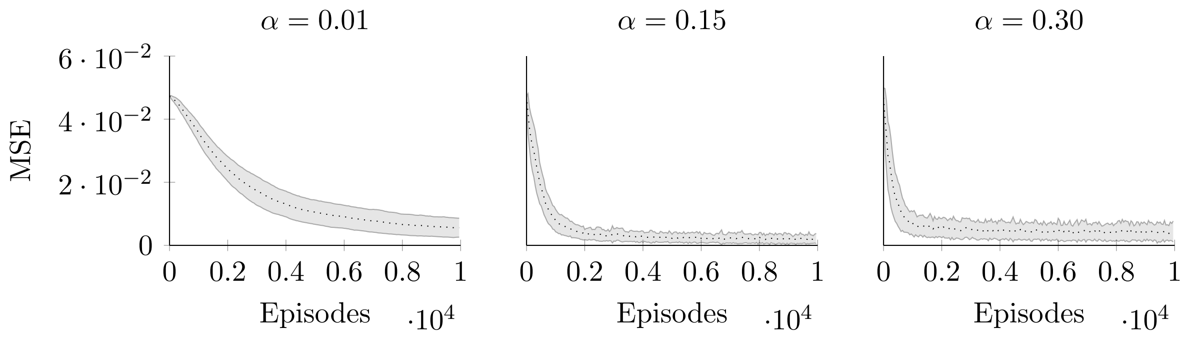

Figure 5.9: MSE (averaged over 100 runs) $\pm$ the standard deviation (over 100 runs) with SARSA and different values of $\alpha$.

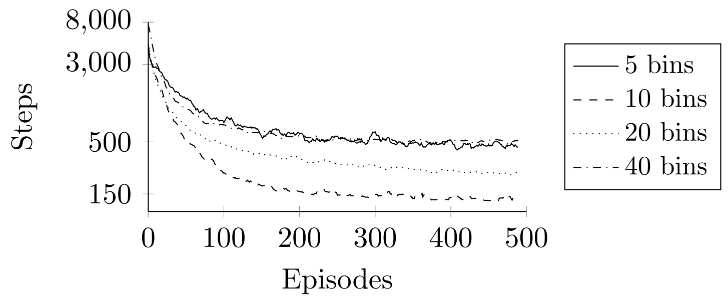

Figure 5.10: Steps per episode during training for various levels of discretization, smoothed with a moving average of 10 episodes.

### 5.3.1 Discretizing the state space

We will start by taking the simple approach of discretizing the state space of the mountain car environment.

A total of 100 agents per discretization level have been trained, each for 500 episodes. Figure 5.10 shows the average steps per episode among all agents during the training process. At the end of each run of training, each agent was evaluated on 100 runs. Since the objective of the environment is for the car to reach the goal as fast as possible, we will use the steps to reach the goal on an episode as a metric to determine how well the agent has learned. Table 5.2 shows the results of this evaluations for each level of discretazation taken into consideration. In particular, we report the average number of steps and the standard deviation across all trained agents.

With both few and many bins we have middling results. In the first case the approximation is too coarse, whereas in the latter one the agents still needs more training episodes in order to fully explore the discretized state space. This environment is rather manageable,

| Bins | Average | Standard deviation |
|---|---|---|
| 5×5 | 3,009 | 3,101 |
| 10×10 | 320 | 1,047 |
| 20×20 | 1,017 | 1,417 |
| 40×40 | 4,666 | 2,454 |

Table 5.2: Steps to reach the goal across 100 runs of evaluation with different discretization levels.

thus allowing good results even with few bins and training episodes. A simple idea to adapt this method to more complex environments is that of discretizing the state space unevenly, in order to have finer discretization in parts of the state space which are more important.

Likely better results can be achieved by training for much longer and with many bins, but clearly this method does not scale well to problems with larger state spaces, as exploring them fully could prove to be too costly. Methods which update states not yet visited are needed instead.

### 5.3.2 Tuning the features

When using radial basis functions as features, we have to tune both the width of each feature, as well as the number of centers, or features. Larger values of $\sigma$ result in smaller responses as the state gets farther from the center. To avoid having spots of the state space in which there is no response, we can increase the number of features which, of course, consequently makes the training harder. The centers of the features are sampled randomly. This might generate clusters of features in uninteresting parts of the state space, which can hinder learning. Notice that, as for the discritized case, if instead we had prior knowledge about the state space, we could place finer features in its most relevant portions, and coarser ones in less relevant ones.

We repeat the experiment of the previous section for different combinations of $\sigma$ and number of features. Figure 5.11 reports the steps per episode during training, while Table 5.3 reports results during evaluation. When the number of features is low, the features are too sparse to properly describe the state space. As the number of features increases, both the average steps episode and the standard deviation between the agents decrease. Perhaps surprisingly, the second-best average is obtained with many features of large width, which intuitively should result in many redundant features. Possibly, both fully covering the state space with large features, and partly covering the state space (i.e., only its most significant portions) with smaller features, produce similar outcomes. This

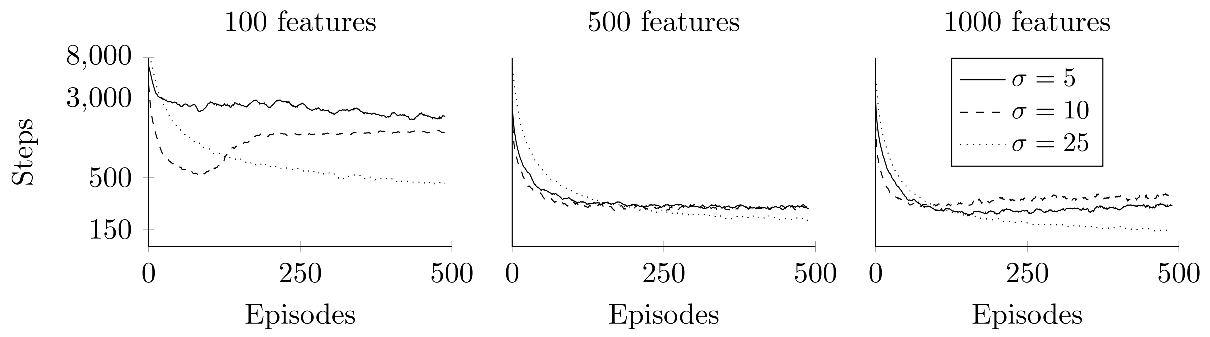

Figure 5.11: Steps per episode during training for different combinations of number and width of features, smoothed with a moving average of 10 episodes.

| Features | $\sigma$ | Average | Standard deviation |
|---|---|---|---|
| 100 | 5 | 9,229 | 2,348 |
| | 10 | 3,165 | 3,320 |
| | 25 | 6,377 | 4,159 |
| 500 | 5 | 590 | 1,876 |
| | 10 | 279 | 316 |
| | 25 | 828 | 1,157 |
| 1000 | 5 | 289 | 208 |
| | 10 | 678 | 990 |
| | 25 | 778 | 804 |

Table 5.3: Steps to reach the goal across 100 runs of evaluation with different combinations of number and width of features.

hypothesis could also explain the slightly higher variance in the case of the parameters that yielded the lowest average, since the effectiveness of the trained agent would depend on where such features are placed.

### 5.3.3 The $\lambda$ parameter

In the case of $n$-step methods and TD($\lambda$) we have a further tuning parameter indicating how farther back in the past each state has contributed to the reward obtained at the current time step. In the case of the mountain car environment, the car has to gather momentum. This process takes many time steps. Thus, previous states and actions have

| $\lambda$ | Average | Standard deviation |
|---|---|---|
| 0 | 570 | 353 |
| 0.1 | 556 | 348 |
| 0.3 | 198 | 35 |
| 0.5 | 227 | 65 |
| 0.8 | 266 | 492 |
| 0.9 | 314 | 956 |
| 0.95 | 486 | 1,640 |
| 0.99 | 665 | 2,097 |

Table 5.4: Steps to reach the goal across 100 runs of evaluation for different values of $\lambda$.

| Method | Average | Standard deviation |
|---|---|---|
| SARSA | 231 | 141 |
| Q-learning | 389 | 264 |
| One-step actor-critic | 248 | 24 |

Table 5.5: Steps to reach the goal across 100 runs of evaluation with different methods.

great importance when considering how to act in the present.

We once again repeat the same experiment with a SARSA($\lambda$) agent, using 500 features with a width of 10, as on average they yielded the best results in the previous section. Since the steps per episode during training are rather close in all cases, we only report the steps per episode during the evaluation process (Table 5.4).

As we mentioned in Section 3.3.2 in the case of $n$ for $n$-step methods, also in the case of $\lambda$ intermediate values produce better results. Here, we can also clearly notice how the value of $\lambda$ balances bias and variance. When $\lambda = 0$ we bootstrap at each time step, and thus we have high bias. As $\lambda$ approaches 1 a larger amount of previous states and actions are taken into consideration when computing the current update, consequently increasing the variance.

### 5.3.4 Comparison of approximate methods

We will now compare some of the approximate methods introduced previously on the mountain car environment. We will once again use 500 features with a width of 10. Figure 5.12 shows the steps per episode during training, whereas Table 5.5 shows the steps per episode during evaluation.

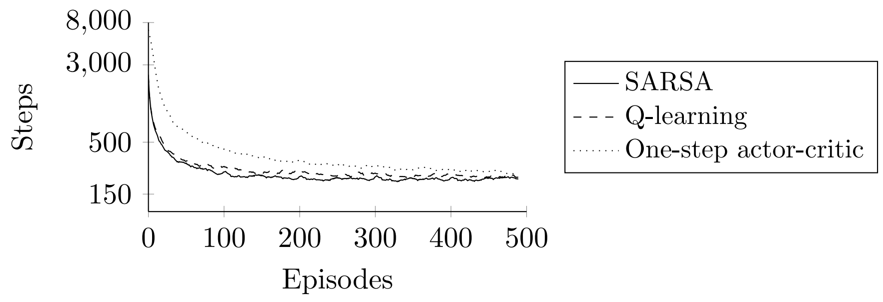

Figure 5.12: Steps per episode during training for different methods, smoothed with a moving average of 10 episodes.

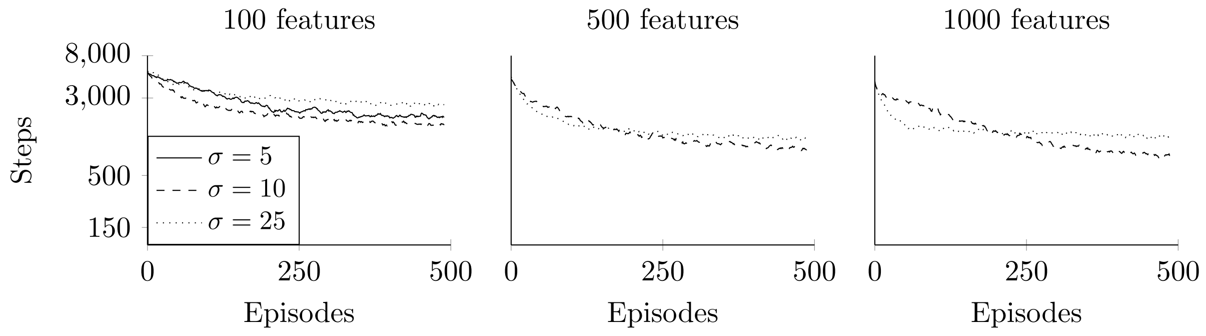

Figure 5.13: Steps per episode during training for different combinations of number and width of features, smoothed with a moving average of 10 episodes.

Although one-step actor-critic is initially lagging behind, during training the average steps per episode for all methods are very similar. Theoretically Q-learning has no guarantee of converging when approximating the value function, yet in practice, although worse than SARSA on average, it is able to reach satisfactory results. Additionally, while SARSA is slightly better than one-step actor-critic on average, the latter has a much lower variance, and achieves similar results for all trained agents.

### 5.3.5 Continuous action space

We now move onto the mountain car environment variant which has continuous actions. In this case the problem is much harder, as the agent has to not only reach the goal, but also to do so without making actions of large magnitude. As such, we report both the steps per episode and the reward per episode in Figures 5.13 and 5.14, and Table 5.6.

A standard deviation of 1 for the Gaussian policy has been used. The cases in which the

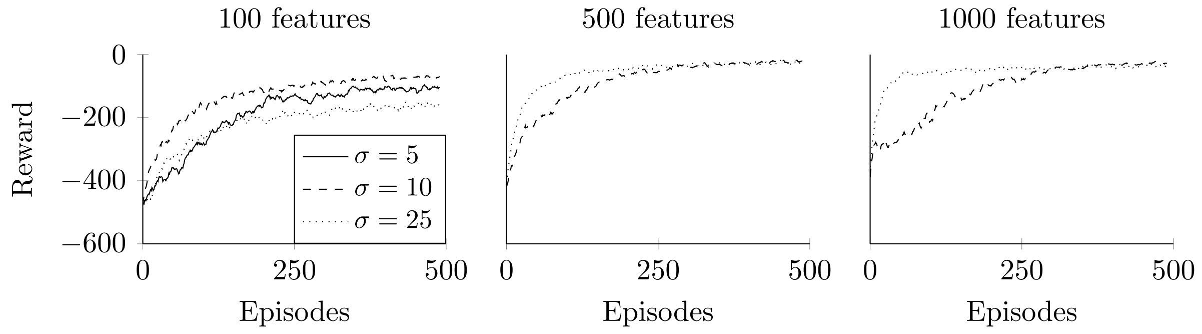

Figure 5.14: Reward per episode during training for different combinations of number and width of features, smoothed with a moving average of 10 episodes.

| Features | $\sigma$ | Average steps | SD steps | Average reward | SD reward |
|---|---|---|---|---|---|
| 100 | 5 | 1870 | 414 | -98 | 40 |
| | 10 | 1576 | 187 | -66 | 19 |
| | 25 | 2525 | 241 | -155 | 25 |
| 500 | 10 | 923 | 132 | -24 | 22 |
| | 25 | 1158 | 127 | -24 | 13 |
| 1000 | 10 | 800 | 161 | -27 | 32 |
| | 25 | 1224 | 166 | -35 | 17 |

Table 5.6: Steps to reach the goal and reward across 100 runs of evaluation for different combinations of number and width of features, smoothed with a moving average of 10 episodes.

number of features is either 500 or 1000 and the width of the features 5 have been omitted, as even with different values of standard deviation of the Gaussian policy the agent was unable to learn properly. With a standard deviation of 1 the agent would often end up with a mean action which was of great magnitude, and a low probability of selecting a better choice and getting back on track. On the other hand, choosing higher values for the standard deviation rendered the learning slow and the results lackluster. This is likely an indication that parameterzing also the standard deviation, rather than only the mean, could be beneficial. However, the standard deviation is generally approximated as $\sigma_\theta(s) \doteq \exp\left(\phi(s)^\top \theta_\sigma\right)$. This greatly limits the number of features that can be used, as the exponential can easily cause an overflow when all features have positive value (as is the case for RBFs).

The results obtained are rather similar to the ones in the case of a discrete action space

(Table 5.5). Of course, as the continuous action space variant is much harder, the number of average steps per episode is higher, but the ranking between each of the combinations of number and width of features is almost identical. It is also interesting to note that in all cases, the agents are able to learn to both reach the goal quickly and efficiently, without restricting themselves to learning just one or the other.

# Chapter 6

# Conclusions

In the first part of this work we introduced dynamic programming methods to solve optimal control problems. Then, by removing the assumption of perfect knowledge of the environment we presented more efficient methods, divided in value- and policy-based methods, also focusing on the more challenging cases of continuous state and action spaces.

The goal of the thesis was that of attempting to understand, empirically, which methods and parameters are best for a given task by testing a variety of them on different environments. As of now, no clear strategy for choosing them has been found. As such, the optimal methods and parameters can only be attained by manually testing many candidate combinations. Nevertheless, we saw that in the case of $\varepsilon$, although theoretically the class of scheduling strategies is wide, only a handful of them ensure convergence in practice. The choice of the scheduling strategy for $\varepsilon$ ultimately depends on both the specific problem at hand, and the number of training episodes. Additionally, we looked at how different methods, but also their parameters, such as the values of $\alpha$, $n$, and $\lambda$, can be interpreted as balancing terms between bias and variance. Ultimately we observed how, in the case of continuous state spaces, the choice of which features to use, and also their parameters, is once again not straightforward. Whenever possible, hand crafted problem-specific features outperform more generic ones, which instead require fine tuning.

Lastly, we report some remarks with respect to future work in the field. First, reward signals strongly influences what and how the agent learns, yet currently no obvious guideline with respect to their design has been presented. Oftentimes, especially when the task is hard to translate to a reward signal, the agent can find a way to bypass the problem by acting in unintended ways, or even discovering new techniques [31]. To address this shortcoming, several alternative approaches that include advice given by an external observer have been proposed [32, 33]. Second, although the methods presented here yield satisfactory on simpler environments with continuous state and action spaces, current research shows that we still lack approximate methods which are both online and incremental and that can be employed on very complex environments. In fact, deep learning methods applied to classic RL methods require workarounds in order to learn properly, such as batch training on large data sets [34], experience replay [35], and self play in the case of games [36]. The third and especially important aspect is that of formalizing curiosity in learning agents. Current reinforcement learning methods solve tasks known and fixed a priori, yet having agents be able to choose what to learn, possibly by having them include a signal representing how novel a state or an action is, has been shown to be beneficial in preliminary studies [37, 38]. Of course, it is also relevant to remark that giving agents freedom to explore, while helpful for training, could also prove to be unsafe in industrial applications, or even unethical, as agents get closer to artificial general intelligence.

# Appendix A

# Implementation details

In this appendix we will briefly present the structure of the implementation, as well as Gym, the library used for simulating the environments.

## A.1 Structure of the implementation

The main directory is the `core` folder. It contains the generic classes for the agents, and the implementation of both the value functions, the policies, and the learning rates.

Figure A.1 shows the hierarchy of agents. The generic dynamic programming agent has a `solve()` method which does policy evaluation and policy improvement iteratively until convergence. In the case of the reinforcement learning agents, the difference between the classes is in the arguments of the method, and what needs to be initialized (either a value function, a policy, or both). Monte Carlo methods, i.e., first- and every-visit Monte Carlo, and REINFORCE, inherit from both the `MonteCarloAgent` class as well as the `ValueBasedAgent` class for the first two, or the `PolicyBasedAgent` class for REINFORCE.

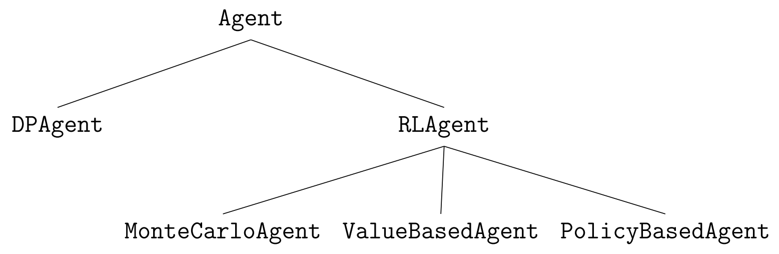

Figure A.1: The agent hierarchy.

The actual implementations of the methods are in the `agents` directory. The reinforcement learning agents are trained through the `train()` method of the `RLAgent` class. In turn, this method repeatedly calls the `episode()` method, which trains the agent for one episode. In most cases, the implementation of the whole agent simply lies in the implementation of one episode of training.

The `core` folder also contains the implementation of the value functions (both state- and action-value functions, either represented by a matrix, or linearly approximated by a weight vector), the policies (a tabular one for dynamic programming, an implicitly derived one for value-based agents, and a parameterized one for policy-based agents), and the step size. The step size can be constant, linearly decaying, or exponentially decaying. The linearly decaying learning sequence of step sizes takes the form

$$h_k \doteq \frac{h_0}{1 + (d \cdot k)}, \tag{A.1}$$

where $h_0$ is some initial value, $d$ is a decay factor, and $k$ denotes the current episode. Unless otherwise noted, values of $\alpha$ (Robbins-Monro sequences of step sizes for value-based methods), $\eta$ (learning rate for gradient methods), and $\varepsilon$ ($\varepsilon$-greedy policies) are computed using this definition.

## A.2 OpenAI Gym

The environments are emulated using OpenAI Gym [39], which is a suite of environments written in Python. We will now introduce it in brief.

Dynamic programming methods simply access the environment's dictionary `P` which, for each state and each possible action in that state, has a list of tuples, each containing: 1) the probability of transitioning to a certain next state; 2) the next state in question; 3) the reward associated with the transition; 4) a boolean indicating whether a terminal state was reached.

In reinforcement learning methods the interaction between agent and environment takes place through the `step()` method, which accepts an action, and returns a tuple similar to the previous, without the transition probability, and the addition of a boolean indicating whether the execution of the environment was truncated (generally because of a time limit).

When an environment has terminated it can be reset to its initial state via the `reset()` method. Additionally, the environment can be rendered visually with the `render()` method. The environments can also be customized by setting a predefined wrapper class as parent of our class. For example, this has been used to implement both discretized and featurized states.

# Bibliography

[1] Richard S. Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction.* MIT press, 2018.

[2] David Silver. Lectures on Reinforcement Learning. URL: `https://www.davidsilver.uk/teaching/`, 2015.

[3] R. E. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[4] Ronald A. Howard. *Dynamic Programming and Markov Processes.* MIT Press, 1960.

[5] Satinder P. Singh and Richard S. Sutton. Reinforcement Learning with Replacing Eligibility Traces. *Machine Learning*, 22(1):123–158, 1996.

[6] Satinder P. Singh, T. Jaakkola, M. L. Littman, and Cs. Szepesvári. Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms. *Machine Learning*, 38(3):287–308, 2000.

[7] Richard S. Sutton. Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3(1):9–44, 1988.

[8] Peter Dayan. The Convergence of TD($\lambda$) for General $\lambda$. *Machine Learning*, 8(3):341–362, 1992.

[9] C. J. C. H. Watkins. *Learning from Delayed Rewards.* PhD thesis, University of Cambridge, 1989.

[10] T. Jaakkola, Michael I. Jordan, and Satinder P. Singh. On the Convergence of Stochastic Iterative Dynamic Programming Algorithms. *Neural Computation*, 6:1185–1201, 1994.

[11] Harm van Seijen and Richard S. Sutton. True online TD($\lambda$). In *ICML 2014*, 2014.

[12] G. A. Rummery and Mahesan Niranjan. On-line Q-Learning Using Connectionist Systems. *Technical Report CUED/F-INFENG/TR 166*, 1994.

[13] C. J. C. H. Watkins and P. Dayan. Q-Learning. *Machine Learning*, 8(3):279–292, 1992.

[14] John N. Tsitsiklis. Asynchronous Stochastic Approximation and Q-Learning. *Machine Learning*, 16(3):185–202, 1994.

[15] Jing Peng and Ronald J. Williams. Incremental Multi-Step Q-Learning. In William W. Cohen and Haym Hirsh, editors, *Machine Learning Proceedings 1994*, pages 226–232. Morgan Kaufmann, San Francisco (CA), 1994.

[16] John N. Tsitsiklis and Benjamin Van Roy. Analysis of Temporal-Difference Learning with Function Approximation. In *Proceedings of the 9th International Conference on Neural Information Processing Systems*, NIPS'96, page 1075–1081, Cambridge, MA, USA, 1996. MIT Press.

[17] Michail G. Lagoudakis and Ronald Parr. Least-Squares Policy Iteration. *J. Mach. Learn. Res.*, 4:1107–1149, 2003.

[18] Geoffrey J. Gordon. Chattering in SARSA($\lambda$): A CMU Learning Lab Internal Report. Technical report, 1996.

[19] Geoffrey J. Gordon. Reinforcement Learning with Function Approximation Converges to a Region. In *Advances in Neural Information Processing Systems*, pages 1040–1046. The MIT Press, 2001.

[20] Leemon Baird. Residual Algorithms: Reinforcement Learning with Function Approximation. In Armand Prieditis and Stuart Russell, editors, *Machine Learning Proceedings 1995*, pages 30–37. Morgan Kaufmann, 1995.

[21] Ronald J. Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Mach. Learn.*, 8(3–4):229–256, 1992.

[22] P. Marbach and John N. Tsitsiklis. Simulation-Based Optimization of Markov Reward Processes. *IEEE Transactions on Automatic Control*, 46(2):191–209, 2001.

[23] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, page 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

[24] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation, 2015.

[25] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983.

[26] Jan Peters and Stefan Schaal. Natural Actor-Critic. *Neurocomputing*, 71(7):1180–1190, 2008. Progress in Modeling, Theory, and Application of Computational Intelligenc.

[27] Thomas Degris, Martha White, and Richard S. Sutton. Off-Policy Actor-Critic, 2012.

[28] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic Policy Gradient Algorithms. ICML'14, page I–387–I–395. JMLR.org, 2014.

[29] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* Adaptive Computation and Machine Learning. MIT Press, 1998.

[30] Andrew W. Moore. Efficient Memory-Based Learning for Robot Control. Technical report, University of Cambridge, 1990.

[31] Bowen Baker, Ingmar Kanitscheider, Todor M. Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent Tool Use from Multi-Agent Autocurricula. *CoRR*, abs/1909.07528, 2019.

[32] Jeffrey A. Clouse and Paul E. Utgoff. A Teaching Method for Reinforcement Learning. In *Proceedings of the Ninth International Workshop on Machine Learning*, ML92, page 92–101, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[33] Richard Maclin and Jude W. Shavlik. Incorporating Advice into Agents that Learn from Reinforcements. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1)*, AAAI '94, page 694–699, USA, 1994. American Association for Artificial Intelligence.

[34] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing Atari with Deep Reinforcement Learning. *CoRR*, abs/1312.5602, 2013.

[35] Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks.* PhD thesis, USA, 1992. UMI Order No. GAX93-22750.

[36] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu,

Thore Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.

[37] Jürgen Schmidhuber. Curious Model-Building Control Systems. In *[Proceedings] 1991 IEEE International Joint Conference on Neural Networks*, pages 1458–1463 vol.2, 1991.

[38] Jürgen Schmidhuber. A Possibility for Implementing Curiosity and Boredom in Model-Building Neural Controllers. In *Proceedings of the First International Conference on Simulation of Adaptive Behavior on From Animals to Animats*, page 222–227, Cambridge, MA, USA, 1991. MIT Press.

[39] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym, 2016.