POLITECNICO DI TORINO

Department of Control and Computer Engineering

Master of Science in Computer Engineering (Software Career)

Master Thesis

Deep Reinforcement Learning algorithms for autonomous systems

Design and implementation of a control system for autonomous driving task of a small robot, exploiting state-of-the-art Model-Free Deep Reinforcement Learning algorithms

Supervisors
prof. Pietro Michiardi
prof. Elena Baralis

Candidate
Piero Macaluso
matricola: s252894

Academic year 2019-2020

Contents

Chapter 1

Introduction

Autonomous systems and in particular self-driving for unsupervised robots and vehicles (e.g. self-driving cars) are becoming more and more integral part of human lives. This topic attracted much attention from both the research community and industry, due to its potential to radically change mobility and transport. In general, most approaches to date focus on formal logic methods, which define driving behaviour in annotated geometric maps. These methods can be challenging to scale, as they rely heavily on an external mapping infrastructure rather than using and understanding the local scene, leaving fully autonomous driving in a real urban environment an essential but elusive goal.

[?] In order to make autonomous driving a truly ubiquitous technology, in this thesis we focus on systems which address the ability to drive and navigate in the absence of maps and explicit rules, relying – just like humans do – on a comprehensive understanding of the immediate environment while following simple high-level directions (e.g. turn-by-turn route commands). Recent work in this area has demonstrated that this is possible on rural country roads, using GPS for coarse localization and LIDAR to understand the local scene [44].

The majority of the approaches adopted to exploit the local scene to learn how to drive, concentrate on deterministic algorithms to recognize the surroundings and select the right action (e.g. lane following problem on well-marked structured highways). However, these methods, like the previous ones, are not able to generalize proficiently in a different environment because of their deterministic nature.

Recently, Reinforcement Learning (RL) – a machine learning subfield focused on solving Markov decision process (MDP), where an agent learns to select actions in an environment in an attempt to maximize some reward function – has been shown to achieve super-human results at games such as Go [54] or chess [55], to be particularly suited for simulated environments like computer games [37], and to be a promising methodology for simple tasks with robotic manipulators [16].

Furthermore, the great fervour produced by the widespread exploitation of Deep Learning opened the doors to function approximation with neural networks and convolutional neural networks, developing what is nowadays known as Deep Reinforcement Learning.

TODO (Macaluso P.): Continue from here

In this thesis, we argue that the generality of RL makes it a useful framework to apply to autonomous driving. For this reason, we design and implement a control system for an autonomous driving task with the small robot Cozmo by Anki [1] on which to exploit state-of-the-art model-free deep reinforcement learning algorithms and discussing possible ways to make them data efficient.

1.1 Motivation

1.2 Structure of the thesis

TODO (Macaluso P.):

- Brief Description of Every Chapter

The aim of this section is to describe the main structure of the thesis.

Chapter 1 - Introduction

The current chapter contains the motivation of this work and the structure of the thesis.

Chapter 2 - Reinforcement Learning Fundamentals

The aim of this chapter is to present a description as detailed as possible about RL state-of-the-art in order to provide the reader with useful tools to enter in this research field. TODO (Macaluso P.): Da qui in poi questo capitolo è da fare

Chapter 3 - Tools and Frameworks

This chapter explains briefly what are the main tools, frameworks and languages used in the thesis. TODO (Macaluso P.): Continue this list

OpenAI Gym framework for developing and comparing Reinforcement Learning algorithms and environments using standardize interface.

Anki Cozmo it looks like a simple toy at first sight, but it hides an infinite potential under the hood, which make it a perfect candidate for the purposes of this thesis.

Chapter 4 - Design of the control system

Chapter 5 - Algorithms for Autonomous Systems

Chapter 5 - Experiments

This is the most important chapter. It shows all the results obtained during the numerous experiments with comments and speculations about them.

Chapter 6 - Conclusions

A summary of the results obtained from experiments with a specific part dedicated to future improvements.

Chapter 2

Reinforcement Learning

Reinforcement learning (RL) is a field of Machine Learning that is experiencing a period of great fervour in the world of research, fomented by recent progress in deep learning (DL) which opened the doors to function approximation developing what is nowadays known as deep reinforcement learning (deep RL). RL represents the third paradigm of machine learning alongside supervised and unsupervised learning. The idea underlying this research field is that the learning process to solve a decision-making problem consists in a sequence of trial and error where the *agent*, the protagonist of RL, could discover and discriminate valuable decisions from penalising ones exploiting information given by a *reward signal.* This interaction has a strong correlation with what human beings and animals do in the real world to forge their behaviour.

Before discussing the results of this thesis, it is good to delineate what today represents the state-of-the-art to understand the universe behind this paradigm better. Indeed, the exploration of this field of research is the main aim of this chapter: the first section begins with the definition of the notation used and with the theoretical foundations behind traditional RL, then in the second section it moves progressively towards what is deep RL through an introduction to the fundamentals of deep learning and a careful discussion of the most essential algorithms paying more attention to those used during the thesis project. The last section aims to illustrate the starting point and ideas of the thesis, drawing what the scenario of deep RL applied to autonomous systems and real-world robotic tasks is today.

The elaboration of this chapter is inspired by [53], [58], [43], [30] and [14].

2.1 Fundamentals of reinforcement learning

Reinforcement learning is a computational approach to sequential decision making. It provides a framework that is exploitable with decision-making problems that are unsolvable with a single action and need a sequence of actions, a broader horizon, to be solved. This section aims to present the fundamental ideas and notions behind this research field in order to help the reader to develop a baseline useful to approach section 2.2 on page 16 about deep reinforcement learning.

2.1.1 The reinforcement learning problem

Agent, environment and reward

The primary purpose of RL algorithms is to learn how to improve and maximise a future reward by relying on interactions between two main components: the *agent* and the *environment*.

The *agent* is the entity that interacts with the environment by making decisions based on what it can observe from the state of the surrounding situation. The decisions taken by the agent consist of *actions* (a_t). The agent has no control over the environment, but actions are the only means by which it can modify and influence the environment. Usually, the agent has a set of actions it can take, which is called *action space*. Some environments have *discrete* action spaces, where only a finite number of moves are available (e.g. $\mathcal{A} = [\text{North}, \text{South}, \text{East}, \text{West}]$ choosing the direction to take in a bidimensional maze). On the other side, there are *continuous* action spaces where actions are vectors of real values. This distinction is fundamental to choose the right algorithm to use because not all of them could be compatible with both types: according to the needs of the specific case, it may be necessary to modify the algorithm to make it compatible. The sequence of states and actions is named *trajectory* (τ): it is helpful to represent an episode in the RL framework.

The *environment (E)* represents all the things that are outside the agent. At every action received by the agent, it emits a *reward* and an *observation* of the environment.

The *reward* r_t is a scalar feedback signal that defines the objective of the RL problem. This signal allows the agent to be able to distinguish positive actions from negative ones in order to reinforce and improve its behaviour. It is crucial to notice that the reward is local: it describes only the value of the latest action. Furthermore, actions may have long term consequences, delaying the reward. As it happens with human beings' decisions, receiving a conspicuous reward at a specific time step does not exclude the possibility to receive a small reward immediately afterwards and sometimes it may be better to sacrifice immediate reward to gain more rewards later.

In this context, many features make RL different from the other two learning paradigm. Firstly, there is no supervisor: when the agent has to decide what action to take, there is no entity that can tell him what the optimal decision is in that specific moment. The agent receives only a reward signal which may delay compared to the moment in which it has to perform the next action. This fact brings out another significant difference: the importance of time. The sequentiality links all actions taken by the agent, making resulting data no more independent and identically distributed (i.i.d).

The concept of return

Given these definitions, it is noticeable that the primary purpose of the agent is to maximise the cumulative reward called *return*.

The *return* g_t is the total discounted reward starting from timestep t defined by eq. (2.1) where γ is a *discount factor*.

$$g_t = r_{t+1} + \gamma r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \quad \gamma \in [0,1) \tag{2.1}$$

Not only the fact that animal and human behaviour show a preference for immediate rewards rather than for the future ones motivates the presence of this factor, but it is also mathematically necessary: an infinite-horizon sum of rewards may not converge to a finite value. Indeed, the return function is a geometric series, so, if $\gamma \in [0,1)$, the series converges to a finite value equal to $1/(1-\gamma)$. For the same convergence sake, the case with $\gamma = 1$ makes sense only with a finite-horizon cumulative discounted reward.

States and observations

The other data emitted by the environment is the *observation* (o_t) that is related to the *state* (s_t). It represents a summary of information that the agent uses to select the next action, while the *state* is a function of the *history* the sequence of observation, actions and rewards at timestep t as shown in eq. (2.2).

$$h_t = o_1, r_1, a_1, \ldots, a_{t-1}, o_t, r_t, \qquad s_t = f(h_t) \tag{2.2}$$

The state described above is also called *agent state* s_t^a, while the private state of the environment is called *environment state* s_t^e. This distinction is useful for distinguishing fully observable environments where $o_t = s_t^e = s_t^a$, from partially observable environments where $s_t^e \neq s_t^a$. In the first case, the agent can observe the environment state directly, while in the second one, it has access to partial information about the state of the environment. Beyond the fact that this chapter will focus on fully observable environments, the distinction between state and observation is often unclear and, conventionally, the input of the agent is composed by the reward and the state as shown in fig. 2.1 on the next page.

Furthermore, a state is called *informational state* (or *Markov state*) when it contains all data and information about its history. Formally, a state is a Markov state if and only if satisfies eq. (2.3).

$$\mathbb{P}[s_{t+1}|s_t] = \mathbb{P}[s_{t+1}|s_1, \ldots, s_t] \tag{2.3}$$

It means that the state contains all data and information the agent needs to know to make decisions: the whole history is not useful anymore because it is inside the state. The environment state s_t^e is a Markov state.

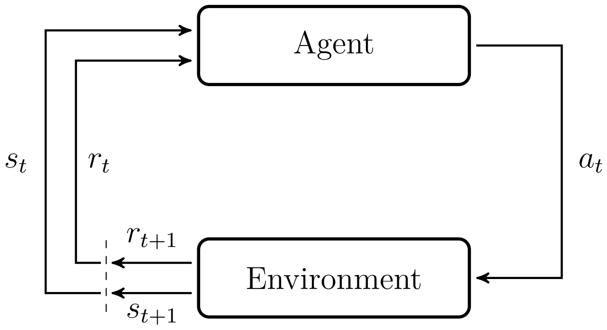

Figure 2.1. Interaction loop between Agent and Environment. The reward and the state resulting from taking an action become the input of the next iteration. [58]

The Markov decision problem

With all the definitions shown so far, it is possible to formalise the type of problems on which RL can unleash all its features: the Markov decision process (MDP), a mathematic framework to model decision processes. Its main application fields are optimization and dynamic programming.

A MDP is defined by:

$$< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$$

$$\begin{aligned} \text{where} \quad & \mathcal{S} \text{ is a finite set of states} \\ & \mathcal{A} \text{ a finite set of actions} \\ & \mathcal{P} \text{ a state transition probability matrix } \mathcal{P}^a_{ss'} = \mathbb{P}[s_{t+1} = s' | s_t = s, a_t = a] \\ & \mathcal{R} \text{ a reward function } \mathcal{R}^a_s = \mathbb{E}[r_{t+1} | s_t = s, a_t = a] \\ & \gamma \text{ a discount factor such that } \gamma \in [0{,}1] \end{aligned} \tag{2.4}$$

The main goal of an MDP is to select the best action to take, given a state, in order to collect the best reward achievable.

Policies, models and value functions

In this quick overview of central units of RL, the components that may compose the agent, the brain of the RL problem can not be missing: they are the *model*, the *policy* and the *value function*.

A *model* consist of information about the environment. These data must not be confused with the ones provided by states and observations: they make it possible to infer prior knowledge about the environment, influencing the behaviour of the agent.

A *policy* is the representation of the agent's behaviour. It is a function that describes the mapping from states to actions. The policy is represented by π and it may be deterministic $a_t = \pi(s_t)$ or stochastic $\pi(a_t|s_t) = \mathbb{P}[a_t|s_t]$. In this perspective, it is evident that the central goal of RL is to learn an optimal policy π^*. The optimal policy is a policy which can show to the agent what the most profitable way to achieve the maximum return is, what is the best action to do in a specific situation. In order to learn the nature of the optimal policy, RL exploits value functions.

A *value function* represents what is the expected reward that the agent can presume to collect in the future, starting from the current state. The reward signal represents only a local value of the reward, while the value function provides a broader view of future rewards: it is a sort of prediction of rewards. It is possible to delineate two main value functions: the *state value* function and the *action value* function.

- The *State Value Function* $V^\pi(s)$ is the expected return starting from the state s and always acting according to policy π.

$$V^\pi(s) = \mathbb{E}_{\tau\sim\pi}[g_t|s_0 = s] \tag{2.5}$$

- The *Action Value Function* $Q^\pi(s)$ is the expected return starting from the state s, taking an action a and then always acting according to policy π.

$$Q^\pi(s,a) = \mathbb{E}_{\tau\sim\pi}[g_t|s_0 = s, a_0 = a] \tag{2.6}$$

2.1.2 Bellman equations

Both eqs. (2.5) and (2.6) satisfy recursive relationships between the value of a state and the values of its successor states. It is possible to see this property deriving *Bellman equations* [4] – shown in eq. (2.7) and demonstrated in appendix A.1 on page 51 – where $s_{t+1} \sim E$ means that the next state is sampled from the environment E and $a_{t+1} \sim \pi$ shows that the policy π determines the next action.

$$\begin{aligned}
V^\pi(s_t) &= \mathbb{E}_{a_t\sim\pi, s_{t+1}\sim E}[r(s_t,a_t) + \gamma V^\pi(s_{t+1})] \\
&= \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}, r\in\mathcal{R}} P(s',r|s,a)\Big[r + \gamma V^\pi(s')\Big] \\
Q^\pi(s_t,a_t) &= \mathbb{E}_{s_{t+1}\sim E}[r(s_t,a_t) + \gamma\mathbb{E}_{a_{t+1}\sim\pi}[Q^\pi(s_{t+1},a_{t+1})]] \\
&= \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}, r\in\mathcal{R}} P(s',r|s,a)\Big[r + \gamma Q^\pi(s',a')\Big]
\end{aligned} \tag{2.7}$$

$r(s_t, a_t)$ is a placeholder function to represent the reward given the starting state and the action taken. As discussed above, the goal is to find the optimal policy π^* to exploit. It can be done using *Bellman optimality equations* defined in eq. (2.8).

$$\begin{aligned}
V^*(s_t) &= \max_a \mathbb{E}_{s_{t+1}\sim E}[r(s_t, a) + \gamma V^*(s_{t+1})] \\
&= \max_a \sum_{s'\in\mathcal{S}, r\in\mathcal{R}} P(s', r|s, a)\Big[r + \gamma V^*(s')\Big] \\
Q^*(s_t, a_t) &= \mathbb{E}_{s_{t+1}\sim E}[r(s_t, a_t) + \gamma \max_{a'}[Q^*(s_{t+1}, a')]] \\
&= \sum_{s'\in\mathcal{S}, r\in\mathcal{R}} P(s', r|s, a)\Big[r + \gamma \max_{a'} Q^*(s', a')\Big]
\end{aligned} \tag{2.8}$$

Therefore, value functions allow defining a partial ordering over policies such that

$$\pi \geq \pi' \text{ if } V_\pi \geq V_{\pi'}, \forall s \in \mathcal{S}$$

This definition is helpful to enounce the *sanity theorem*. It asserts that for any MDP there exists an optimal policy π^* that is better than or equal to all other policies, $\pi^* \geq \pi, \forall\pi$, but also that all optimal policies achieve the optimal state value function and the optimal action-value function.

The solution of the Bellman optimality equation is not linear and, in general, there is no closed-form solution. For this reason, there are many iterative methods: sections 2.1.4 and 2.1.5 on page 11 and on page 12 contain some of them.

2.1.3 Approaches to Reinforcement Learning

Every agent consists of an RL algorithm that it exploits to maximise the reward it receives from the environment. Every single algorithm has its singularity, and it could work with a specific application field which depends on the particular approach it supports. Understanding differences among these groups is useful to adequately understand what type of algorithm satisfies better the needs of a specific problem. Nowadays, RL algorithms are numerous, and drawing the complete picture behind them could be a complicated purpose. The distinctions presented in this section aims to describe the most crucial distinctions that are useful in the context of the thesis without claiming to be exhaustive.

Components of learning

The first worthy distinction between RL algorithms can be made analysing how the algorithms exploit the different components of the agent: indeed it is possible to explain the main strategies in RL using *policy*, *model* and *value function* defined in section 2.1.1 on page 7.

One of the most crucial aspects of an RL algorithm is the question of whether the agent has access to or learns the model of the environment: this element enables the agent to predict state transitions and rewards. A method is *model-free* when it does not exploit the model of the environment to solve the problem. All the actions made by the agent results from direct observation of the current situation in which the agent is. It takes the observation, does computations on them and then select the best action to take. This last representation is in contrast with *model-based* methods. In this case, the agent tries to build a model of the surrounding environment in order to infer information useful to predict what the next observation or reward would be. Both groups of methods have strong and weak sides. Ordinarily, *model-based* methods show their potential in a deterministic environment (e.g. board game with rules). In these contexts, the presence of the model enables the agent to plan by reasoning ahead, to recognise what would result from a specific decision before taking action. The agent can extract all this knowledge and learn an optimal policy to follow. However, this opportunity is not always achievable: the model may be partially or entirely unavailable, and the agent would have to learn the model from its experience. Learning a model is radically complex and may lead to various hurdles to overcome: for instance, the agent can exploit the bias present in the model, producing an agent which is not able to generalise in real environments. On the other hand, model-free methods tend to be more straightforward to train and tune because it is usually hard to build models of a heterogeneous environment. Furthermore, model-free methods are more popular and have been more extensively developed and tested than model-based methods.

The use of policy or value function as the central part of the method represents another essential distinction between RL algorithms. The approximation of the policy of the agent is the base of *policy-based* methods. The representation of the policy is usually a probability distribution over available actions. This method points to optimise the behaviour of the agent directly and may ask manifold observations from the environment: this fact makes this method not so sample-efficient. On the opposite side, methods could be *value-based.* In this case, the agent is still involved in finding the optimal behaviour to follow, but indirectly. It is not interested anymore about the probability distribution of actions. Its main objective is to determine the value of all actions available, choosing the best one. The main difference from the policy-based method is that this method can benefit from other sources, such as old policy data or replay buffer.

Learning settings

The learning setting could be *online* or *offline.* In the first case, the learning process is done in parallel or concurrently while the agent continues to gather new information to use, while the second one progresses toward learning using limited data. Generalisation becomes a critical problem in the latter approach because the agent is not able to interact anymore with the environment. In the context of this thesis, what matters is *online learning*: the learning phase is not bound to already

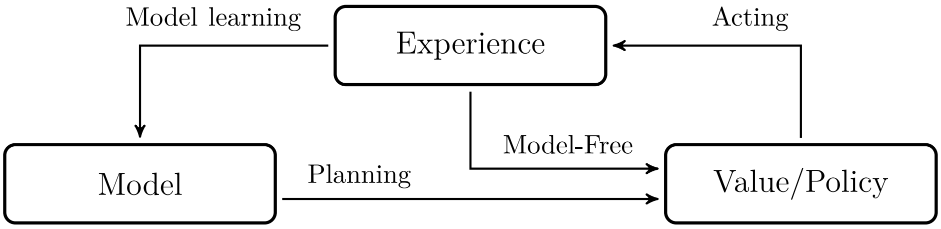

Figure 2.2. Overview of components of an agent with their relation in respect of different approaches of learning. Model-Free methods works with the experience, value functions and the policy, while model-based techniques tries to build up a model to derive value functions and policy to act in the environment.

gathered data, but the whole process goes on using both old data coming from replay buffers and brand new data obtained in the most recent episode.

Another significant difference in RL algorithms consists of the distinctive usage of the policy to learn. *On-policy* algorithms profoundly depend on the training data sampled according to the current policy because they are designed to use only data gathered with the last learned policy. On the other hand, an *off-policy* method can use a different source of valuable data for the learning process instead of direct experience. This feature allows the agent to use, for instance, large experience buffers of past episodes. In this context, these buffers are usually randomly sampled in order to make the data closer to being independent and identically distributed (i.i.d): random extraction guarantees this fact.

2.1.4 Dynamic programming

Dynamic programming (DP) is one of the approaches used to resolve RL problems. Formally, it is a general method to explain complex problems by breaking them into more manageable sub-problems. After solving all sub-problems, it is possible to sum them up in order to obtain the final solution to the whole original problem. This technique provides a practical framework to solve MDP problems and to observe what is the best result achievable from it, but it assumes to have full knowledge about the specific problem. For this reason, it applies primarily to model-based problems.

Furthermore, dynamic programming methods *bootstrap*: it means that these strategies use one or more estimated values in the update step for the same kind of estimated value, leading to results more sensitive to initial values.

Policy Iteration

The *policy iteration* aims to find the optimal policy by directly manipulating the starting policy. However, before proceeding with this process, a proper evaluation of the current policy is essential. This procedure can be done iteratively following algorithm A.1 on page 52 where θ is the parameter that defines the accuracy: the more the value is closer to 0, the more the evaluation would be precise.

Policy improvement is the second step towards policy iteration. Intuitively, it is possible to find a more valuable policy than the starting one by changing the action to select in a specific state with a more rewarding one. The key to check if the new policy is better than the previous one is to use the action-value function $Q_\pi(s, a)$. This function returns the value of taking action a in the current state s and, after that, following the existing policy π. If $Q_\pi(s, a)$ is higher than $V_\pi(s)$, so the action selected is better than the action chosen by the current policy, and consequently, the new policy would be better overall.

Policy improvement theorem is the formalisation of this fact: appendix A.2 on page 53 shows its demonstration. Thanks to this theorem, it is reasonable to act greedily to find a better policy starting from the current one iteratively selecting the action that produces the higher $Q_\pi(s, a)$ for each state.

The iterative application of policy improvement stops after an improvement step that does not modify the initial policy, returning the optimal policy found.

Value Iteration

The second approach used by dynamic programming to solve Markov decision processes is *value iteration.* Policy iteration is an iterative technique that alternate evaluation and improvement until it converges to the optimal policy. On the contrary, value iteration uses a modified version of policy evaluation to determine $V(s)$ and then it calculates the policy. The pseudocode of this method is available algorithm A.2 on page 54.

Generalised Policy Iteration

Generalised Iteration Policy (GPI) indicates the idea underlying the interaction between evaluation and improvement steps seen in value and policy iteration. Figure 2.3 on the next page reports how the two processes compete and cooperate to find the optimal value function and an optimal policy. The first step, known as policy evaluation step, exploits the current policy to build an approximation of the value function. The second step, known as policy improvement step, tries to improve the policy starting from the current value function. This iterative scheme of dynamic programming can represent almost all reinforcement learning algorithm.

2.1.5 Model-free approach

As reported in the previous section, having a comprehensive knowledge of the environment is at the foundation of dynamic programming methods. However,

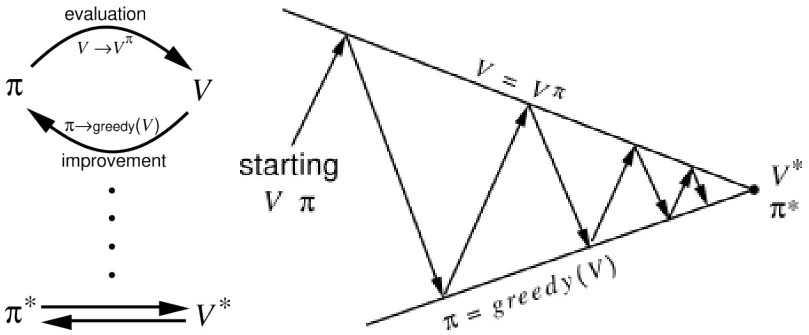

Figure 2.3. Generalised policy iteration schema [58]. Value and policy functions compete and cooperate to reach the joint solution: the optimal value function and an optimal policy.

this fact is not always accurate in practice, where it is infrequent to have a full understanding of how the world works. In these cases, the agent has to infer information using its experience, so it has to exploit model-free methods, based on the assumption that there is no prior knowledge about state transitions and rewards. This section intends to provide a brief description of two model-free approaches to prediction and control: Monte Carlo (MC) methods and Temporal-Difference (TD) ones.

Monte Carlo learning

Monte Carlo methods [58, Chapter 6] can learn from episodes of experience using the simple idea that averaging sample returns provide the value. This lead to the main caveat of these methods: they work only with episodic MDPs because the episode has to terminate before it is possible to calculate any returns. The total reward accumulated in an episode and the distribution of the visited states is used to calculate the value function while the improvement step is carried out by making the policy greedy concerning the value function.

This approach brings to light the exploration dilemma about how it is possible to guarantee that the algorithm will explore all the states without prior knowledge of the whole environment. ϵ-greedy policies are exploited instead of full greedy policy to solve this problem. An ϵ-greedy policy is a policy that acts randomly with probability ϵ and follows the policy learned with probability $(1-\epsilon)$.

Unfortunately, even though Monte Carlo methods are simple to implement and they are unbiased because they do not bootstrap, they require a high number of iteration to converge. Furthermore, they have a wide variance in their value function estimation due to lots of random decisions within an episode.

Temporal Difference learning

Temporal Difference (TD) is an approach made combining ideas from both Monte Carlo methods and dynamic programming. TD is a model-free method like MC but uses bootstrapping to make updates as in dynamic programming. The central distinction from MC approaches is that TD methods calculate a *temporal error* instead of using the total accumulated reward. The temporal error is the difference between the new estimate of the value function and the old one. Furthermore, they calculate this error considering the reward received at the current time step and use it to update the value function: this means that these approaches can work with continuing (non-terminating) environments. This type of update reduces the variance compared to Monte Carlo one but increases the bias in the estimate of the value function because of bootstrapping.

The fundamental update equation for the value function is shown in eq. (2.9), where *TD error* and *TD target* are in evidence.

$$V(s_t) \leftarrow V(s_t) + \alpha\Big(\underbrace{\overbrace{r_{t+1} + \gamma V(s_{t+1})}^{\text{TD target}} - V(s_t)}_{\text{TD error } (\delta_t)}\Big) \tag{2.9}$$

Two TD algorithms for the control problem which are worth quoting because of their extensive use to solve RL problems are *SARSA (State-Action-Reward-State-Action)* and *Q-Learning*.

SARSA is an on-policy temporal difference algorithm whose first step is to learn an action-value function instead of a state-value function. This approach leads to focus not to estimate the specific value of each state, but to determine the value of transitions and state-action pairs. Equation (2.10) represents the update function of *SARSA*, while algorithm 2.1 on the following page summarise its pseudocode.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.10}$$

Q-learning [64] is an off-policy TD control algorithm which represents one of the early revolution and advance in reinforcement learning. The main difference from SARSA is the update rule for the Q-function: it selects the action in respect of an ϵ-greedy policy while the Q-function is refreshed using a greedy policy based on the current Q-function using a max function to select the best action to do in the current state with the current policy. Equation (2.11) represents the update function of *Q-learning*, while algorithm 2.2 on the following page summarise its pseudocode.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.11}$$

Algorithm 2.1: SARSA (on-policy TD control) for estimating $Q \approx q_*$

```
Input: step size $\alpha \in (0,1]$, small $\epsilon > 0$
Initialise $Q(s,a)\ \forall\ s \in \mathcal{S}, a \in \mathcal{A}$ arbitrarily, except that $Q(\text{terminal}, \cdot) = 0$
foreach episode do
    Initialise $s_t$
    Choose $a_t$ from $s_t$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
    repeat
        Take action $a_t \rightarrow$ obtain $r_{t+1}$ and $s_{t+1}$
        Choose $a_{t+1}$ from $s_{t+1}$ using policy derived from Q (e.g. $\epsilon$-greedy)
        $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
        $s_t \leftarrow s_{t+1}$ ; $a_t \leftarrow a_{t+1}$
    until $s_t$ is terminal
end
```

Algorithm 2.2: Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

```
Input: step size $\alpha \in (0,1]$, small $\epsilon > 0$
Initialise $Q(s,a)\ \forall\ s \in \mathcal{S}, a \in \mathcal{A}$ arbitrarily, except that $Q(\text{terminal}, \cdot) = 0$
foreach episode do
    Initialise $s_t$
    Choose $a_t$ from $s_t$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
    repeat
        Take action $a_t \rightarrow$ obtain $r_{t+1}$ and $s_{t+1}$
        Choose $a_{t+1}$ from $s_{t+1}$ using policy derived from Q (e.g. $\epsilon$-greedy)
        $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$
        $s_t \leftarrow s_{t+1}$ ; $a_t \leftarrow a_{t+1}$
    until $s_t$ is terminal
end
```

Temporal Difference Lambda Learning

As reported previously, Monte Carlo and Temporal Difference learning perform updates in different ways. The first approach exploits the total reward to update the value function, while the second one, on the other hand, works with the reward of the current step. Temporal Difference Lambda, also known as TD(λ) [58, Chapter 7,12], represents a combination of these two procedures and it takes into account the results of each time step together with the weighted average of those returns. The idea of calculating TD target looking n-steps into the future instead of considering only a single step is the baseline of TD(λ). This lead to the formalisation of the

λ-weighted return G_t^λ presented in eq. (2.12).

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^{(n)} \tag{2.12}$$

TD(λ) implementation takes into account an additional variable called eligibility trace $e_t(s_t)$ which indicates how much learning should be carried out for each state for each timestep. It aims to describe how much the agent encountered a specific state recently and eq. (2.13) describes the updating rule of this value where the λ represents the trace-decay parameter.

$$e_t(s) = \gamma\lambda e_{t-1}(s) + \mathbb{1}(s = s_t) \tag{2.13}$$

2.1.6 Model-based approach

Heretofore, the focus of this section was on methods which have no prior knowledge of the environment, since this thesis grows on model-free foundations. Despite this point, it is worth to summarise the main concepts behind model-based approaches. Model-based methods gather information to enable the ability of planning, which can enhance the sample efficiency of the algorithm.

There are two primary principles to model-based learning. The first one implies to assemble a model starting from prior knowledge and to exploit it to calculate the policy and the value-function, while the second one is to infer the model from the environment by sampling experience. The central drawback of the first technique is that prior knowledge could be not as accurate as expected, leading to sub-optimal results. Consequently, the preferred way to learn is the second one.

The decisive point behind these approaches is that they are more sample-efficient concerning model-free ones: they require fewer data to learn a policy. On the other hand, the algorithm must learn the policy as well as the model: this translates to two different sources of approximation errors and an increase of computational complexity.

2.2 Deep reinforcement learning

The strategies shown so far works smoothly with systems with well-defined states and actions. In this context, it is reasonable to use lookup tables to describe the problem: state-value function V has an entry for each state while in action-value function Q has an entry for each state-action pair. It is easy to understand how this setting cannot scale up with very large MDPs: problems regarding the availability of memory arise as it becomes difficult to manage the storage of a large number of states and actions. Also, there may be obstacles concerning the slowness of learning the value of each state individually. Furthermore, the tabular form could lead to expensive computation in linear lookup and can not work with continuous action and state space.

Function approximators represent the solution to overwhelm this problem. The underlying intention is to use a vector $\theta = (\theta_1, \theta_2 \ldots, \theta_n)^T$ to estimate state-value and action-value function as shown in eq. (2.14), generalise from seen states to unseen states and finally update parameter θ using MC or TD Learning strategies.

$$\begin{aligned} V(s, \theta) &\approx V_\pi(s) \\ Q(s, a, \theta) &\approx Q_\pi(s, a) \end{aligned} \tag{2.14}$$

In these terms, function approximators can be considered as a mapping from the vector θ to the value function. This choice leads to a reduction in the number of parameters to learn and consequently to a system which can generalise better in fewer training samples.

Nowadays, since its widespread use in research, neural networks represent the most intuitive option to take as function approximator: it reduces the training time for high dimensional systems, and it requires less space in memory. This point represents the bridge between traditional reinforcement learning and recent discoveries in the theory of deep learning. Thanks to the last decade great fervour of deep learning, neural networks have become the fundamental tool to exploit as function approximator to develop deep reinforcement learning (Deep RL) which accomplished remarkable results. One of the first steps towards Deep RL and general artificial intelligence – an AI broadly applicable to a different set of various environments – was done by DeepMind with their pioneering paper [37] and the consequent [38].

Because of the nature of this work, the focus of this section will be on model-free algorithms. This section aims to explain the state-of-the-art and the leading theory behind Deep RL framework together with an overview about deep learning and to define two deep actor-critic algorithms used in the experiments of this thesis: Deep Deterministic Policy Gradient (DDPG) and Soft Actor-Critic (SAC).

2.2.1 Fundamentals of Deep Learning

Artificial Neural Networks

Deep learning (DL) is an approach to learning based on a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ parametrised with $w \in \mathbb{R}^{n_w}(n_w \in \mathbb{N})$ such that $y = f(x; w)$.

The starting point of this research field is the artificial neuron, inspired by the biological neuron from the brain of animals and human being. A neuron consists of numerous inputs called *dendrites* coming from preceding neurons. Therefore, the neuron elaborates the input and, only if the value reaches a specific potential, it *fires* through its single output called *axon*.

The neuron elaborates the inputs by taking the weighted sum, adding a bias b and applying an activation function f following the relation $y = f(\sum_n wx_i + b)$. Figure 2.4 shows the parallel comparison between the biological neuron and the artificial one. The set of parameter w needs to be adjusted to find a good parameter set: this process is called *learning*.

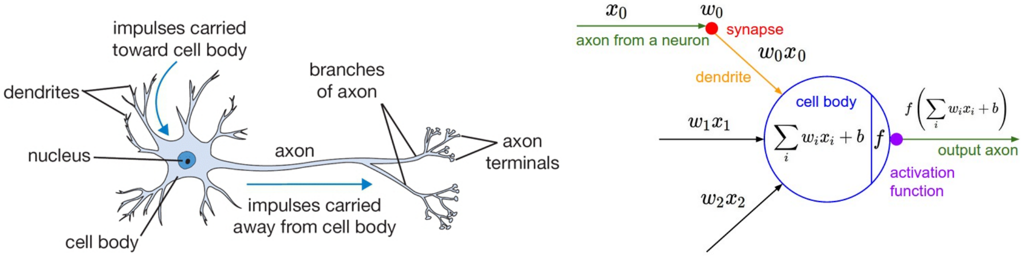

Figure 2.4. Comparison between biological neuron (left) and artificial neuron (right). The artificial neuron designs the dendrites as weighted inputs and returns the sum through an activation function. [57].

A deep neural network (NN) organises a set of artificial neurons in a series of processing layers to which correspond non-linear transformation. The whole sequence of these alterations directs the learning process through different levels of abstraction [12]. To better understand the nature of a deep neural network, it is convenient to describe a neural network with one fully-connected layer represented by fig. 2.5 on the next page.

$$\begin{aligned} h &= g(w_1 \cdot i + b_1) \\ o &= w_2 \cdot h + b_2 \end{aligned} \tag{2.15}$$

The input layer receives as input a column vector of input-features i of size $n \in \mathbb{N}$. Every value of the hidden-layer represented by a vector h of size $m \in \mathbb{N}$ is the result of a transformation of the input values given by eq. (2.15) where w_1 is a matrix of size $m \times n$ and b_1 is a bias term of size m. g is a non-linear parametric function called activation function, which represents the core of neural networks. Subsequently, the second and last transformation manipulates the hidden layer h to produce the values of the output layer following eq. (2.15) using w_2 with size $o \times m$ and b_2 with size o.

Learning process

The learning process aims to seek a set of parameters θ that results in the best possible function approximation for a specific objective. In supervised learning, the actual output Y is available for each particular input X, and it is used to update the parameters. The learning process can be carried out iteratively according to the following steps.

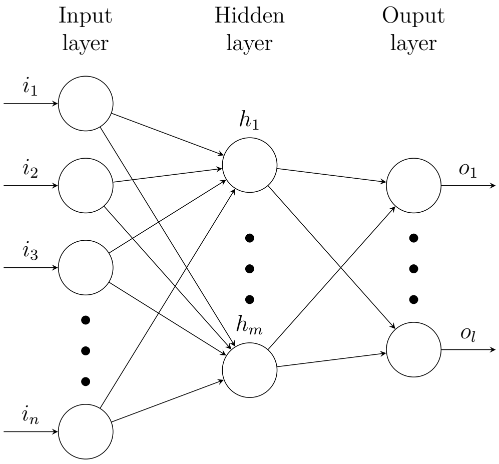

Figure 2.5. An example representation of a deep neural network with one fully-connected hidden layer.

Forward pass The input X is forwarded through the neural network and the output $Y_{pred} = f(X, \theta)$ is gathered.

Loss The resulting predicted value Y_{pred} is compared with the actual value Y computing the loss function $L(\theta)$. There are a lot of loss functions available to satisfy the particular needs of specific learning tasks. The *error* is the difference between the output of the neural network for a specific input data and the actual value: it is essential to calculate the loss function. One of the most exploited loss function is the *Mean-Squared Error* (MSE) [52] shown in eq. (2.16) which works with L2-distance.

$$\begin{aligned} L(y, \hat{y}) &= (y_\theta - y)^2 \\ J &= \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(x_i)) \end{aligned} \tag{2.16}$$

Backpropagation The next step is the computation of the global gradient of the loss function $\nabla L(\theta)$, which is carried out together with its backpropagation through the network. The backpropagation algorithm [49] calculates the local gradient of loss for each neuron in the hidden layers. The concept underlying this procedure and shown in eq. (2.17) is the *chain rule* [32], which computes derivatives of composed functions by multiplying local derivatives.

$$y = g(x), \quad z = f(g(x)), \quad \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \tag{2.17}$$

Therefore, the chain rule is exploited to propagate the calculated global gradient loss $\frac{\partial L(\theta)}{\partial \theta}$ back through the network, in the opposite direction of the forward pass. The procedure calculates the local derivatives during the forward pass, while estimates the local gradient of loss during backpropagation determining the multiplication between the local derivative and the local gradient of the loss of the connected neuron of the next layer: if the neuron has multiple connections neurons, the algorithms adds up all the gradients.

Update In this final step consists in the update of the weights of all neurons. There are many ways developed through the years to carry out the update phase, but the most common one is the gradient descent. The objective of the gradient descent is to minimise the loss function by refreshing the internal parameters of the network in the negative direction of the gradient loss: this choice leads the function approximation process closer to the minimum at each iteration. Equation (2.18) describes the update rule presented by the gradient descent where α is the learning rate. The last-mentioned parameter determines how quickly the algorithm should approach the minimum. A higher learning rate leads to a more significant step towards the minimum, which threatens to overshoot the target.

$$\theta \leftarrow \theta - \alpha \nabla_{\theta} J \tag{2.18}$$

Nowadays, the technique applied in the majority of research projects is stochastic gradient descent which combines batch learning [57] and gradient descent, but also its various improved extensions and variants, such as ADAM [25] and AdaGrad [11]: these extensions manage to improve the convergence of SGD thanks to the introduction of adaptive learning rates.

Regularization

The final aim of the learning process is to obtain a function approximator capable of generalising over data. This fact means that a neural network should show performances on unseen data comparable to the one obtained from training data. For this reason, it is necessary an appropriate trade-off between underfitting and overfitting.

A shallow approximated function and insufficient training data with a lack of diversity are the leading cause to the first situation: the network generalises on the data, but the prediction error is always too high for all data points. The phenomenon of overfitting describes the exact contrary of underfitting. The leading cause is too complex approximation function: this lead to a network which scores an excellent performance on training data, but poorly predicts unseen points.

Regularisation [5, 32] represents an approach to overcome and prevent the problem of overfitting. It works extending the loss function with a regularised term $\Omega(\theta)$ as shown in eq. (2.19) where λ is the regularisation factor.

$$L'(\theta) = L(\theta, Y, Y_{pred}) + \lambda\Omega(\theta) \tag{2.19}$$

Equations (2.20) and (2.21) show two examples of regularisation terms. The first is L^2-regularisation which exploits the squared sum of the weights θ in order to keep the weights small. The second approach is known as L^1-regularisation: in this case, large weights are less penalised, but this method leads to a sparser solution.

$$L'(\theta) = L(\theta, Y, Y_{pred}) + \lambda\frac{1}{2}||\theta||^2 \tag{2.20}$$

$$L'(\theta) = L(\theta, Y, Y_{pred}) + \lambda\frac{1}{2}||\theta|| \tag{2.21}$$

Activation function

Equation (2.22) shows the most common activation functions: in general, *ReLu* achieves better performance over a wide variety of tasks, but usually the selection of the best activation function has to be done starting from all information and requirements of the deep learning model.

$$\begin{aligned} \text{Sigmoid } &\rightarrow g(x) = \frac{1}{1+e^{-x}} \\ \text{Hyperbolic Tangent } &\rightarrow g(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \\ \text{Rectified Linear Unit (ReLu) } &\rightarrow g(x) = \max(0, x) \end{aligned} \tag{2.22}$$

Batch Learning and Normalization

The basic concept underlying *batch learning* [57] is to process a set of n training samples called also *mini-batches* in the place of a single one. This method works with the gradient averaged over all the samples in the mini-batch: it leads to a more accurate gradient reducing its variance and the training time.

Batch normalisation consists of zero-centring and rescaling all data in a specific batch, resulting in a mean of normalised data close to 0 and a variance close to 1. The algorithm presented in algorithm 2.3 on the following page is the one provided in [23]. This method computes the mean μ_β and the variance σ_β element-wise for each spatial position in the batch: $\epsilon > 0$ is a small value to avoid the division by zero. The batch normalisation layer then processes the resulting normalised value: γ and β are the parameters of this layer that are in addition to the original parameter set θ of the Neural Network in the learning process. The new learning dynamic provided by the addition of this class of layer increases the network expressivity: applying this method to the input data and the output of any hidden layer results in the reduction of the training time, better regularisation during learning and the reduction of the overfitting phenomenon.

Algorithm 2.3: Batch normalisation

```
Input: Mini-batch $\mathcal{B} = x_{1\ldots m}$; $\gamma$ and $\beta$ prameters to be learned
Output: $y_i = \mathrm{BN}_{\gamma,\beta}(x_i)$
$\mu_\beta \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                          // Mini-batch mean
$\sigma_\beta^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\beta)^2$            // Mini-batch variance
$\hat{x}_i \leftarrow \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \epsilon}}$                       // Normalisation
$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$                // Scale and shift
```

Convolutional Neural Networks

Sensory reception represents how humans and animals react to changes: it consists of sensors which process the input data and are sensitive to specific stimuli. This system inspires the architecture underlying Convolutional Neural Networks: they could handle efficiently significant input data with many applications in computer vision. Figure 2.6 displays the *LeNet-5* [33] which is capable to recognise digits in images. It represents a perfect example of a standard convolutional neural network architecture: it consists of a series of convolutional layers followed by a subsampling pooling layer. At the end of the convolutional stack, the values map into final hidden layers of the network to compute the final low-dimensional output of the network: fully-connected layers usually compose these final layers. It is possible to suppose that the first layers have to learn low-level features of the input data while succeeding layers are responsible for combining the last-mentioned features in high-level ones.

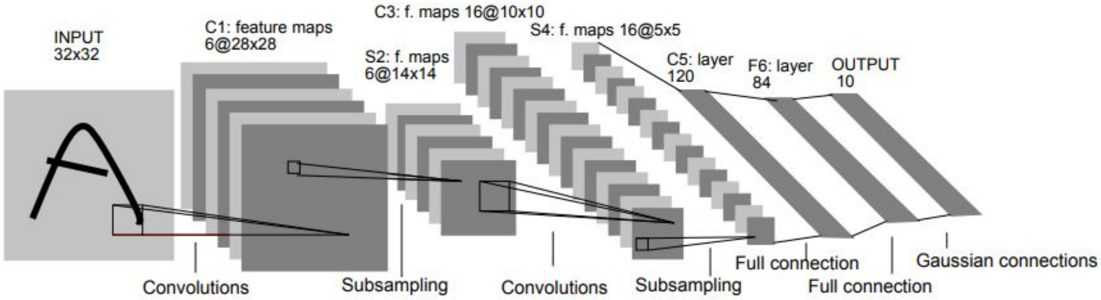

Figure 2.6. Comparison between biological neuron (left) and artificial neuron (right). The artificial neuron designs the dendrites as weighted inputs and returns the sum through an activation function. [57].

Convolutional Layer Convolutional layers [31] operates with a set of learnable filters (called kernels) with dimensions $n \times m$ smaller than the whole input image. The convolution is the basic operation employed in these class of layer: it consists in convolving each filter across the width and height of the input data and computing dot products between the values in the filter and the ones in the input at any position. The result of this operation is a 2-dimensional activation map that contains the results of that filter at every spatial position. In this context, the network can learn filters that detect specific features in the image such as edges, textures and patterns [12].

It is possible to compute the output size $W_2 \times H_2 \times D_2$ of a pooling layer using starting from the input size $W_1 \times H_1 \times D_1$ and from the hyperparameters of this class of layer: the number of filters K, their spatial extent F, the stride S and the amount of zero padding P. The resulting volume size can be calculated using the relations reported in eq. (2.23).

$$\begin{aligned} W_2 &= (W_1 - F + 2P)/S + 1 \\ H_2 &= (H_1 - F + 2P)/S + 1 \\ D_2 &= K \end{aligned} \tag{2.23}$$

The number of parameters introduced by a single kernel is equal to $F \cdot F \cdot D_1$, so the convolutional layer has a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K bias.

In a convolutional layer, the number of weights is kept small and then the computation is more efficient than the one of a fully-connected layer: small filters need fewer parameters and less work in the convolutional operation. Besides this motivation, the filters are kept small also because it makes them capable of learning small and low-level features. The great innovation behind convolutional layers is that the same neuron can recognise the related learned features even if they emerge in different locations of the image: this is the property of translation invariance of feature detection with convolutional layers.

Pooling Layer It is common to insert a pooling layer in-between successive convolutional layers. The main objective of this class of layers is to apply a downsampling filter on the input: it progressively reduces the spatial size of the representation, decreasing the number of parameters and the computational cost in the network. It is also useful to control overfitting.

It is possible to compute the size of the output $W_2 \times H_2 \times D_2$ of a pooling layer using starting from the size of the input $W_1 \times H_1 \times D_1$ and from the hyperparameters of this class of layer: the spatial extent of the filter F, the stride S. The resulting volume size can be calculated using the relations reported in eq. (2.24).

$$\begin{aligned} W_2 &= (W_1 - F)/S + 1 \\ H_2 &= (H_1 - F)/S + 1 \\ D_2 &= D_1 \end{aligned} \tag{2.24}$$

The most common types of pooling layer are the *max-pooling* and the *average-pooling* layer. Both classes return a single value for each position of the filter: the first returns the maximum value, while the second returns the average among the values in the specific section of the input.

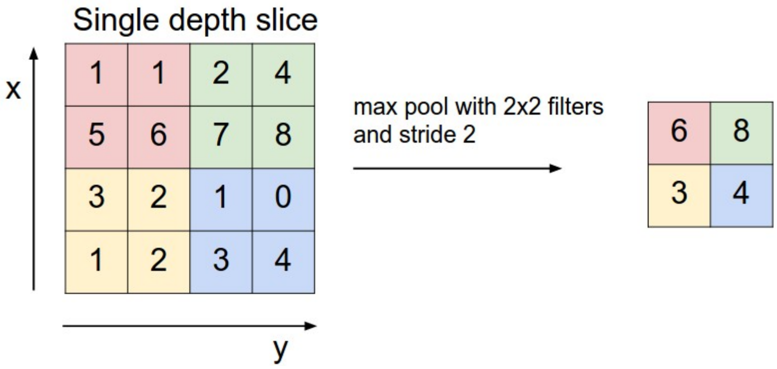

Figure 2.7. Example of a max-pooling operation with a stride of 2 and a spatial extent of 2. [57].

It is worthy of using a pooling layer in a situation where the exact feature position is not relevant but rather whether a particular feature exists in the input at all.

2.2.2 Value-based methods

The first class of algorithms to explore is value-based one. They works learning an approximator $Q_\theta(s,a)$ to infer the optimal action-value function $Q^*(s,a)$ using an objective function based on Bellman equations. The preponderance of optimisations belonging to this category is *off-policy*: this means that the optimisation step is done using all data collected during the whole training, not only with the most recent policy available. In this configuration, the information used for the learning phase could also come from exploration decision, apart from ones obtained with the most recent policy. Indeed, value-based approaches are more sample efficient because they can reuse data more efficiently, but they are considered less stable than policy gradient ones. Thanks to the relation expressed by $a(s) = \operatorname{argmax}_a Q_\theta(s,a)$, it is possible to obtain the policy learned so far from the current action-value function.

Deep Q-Network (DQN)

This algorithm grows from the ideas underlying Q-Learning [64] shown previously in section 2.1.5 on page 14. The team of DeepMind introduced the Deep Q-Network (DQN) in [37] and in the next cutting-edge paper [38]: they managed to create an algorithm capable of learning to play ATARI video-games online using raw image and pixels. It works with neural networks as a function approximator, employing convolutional layers in the first layers of the neural network and performing the optimisation with a variant of stochastic gradient descent called RMSprop [59]. The exploited neural network provides as output a probability distribution over all possible discrete actions to determine what is the best action to take.

To overcome the instability problem of value-based methods, DQN utilises two heuristics to narrow instabilities.

Target Network The presence of a second network, also called *target network*, enriches the update phase of this algorithm. Equation (2.25) defines the loss function of DQN where y_i value is computed using the target network instead of the local network. Therefore, the parameters of the target network are hard updated every $I \in \mathbb{N}$ iterations: this choice precludes instabilities and avoids divergence because of target networks parameters remain fixed for I iterations.

$$\begin{aligned} L_i(\theta_i) &= \mathbb{E}_{s,a\sim\pi}\Big[(y_i - Q(s,a;\theta_i))^2\Big] \\ y_i &= \mathbb{E}_{s'\sim E}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a] \end{aligned} \tag{2.25}$$

Experience Memory Replay Another crucial introduction in this algorithm is *experience memory replay buffer* [35]. Trajectories sampled from the environment are temporally correlated, and this could lead to overfitting the parameters of the neural network because the data are not independent and identically distributed (*i.i.d.*). The setting of this algorithm is *online* because the replay buffer stores $N_{\text{replay}} \in \mathbb{N}$ replacing old steps as new ones arrive. The experience is collected as tuples (s_t, a_t, r_t, s_{t+1}) using the ϵ-greedy policy. The learning phase samples a set of limited tuples called *mini-batch* allowing a wider set of state-action pair in the update of the network and improving the procedure in terms of variance in respect of single tuple update. Trajectories sampled from the environment are temporally correlated, and this could lead to overfitting the parameters of the neural network. Using a batch sample from the replay buffer makes the data *i.i.d.* and consequently improves the learning.

Improvements of DQN

Further investigation and speculation followed the publication of the thriving DQN. Summing up and comparing these new approaches to original DQN is the main aim of *Rainbow* [22]: it also introduces an algorithm called *Rainbow DQN* with all the techniques proposed. The following paragraphs will delineate three main improvements of DQN.

Double DQN The double DQN [20, 62] improvement can handle the intricacy of overestimation of Q-values caused by the maximisation step in eq. (2.25) on the preceding page. It works with two separate Q-Network with parameters θ and θ^- for estimating TD-target. It allows for removing the positive bias in estimating action values, leading to less overestimation of Q-learning values, improved stability and performance. In this context, the target y_i is replaced by eq. (2.26).

$$y_i = \mathbb{E}_{s'\sim E}[r + \gamma Q(s', \underset{a}{\operatorname{argmax}}\, Q(s', a; \theta_{i-1}); \theta^-_{i-1})|s, a] \tag{2.26}$$

Prioritised Experience Replay The fundamental idea underlying *prioritised experience replay* [50] is precisely to prioritise experiences that contain more crucial information than other ones. An additional value that defines the priority of a specific transition joins each tuple stored in the replay buffer: thanks to this approach, experiences with higher priority has a higher sampling probability and are more likely to remain longer in the replay buffer. It is possible to use *TD-error* to measure the importance of each tuple in the experience. A high TD-error means that the agent behaved better or worse than expected in that particular moment, and therefore, it can learn more from that specific passage.

Dueling DQN The dueling DQN architecture [63] presented in fig. 2.8 works decoupling the Q-value estimation in two distinct sequences of fully-connected layers right after convolutional layers.

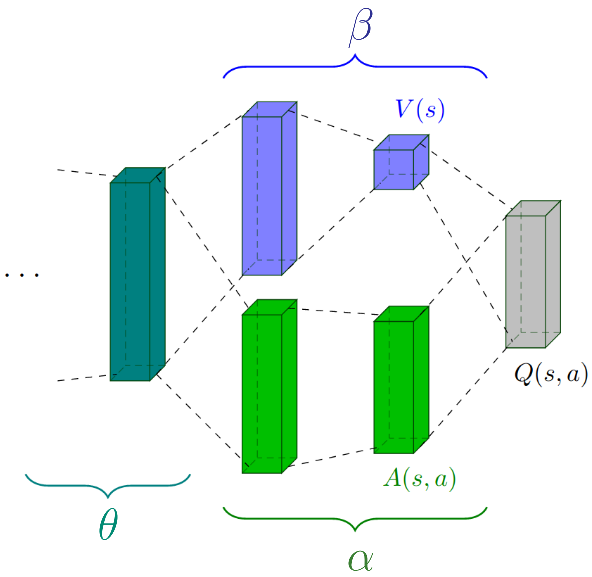

Figure 2.8. Dueling DQN architecture [63]: It consists in two stream to estimate state-value parametrised with β and advantages values parametrised with α for each action. The last layer represent the combination of these two types of values to obtain the Q-function. [14]

These streams are capable of providing separate estimates of the state-value $V(s;\theta,\beta)$ and advantage $A(s,a;\theta,\alpha)$ functions exploited in the end to obtain the Q-value function $Q(s,a;\theta,\alpha,\beta)$ estimate where θ represents the parameters of convolutional layers while α and β the ones of state-value and advantage function respectively.

The first formalisation of the Q-value function is shown by eq. (2.27), but [63] also suggests a different approach shown in eq. (2.28) with increased stability in practice.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \Big(A(s,a;\theta,\alpha) - \max_{a'\in\mathcal{A}} A(s,a';\theta,\alpha)\Big) \tag{2.27}$$

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \Big(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'\in\mathcal{A}} A(s,a';\theta,\alpha)\Big) \tag{2.28}$$

2.2.3 Policy gradient methods

Policy gradient algorithms aim to optimise the policy's performance measure in eq. (2.29) by finding a suitable policy $\pi_\theta(s|a)$ capable of generating a trajectory τ that maximises the expected rewards eq. (2.30) instead of learning a value function. Indeed the objective function in policy gradient methods consists of maximising $J(\theta)$ value by finding a proper policy updating θ parameters directly.

$$J(\theta) = \mathbb{E}\Big[\sum_{t=0}^{N} r(s_t,a_t);\pi_\theta\Big] = \sum_{\tau} P(\tau;\theta) r(\tau) \tag{2.29}$$

$$\theta^* = \operatorname*{argmax}_{\theta} J(\theta) \tag{2.30}$$

Stochastic gradient ascent is used to refresh the parameters of the policy θ. Gradient ascent is the inverse of gradient descent and updates the parameters θ_t in the positive direction of the gradient of the policy's performance measure $\nabla_\theta J(\theta)$ following eq. (2.31) where α is the learning rate which defines the strength of the steps in the direction of the gradient.

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_\theta J(\theta_t) \tag{2.31}$$

The main advantage of policy gradient approaches consists in the stability of their convergence: these methods work updating their policy directly at each time step instead of renewing value function from which to derive the policy like value-based methods. Last-mentioned approaches can lead to a radical change in the policy output even for a small change in the value function: this event can cause prominent oscillation during training. Furthermore, policy gradient algorithms can face infinite and continuous action space because the agent estimates the action directly instead of calculating the Q-value for each possible discrete action. The third feature is their ability to learn stochastic policies, useful in uncertain contexts or partially observable environments. Despite the presence of the advantages just mentioned, policy gradient methods have an substantial disadvantage: they tend to converge to a local maximum instead of the global optimum.

Actor-Critic Architecture

Actor-critic architecture, shown in fig. 2.9, represents the point of contact between value-based approaches and policy gradient methods. They are policy gradient methods basically but exploit value-function to learn the parameters θ of the policy. As its name suggests, these approaches work with two different parts called *actor* and *critic* [27]. The *actor* relates to the policy, while the *critic* deals with the estimation of a value function (e.g. Q-value function). In the context of deep reinforcement learning, they can be represented using neural networks function approximator [36]: the actor exploits gradients derived from the policy gradient theorem and adjusts the policy parameters, while the critic estimates the approximate value function for the current policy π.

Standard practice is to update both networks with the *TD-Error*, discussed in section 2.1.5 on page 14. Estimation made by the critic is useful to determine the contribution that expected values of the current and next state gives to the TD-error. Essentially, the output of the critic contributes to the update of the actor.

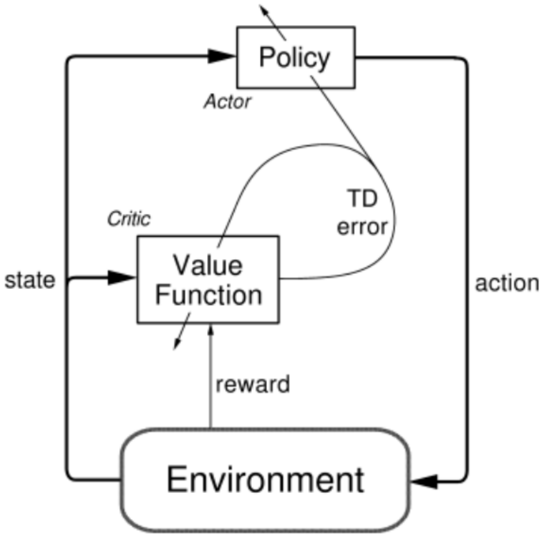

Figure 2.9. Actor-critic architecture schema: the actor represents the policy and maps the input state to an output action, while the critic represents the value function. Both networks can be updated with, e.g. the TD-error, using the contribution of the critic. It is noticeable that the actor uses the critic during the learning process [58].

2.2.4 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) [34] is a policy gradient algorithm that works learning a Q-function and a policy, and that grows from the deterministic policy gradient algorithm (DPG) [56]. It is a model-free, off-policy, actor-critic algorithm which utilises deep function approximators to learn policies in high-dimensional, continuous action spaces. It can be applied to situations that can not be solved using *DQN* algorithm [38] because of the presence of continuous action spaces. A fine discretisation of the action space to adapt the situation to DQN would lead to an explosion in the number of discrete actions and the curse of dimensionality. *Bellman equation* and *Q-learning* are integral parts of this algorithm. The algorithm concurrently learns a Q-value function and a policy: it uses off-policy data and the Bellman equation to learn the Q-value function and uses the Q-value function to learn the policy.

Usually, in Reinforcement Learning, if the optimal action-value function $Q^*(s, a)$ is known, then in any given state, the optimal action $a^*(s)$ can be found by solving eq. (2.32).

$$a^*(s) = \arg\max_a Q^*(s, a) \tag{2.32}$$

When the number of discrete actions is finite, calculating the max poses no problem, because the Q-values can be calculated for each action separately, then directly compared. However, when the action space is continuous, this process becomes highly non-trivial: it would need to be run at every step of the episode, whenever the agent wants to take any action in the environment, and this can not work.

Because the action space is continuous, the function $Q^*(s, a)$ is presumed to be differentiable concerning the action argument. For this reason, an efficient, gradient-based learning rule for a policy $\pi(s)$ which exploits that fact can be set up, approximating it with $\max_a Q(s, a) \approx Q(s, \pi(s))$.

Target Networks

DDPG algorithm exploits 4 neural networks: the *local actor*, the *local critic*, the *target actor* and the *target critic*. Actor networks aim is to approximate the policy using parameters θ while critic networks approximate the Q-Value function using parameters ϕ.

Initially, actor and critic networks have both randomly initialised parameters. Then the local actor – the current policy – starts to propose actions to the agent, given the current state, starting to populate the experience replay buffer.

When the replay buffer is big enough, the algorithm starts to sample randomly a mini-batch of experiences for each timestep t. This mini-batch is used to update the local critic minimising the Mean Squared Error (MSE) between the local Q-value and the target one shown in eq. (2.33) on the next page where $y_i = y_t$ given by eq. (2.36) on the following page and to update the actor policy using the sampled policy gradient defined in eq. (2.39) on the next page.

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\phi))^2 \tag{2.33}$$

We can imagine the target networks as the *labels* of supervised learning.

Also the target networks are updated in this *learning step*. A mere copy of the local weights is not an efficient solution, because it is prone to divergence. For this reason, a *soft* target updates is used. It is given by eq. (2.34) where $\tau \ll 1$.

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta \tag{2.34}$$

Learning Equations

The two fundamental functions in Reinforcement Learning exploited in DDPG are the *action-value function* – see eq. (2.6) on page 8 – and the correspondent *Bellman equation* – see eq. (2.7) on page 8. If this policy is deterministic we can describe it as a function $\pi : \mathcal{S} \leftarrow \mathcal{A}$ obtaining eq. (2.35) which depends only on the environment.

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1}\sim E}[r(s_t, a_t) + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1}))] \tag{2.35}$$

This means that it is possible to learn Q^π off-policy, using transition generated by a different stochastic behaviour policy β.

Focusing more and more on DDPG, the Bellman equation is the starting point for learning an approximator to $Q^*(s, a)$ of Q-Learning. The approximator is parametrised by ϕ and the value network is updated and optimised by minimising the loss defined in eq. (2.36) where d_t is a flag which indicates whether the state s_{t+1} is terminal.

$$\begin{gathered} L(\phi) = \mathbb{E}_{s_t\sim\rho^\beta, a_t\sim\beta, r_t\sim E}[(Q(s_t, a_t|\phi) - y_t)^2] \\ y_t = r(s_t, a_t) + \gamma(1-d_t)Q(s_{t+1}, \pi(s_t+1)|\phi) \end{gathered} \tag{2.36}$$

It is clear from the eq. (2.36) that the loss is calculated starting from the transitions generated by the policy β. For this reason a great importance in this algorithm is given to *replay buffer* – see section 2.2.2 on page 25 – and *target networks* – see section 2.2.4 on the previous page.

From the policy perspective, the objective is to maximise eq. (2.37) calculating the policy loss through the derivative of the objective function with respect to the policy parameter eq. (2.38). However, since the algorithm is updating the policy in an off-policy way with batches of experience, it is possible to use the mean of the sum of gradients calculated from the mini-batch eq. (2.39) on the following page.

$$J(\theta) = \mathbb{E}[Q(s, a)|_{s=s_t, a=\pi(s_t)}] \tag{2.37}$$

$$\nabla_\theta J(\theta) \approx \nabla_a Q(s, a)\nabla_\theta \pi(s|\theta) \tag{2.38}$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_i \left[\nabla_a Q(s,a|\phi)|_{s=s_i,a=\pi(s_i)} \nabla_\theta \pi(s|\theta)|_{s=s_i}\right] \tag{2.39}$$

Algorithm 2.4 on the next page shows the pseudocode of the DDPG algorithm.

Exploration vs. Exploitation

In reinforcement learning for discrete action spaces, exploration is done selecting a random action (e.g. epsilon-greedy). For continuous action spaces, exploration is done adding noise to the action itself. In [34], the authors use Ornstein-Uhlenbeck process [61] to add noise to the action output $a_t = \pi(s_t|\theta) + \mathcal{N}$. After that the action is clipped in the correct range.

Hyperparameters

Exploration noise The exploration noise consists of two sets of parameters. The first set refers to the Ornstein-Uhlenbeck Process noise parameters π, σ, θ as reported in [61]. The second one consists of the parameters of ϵ, a small value to decrease the impact of the noise on the action. Equation (2.40) describes how the impact of the noise decreases in function of the current episode number e, where ϵ_{start} represent the starting value of the noise and ϵ_{end} the final one.

$$\epsilon = \epsilon_{\text{start}} - (\epsilon_{\text{start}} - \epsilon_{\text{end}}) \min\left(1.0, \frac{e}{\epsilon_{\text{decay}}}\right) \tag{2.40}$$

Replay buffer The parameters available for replay memory are its maximum size, its minimum size to start the learning phase and the mini-batch size to sample for each learning step.

Neural Network The neural network can be considered as a whole complex hyperparameter because it is possible to select among different layers to exploit for specific problems – e.g. the number of layers, the type of layers, the number of hidden features. Given the network architecture, the primary neural networks hyperparameters are the learning rate α and the update method – e.g. Adaptive Momentum Estimation (ADAM).

Learning Update The parameters of the learning phase of the algorithm are mainly two. The first one is γ, the main parameter in the reinforcement learning framework, which characterises the discounted return. The second is the soft target update parameter τ, which determinates the entity of the update of the network at each learning step.

Algorithm 2.4: DDPG Algorithm [34]

```
Input: Initial critic network parameter $\phi$ and actor network parameter $\theta$
Initialise target network weights $\bar{\phi} \leftarrow \phi$, $\bar{\theta} \leftarrow \theta$
Initialise a random process $\mathcal{N}$ for action exploration
for episode = 1, M do
    Receive the initial observation state $s_t \leftarrow s_1$
    repeat
        Select action $a_t = \pi(s_t|\theta) + \mathcal{N}$ and clip results
        Execute action $a_t$ and obtain tuple $(r_t, s_{t+1}, d_t)$
        Store transition $(s_t, a_t, r_t, s_{t+1}, d_t)$ in $\mathcal{D}$
        if it is time to update then
            Sample random minibatch of $N$ transitions $(s_t, a_t, r_t, s_{t+1}, d_t)$ from $\mathcal{D}$
            Compute the target $y_i = r_i + \gamma(1 - d_i)Q'(s_{i+1}, \pi'(s_{i+1}|\theta)|\phi)$
            $\phi \leftarrow \phi - \lambda_Q \nabla_\phi J_Q(\phi)$                // Update critic
            $\theta \leftarrow \theta - \lambda_\pi \nabla_\theta J_\pi(\theta)$                 // Update actor
            $\phi \leftarrow \tau\phi + (1 - \tau)\phi$                   // Soft update critic
            $\theta \leftarrow \tau\theta + (1 - \tau)\theta$                   // Soft update critic
        end
        $s_t \leftarrow s_{t+1}$
    until $s_t$ is terminal
end
Output: Optimised parameters $\theta$ and $\phi$
```

2.2.5 Soft Actor-Critic (SAC)

Soft Actor-Critic (SAC) [18, 19] combines the off-policy actor-critic setup with a stochastic policy (actor), devising a bridge between stochastic policy optimization and DDPG-style approaches. As DDPG, SAC can work in situations characterised by the presence of continuous action spaces, and it is a model-free, off-policy and actor-critic algorithm.

SAC algorithm can overcome some of the problems of DDPG. The latter can achieve excellent performance, but the interaction between the deterministic actor-network and the Q-function makes it difficult to stabilise and brittle concerning hyperparameters and other kinds of tuning [10, 21]. The learned Q-function begins to dramatically overestimate Q-values, which then leads to the policy breaking because it exploits the errors in the Q-function. For this reason, SAC exploits *Clipped Double-Q Learning* also used by Twin Delayed DDPG (TD3) [15]. It learns two Q-functions instead of one and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.

Another feature of SAC is *entropy regularization* [66, 60, 46, 13, 17]. The policy is trained to maximise a trade-off between expected return and entropy, a measure of randomness in the policy. This peculiarity is strongly related to the exploration-exploitation trade-off: increasing entropy results in more exploration, which can accelerate learning later on, but it can also prevent the policy from prematurely converging to a local optimum.

Target Networks

SAC algorithm exploits 5 neural networks: the local stochastic policy network with parameter θ, two local Q-Networks with parameters ϕ_1, ϕ_2 respectively, two target Q-Networks with parameters $\bar{\phi_1}$ and $\bar{\phi_2}$ respectively. Their behaviour is the same as the one of DDPG target network: the algorithm updates the target networks following eq. (2.34) on page 30.

Algorithm 2.5 on page 36 shows the pseudocode of the SAC algorithm.

Entropy-Regularised Reinforcement Learning

Entropy represents the average rate at which a stochastic source of data produces information. It is, in simple terms, a quantity which describes how random a random variable is. The motivation behind the use of entropy is that when the data source produces a low-probability value, the event carries more information than when the source data produces a high-probability value.

Let x be a random variable with probability mass or density function P. The entropy $\mathcal{H}$ of x is computed from its distribution P according to eq. (2.41).

$$\mathcal{H}(P) = \mathbb{E}_{x\sim P}[-\log P(x)] \tag{2.41}$$

In *entropy-regularised* reinforcement learning the standard objective is generalised by augmenting it with entropy. The agent gets a bonus reward at each time step proportional to the entropy of the policy at that timestep. Assuming an infinite-horizon discounted setting, this changes the RL problem as shown in eq. (2.42) where $\alpha > 0$ is the temperature parameter that determines the relative importance of the entropy term controlling the stochasticity of the optimal policy.

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t \Big(R(s_t, a_t, s_{t+1}) + \alpha\mathcal{H}(\pi(\cdot|s_t)) \Big)\right] \tag{2.42}$$

It is clear that the standard maximum expected return can be retrieved in the limit as $\alpha \to 0$.

From eq. (2.42) it is possible to derive *state-value function* $V^\pi(s)$ and *action-value function* $Q^\pi(s,a)$ as shown in eq. (2.43) and eq. (2.44) on the next page.

$$V^\pi(s) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t \Big(R(s_t, a_t, s_{t+1}) + \alpha\mathcal{H}(\pi(\cdot|s_t)) \Big) \middle| s_0 = s\right] \tag{2.43}$$

$$Q^{\pi}(s,a) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^{t}R(s_t,a_t,s_{t+1}) + \alpha\sum_{t=1}^{\infty}\gamma^{t}\mathcal{H}(\pi(\cdot|s_t))\middle| s_0 = s, a_0 = a\right] \tag{2.44}$$

From these equations is possible to derive the connection between state-value and action-value function given by eq. (2.45) and the Bellman equation given by eq. (2.46).

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi}[Q^{\pi}(s,a)] + \alpha\mathcal{H}(\pi(\cdot|s)) \tag{2.45}$$

$$\begin{aligned} Q^{\pi}(s,a) &= \mathbb{E}_{s'\sim P, a'\sim\pi}[R(s,a,s') + \gamma(Q^{\pi}(s',a') + \alpha\mathcal{H}(\pi(\cdot|s')))] \\ &= \mathbb{E}_{s'\sim P}[R(s,a,s') + \gamma V^{\pi}(s')] \end{aligned} \tag{2.46}$$

Learning Equations

SAC algorithm learns a policy π_θ with θ parameter set and two Q-functions Q_{ϕ_1}, Q_{ϕ_2} with ϕ_1 and ϕ_2 parameter sets respectively. The state-value function is implicitly parametrised through the soft Q-function parameters thanks to eq. (2.47). In [18] a function approximator for this function was introduced, but later [19] the authors found it to be unnecessary.

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{a\sim\pi}[Q^{\pi}(s,a)] + \alpha\mathcal{H}\left(\pi(\cdot|s)\right) \\ &= \mathbb{E}_{a\sim\pi}[Q^{\pi}(s,a) - \alpha\log\pi(a|s)] \\ &\approx Q^{\pi}(s,\tilde{a}) - \alpha\log\pi(\tilde{a}|s), \qquad \tilde{a}\sim\pi(\cdot|s). \end{aligned} \tag{2.47}$$

Learning Q Q-functions are learned by Mean Squared Bellman Error (MSBE) minimization, using a target value network to form the Bellman backups using eq. (2.48). Equation (2.47) implicitly parametrise the state-value function.

$$J_Q(\phi_i) = \mathbb{E}_{(s_t,a_t)\sim\mathcal{D}}\left[\frac{1}{2}\left(Q_{\phi_i}(s_t,a_t) - \left(r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1}\sim p}V_{\bar{\phi}_i}(s_{t+1})\right)\right)^2\right] \tag{2.48}$$

The update shown makes use of target soft Q-function with parameters ϕ_i, which are calculated, like in the DDPG algorithm, as an exponentially moving average of the soft Q-function parameters [38]. It can be optimised using stochastic gradients.

Learning the Policy It is possible to derive SAC starting from the definition of soft policy iteration demontrated in [19, Section 4]. In particular, the policy has to be learned starting from the minimization of the expected KL-divergence [28, 29] and exploiting eq. (2.49).

$$J_\pi(\theta) = \mathbb{E}_{s_t\sim\mathcal{D}}[\mathbb{E}_{a_t\sim\pi_\theta}[\alpha\log\pi_\theta(a_t|s_t) - Q_\phi(s_t,a_t)]] \tag{2.49}$$

As [19] reports, there are several options for the minimization of $J_\pi(\theta)$, but the most straightforward one using neural network as function approximator is to apply the *reparametrization trick*. It works reparametrising the policy using a neural network transformation following eq. (2.50) where ϵ_t in a noise vector sampled from some fixed distribution.

$$a_t = f_\theta(\epsilon_t; s_t) \tag{2.50}$$

Therefore, it is possible to rewrite the expectation over actions in eq. (2.49) on the previous page into an expectation over noise as in eq. (2.51).

$$J_\pi(\theta) = \mathbb{E}_{s_t \sim \mathcal{D}, \epsilon_t \sim \mathcal{N}}[\alpha \log \pi_\theta(f_\theta(\epsilon_t; s_t)|s_t) - Q_\phi(s_t, f_\theta(\epsilon_t; s_t))] \tag{2.51}$$

To get the policy loss, the final step is to substitute Q_ϕ with one of our function approximators: the choice falls $\min_{i=1,2} Q_{\phi_i}(s_t, f_\theta(\epsilon_t; s_t))$, as suggested from the authors of [19].

Exploration vs. Exploitation

SAC algorithm trains a stochastic policy using *entropy regularization*. α is the entropy regularisation coefficient which is the parameter that explicitly controls the exploration-exploitation trade-off. A higher α corresponds to more exploration, while a lower α corresponds to more exploitation.

This parameter has fundamental importance in the algorithm, and it may vary from environment to environment. Choosing the optimal α parameter is a non-trivial task that could require careful tuning in order to find the one which leads to the stablest and highest-reward learning. In [19, Section 5], the authors formulated a different maximum entropy reinforcement learning algorithm to overcome this problem. Forcing the entropy to a fixed value is a weak solution because the policy should be free to explore more where the optimal action is uncertain, and to exploit the learned mapping in states with a more clear optimal action. The gradients are computed using eq. (2.52)

$$J(\alpha) = \mathbb{E}_{a_t \sim \pi}[-\alpha \log \pi(a_t|s_t) - \alpha\bar{\mathcal{H}}] \tag{2.52}$$

During the test phase, the algorithm uses the mean action instead of a sample from the distribution learned. This choice tends to improve performance over the original stochastic policy, allowing to see how well the policy exploits what it has learned.

Hyperparameters

Entropy regularisation parameter The only parameter of this section is α. It can be set as constant through all the training or it can be learned thanks to the approach described in [19]. Further details in section 2.2.5.

Replay buffer See section 2.2.4 on page 31.

Neural Network See section 2.2.4 on page 31.

Learning Update See section 2.2.4 on page 31.

Algorithm 2.5: Soft Actor-Critic [19]

```
Input: Initial policy parameter $\theta$ and Q-function parameters $\phi_1, \phi_2$
Initialise target network weights $\bar{\phi}_1 \leftarrow \phi_1, \bar{\phi}_2 \leftarrow \phi_2$
Initialise an empty replay buffer $\mathcal{D}$
for episode = 1, M do
    Receive initial state $s_t \leftarrow s_1$
    repeat
        Observe state $s_t$ and select action $a_t \sim \pi_\theta(\cdot|s_t)$
        Execute $a_t$ and obtain tuple $(r_t, s_{t+1}, d_t)$
        Store $(s_t, a_t, r_t, s_{t+1}, d_t)$ in replay buffer $\mathcal{D}$
        $s_t \leftarrow s_{t+1}$
    until $s_t$ is terminal
end
if it is time to update then
    Sample random minibatch of $N$ transitions $(s_t, a_t, r_t, s_{t+1}, d_t)$ from $\mathcal{D}$
    Calculate targets
    $\phi_i \leftarrow \phi_i - \lambda_Q \nabla_{\phi_i} J_Q(\phi_i)$, for $i \in \{1, 2\}$
    $\theta \leftarrow \theta - \lambda_\pi \nabla_\theta J_\pi(\theta)$
    $\alpha \leftarrow \alpha - \lambda \nabla_\alpha J(\alpha)$
    $\bar{\phi}_i \leftarrow \tau\phi_i + (1 - \tau)\bar{\phi}_i$, for $i \in \{1, 2\}$
end
Output: Optimised policy parameter $\theta$ and Q-function parameters $\phi_1, \phi_2$
```

2.3 Related Work

A truly inspiring work for this thesis is [24] where the authors show, probably for the first time, that deep reinforcement learning is a viable approach to autonomous driving. Nowadays, most approaches focus on formal logic which determines driving behaviour using annotated 3D geometric maps: the external mapping infrastructure intuitively makes this approach limited to the models and the representation of the surroundings. This technique is not able to scale up efficiently because of last-mentioned strong dependencies.

The fundamental concept underlying [24] to make autonomous driving systems a ubiquitous technology is the design of a system which can drive relying - just like humans - on a comprehensive understanding of the immediate environment [2]. [44] is useful to motivate the research in this direction because it represents an example of autonomous vehicle navigation exploiting GPS for coarse localisation and LIDAR to understand the local scene instead of detailed prior maps. Many sensors have been developed through the years to gather information and observations increasingly sophisticated. However, the major problem is the massive budget needed to afford all these technologies. The extraordinary results obtained by [24] are not based on intelligent sensing techniques, but on the usage of a monocular camera image together with vehicle speed and steering angle. They decided to apply the model-free approach of reinforcement learning because it is exceptionally general and useful to solve tasks that are complex to model correctly. As discussed previously in this chapter, model-based algorithms tend to be more data-efficient than model-free ones; however, the quality of the adopted model limits the results [8].

The authors decided to exploit Deep Deterministic Policy Gradient (DDPG) [34]. Firstly, they developed a 3D driving simulator using Unreal Engine 4 to tune reinforcement learning hyperparameters such as learning rates and number of gradient steps to take after each training episode. Then, they tried to apply the DDPG algorithm in the real world using the parameters learnt. They also did some experiments using a compressed state representation provided by a Variational Autoencoder [26, 48]. The excellent results obtained in [24] have been exceeded by the same authors in [65] that shows astonishing performances driving on narrow and crowded urban road never-seen during training.

Starting from all these outlined ideas, we decided to investigate ways and approach to autonomous driving with reinforcement learning without using simulators or prior data in order to make a step towards the so-called *reinforcement learning in the wild* [7]. The most popular and prominent achievements in reinforcement learning to date consist of experiments done using simulated environments or ones which exploit the knowledge acquired in simulated environments in real ones. The team of the DeepMind manages to develop smart agents capable of performing super-human results in numerous games and videogames such as *Go* [54, 55], while OpenAI engineers an agent capable of beating the world champion of the multiplayer Dota 2 game [41, 42]. These problems are not easy to solve but have the benefit of having a training environment equal to the test one. This fact does not apply to the approach which exploits simulated experiments results in real environments.

The last-mentioned method encloses a critical caveat: the awareness that the simulator has limits. Unlike what happens in some environments where there are many similarities between the simulator and the environment – e.g. games and videogames –, in harder environments – e.g. autonomous vehicle on public roads, ads marketplaces, biology, and applications around human behaviour – is very difficult to get a simulator capable of reproducing with high-fidelity the wild environment because of approximations that could mislead the learning. On the other hand, learning directly on the environment need a fast learning cycle because of the significant number of interaction with the environment and, above all, cheap or low-risk exploration cost: millions of self-driving episodes in simulator ending with a crash of the car has a small cost in respect of one episode with the same ending crash in the real world.

Between the end of 2018 and the beginning of 2019, UC Berkeley and Google developed jointly a state-of-the-art off-policy model-free reinforcement learning algorithm called soft actor-critic (SAC) (see section 2.2.5 on page 32). The critical goal is to provide a deep RL algorithm suitable for real-world problems and the related new challenges that they pose. They outlined the desired properties of the ideal deep reinforcement learning algorithm:

Sample Efficiency: the process of learning in the real world can be a very time-consuming task. For this reason, it is desirable a functional sample complexity to learn skills successfully.

No sensitive hyperparameters: as mentioned before, hyperparameter tuning could be a tough task to complete in real-world experiments as it could require numerous repetitions in situations where the cost in terms of time and money could be burdensome. The proposed solution of the authors is maximum entropy RL which provides a robust framework that minimises the need for hyperparameter tuning.

Off-policy learning: this learning approach allows the reuse of collected data for various tasks. This possibility helps the process of prototyping a new task when adjusting parameters or choosing a different reward function.

Soft actor-critic (SAC) represents a deep RL algorithm capable of satisfying the previously mentioned requirements. The authors of [18, 19] revealed that the algorithm is capable of solving real-world robotic tasks in a conspicuous but acceptable number of hours, showing its robustness to hyperparameters and working on a large variety of simulated environments always using the same set of parameters. Not only it shows great performances in numerous challenging tasks compared to deep deterministic policy gradient (DDPG), twin delayed deep deterministic policy gradient (TD3) [15] and proximal policy optimisation (PPO) [51], but reveals its power by solving three tasks from scratch without relying on simulators or demonstrations [47].

The real-world robotic task involving a 3-finger dexterous robotic hand to manipulate an object similar to a sink faucet with a coloured end presented in [19, Section 7.3] shows similar concepts to the task analysed in this thesis for what concerns the framework of deep reinforcement learning exploited. The algorithm exploits raw RGB images and processes them through a convolutional neural network. The final goal of the robot is to rotate the valve into the correct position – with the coloured part pointing to the right – starting from a random position for each episode. The results obtained represents one of the most sophisticated real-world robotic manipulation tasks learned end-to-end with deep reinforcement learning, starting from raw images and without any previous simulation or pre-training phase.

Taking all arguments into account, we decided to follow [24] implementing a similar self-driving framework based on the design of a control system for a small toy robot called Anki Cozmo (see **??** on page ??). Therefore, we formalise the autonomous driving learning problem as a Markov decision process to enable the application of reinforcement learning algorithms, taking into account the improvements mentioned in this section about model-free reinforcement learning algorithms applied to real-world robotic problems. **??** on page ?? provides a detailed description of the choice made and the system design.

Chapter 3

Tools and Frameworks

This chapter aims to describe the main tools and frameworks used in this thesis. The first section will describe the OpenAI Gym framework, a central toolkit for developing and comparing reinforcement learning algorithms. The second section will outline PyTorch, an optimised tensor library for deep learning using GPUs and CPUs used to build up the convolutional neural network of this work, while the explanation and description of the Anki Cozmo robot used for the autonomous self-driving experiment will occupy the last part of this chapter.

3.1 OpenAI Gym

Nowadays, OpenAI Gym [39, 40, 6], released in 2016 with its public beta, is one of the most popular toolkits and frameworks in the reinforcement learning scenario. A brief analysis of RL research and RL features could be useful to outline the motivations underlying the need for a reinforcement learning framework.

As reported previously in chapter 2 on page 4 reinforcement learning is a subfield of machine learning dedicated to the world of decision making and motor control, studying how an agent can learn and improve to achieve a specific goal in a complex, usually unknown environment. RL is becoming more and more attractive for researchers because of its visionary property of being very general. An RL algorithm can be exploited to control a robot's motor in order to make it capable of running or jumping, play a videogame or board game, make critical business decisions like pricing and inventory management, but also learn how to invest in financial trading environments. The general feature of reinforcement learning became engaging thanks to the remarkable results achieved in many challenging environments, as reported in chapter 2 on page 4.

Despite these appealing features, the research was slowed down by other factors no less critical. The need for better benchmarks represents the first factor. As an example, the abundant availability of conspicuous datasets like ImageNet [9] has driven supervised learning improvement in the research. For what concerns reinforcement learning, the nearest equivalent to supervised learning datasets would be a broad collection of different environments in order to test various algorithms with different kinds of observations or rewards. The second withdraw of this type of approach to learning is the lack of standardisation of environments designed in publications. In reinforcement learning, subtle differences in the definition of the problem or the design of the reward function or the action space could make the difficulty of the task grow. This fact threatens to slow down and corrupts experiments reproducibility making an objective comparison between the results of different papers almost impossible.

The need to fix both problems was the primary motivation behind the design and implementation of OpenAI Gym.

3.1.1 Environments

The agent and the environment represent the core of reinforcement learning. The choice of OpenAI was to implement and provide the abstraction mainly for environments, not for agents, leaving developers independent concerning agent interfaces, but providing a standard interface for environments. Thanks to this approach, all agents implemented with OpenAI Gym can be used with the whole set of environments provided by the framework. Therefore, it is possible to create a personalised environment to suit the needs of a specific experiment that can be used by all agents exploiting OpenAI Gym environment interfaces.

In this scenario, we realised the first contribution to our thesis. Thanks to this framework features, we implemented an OpenAI Gym environment capable of interacting with Anki Cozmo by providing a binding between the function of Cozmo SDK and the interfaces of the reinforcement learning framework. In chapter 4 on page 45 we will provide further information and details about our contribution.

The importance related to the high quantity of environment is fundamental to build a reliable and sustainable framework for reinforcement learning algorithms. OpenAI Gym contains a various and heterogeneous environments database, ready to be used by every developed agent thanks to the standard interface offered by the framework.

Interface Functions

Exploring OpenAI Gym, it is essential to focus on the most crucial interface functions that the agent will exploit to interact with the environment. The functions which constitute the skeleton of an OpenAI Gym environment are the following:

- `def step(self, action)`: through this function, the agent can communicate the action it wants to take. The input data depends on the type and number of variables in the actions space (e.g. discrete or continuous). As will be discussed in section 3.1.2 on the following page, the values returned by this function represent the environment state after the manipulation caused by the action of the agents. Thanks to these data, the agent will be able to select the next action following the reinforcement learning loop.

- `def reset(self)`: during the episode, internal variables of the environment changes, influenced by the action taken by the actor. This function allows the agent to restart the initial situation of the environment. This procedure is particularly suitable when an episode finishes and the agent has to restart the next learning episode in a brand new copy of the environment.

- `def render(self, mode='human', close=False)`: this function is particularly suitable for simulated environments. It enables the visual render (if available) of the environment.

- `def close(self)`: the final function to close the environment after all experiments and episodes.

Available environments

To date, OpenAI Gym includes the following environments:

- **Algorithms**: learning to imitate computations, such as copying or reversing symbols from the input tape, is the main aim of this typology. These environments might seem easy to be solved by a computer, but it is important to remember that the objective here is to learn to solve these tasks purely from examples. Therefore, it is possible to vary the sequence length to increase or decrease task difficulties easily.

- **Atari**: various videogames of Atari 2600, a home video game console developed in 1977 which spread the use of general-purpose CPUs into gaming with game code distributed through cartridges, constitute this environment section. OpenAI Gym exploits *Arcade Learning Environment* (ALE) [3] providing RAW pixel images or RAM as input. This database contains more than 100 different environments.

- **Box2D**: here is possible to find some continuous control tasks in a simple 2D simulator such as *BipedalWalker*, *CarRacing* and *LunarLander*

- **Classic Control**: in this group, it is possible to find the set of problem borrowed by control theory and widely exploited in the classic reinforcement learning literature. Some task examples are balancing a pole on a cart or swing up a pendulum.

- **MuJoCo**: this class contains continuous control tasks running in a fast physics three-dimensional simulator called *MuJoCo* which stands for *Multi-Joint dynamics with Contact*. This physics engine aims to facilitate research and development in robotics, biomechanics, graphics and animation. The simulator is particularly suitable for model-based optimisation allowing to scale up computationally-intensive techniques. Thanks to its features, it became useful as a source for reinforcement learning algorithms.

- **Robotics**: OpenAI released this algorithm typology to provide eight robotics environments with manipulation tasks significantly more difficult than the MoJoCo ones. It contains *Fetch*, a robotic arm to move and grab objects, and *ShadowHand*, a robotic hand to manipulate and grab pens, cubes and balls. [45]

3.1.2 Observations

In the previous section the func

3.2 Anki Cozmo

TODO (Macaluso P.):

- General description of Cozmo and Anki

- General information about the mechanics and features of Cozmo (LINK)

- Discussion about the selection of Cozmo instead of other solutions

  - Amazon Deep Racer: not available at the start of the thesis. It provides a simulator to train the agent. Using AWS for computation which can be a benefit, but also a drawback because it is a lock-in solution.

  - Building a Car: one of the best path to follow because of the personalization available. Main drawbacks are the length of the car construction process but also the time to spend in the creation of interfaces between the car and Python.

  - In the end, Cozmo is the best trade-off between functionalities and fast-developing. It provides plain and straightforward control of the car and a rich Python SDK to use with OpenAI Gym.

- Discussion about the on-board or off-board computation

3.2.1 Features of Cozmo

3.2.2 Cozmo SDK

3.3 PyTorch

TODO (Macaluso P.):

- General description of Pytorch framework

3.3.1 Tensor and Gradients

3.3.2 Building a Convolutional Neural Network

3.3.3 Loss function and Optimizers

3.3.4 TensorboardX

Chapter 4

Tools and Frameworks

TODO (Macaluso P.):

- Introduction to the chapter

4.1 OpenAI Gym

TODO (Macaluso P.):

- General description of the framework
- Discussion about the motivation and the importance of this framework, such as the necessity of a Reinforcement Learning Framework to test different algorithms with different environments providing the same interface.
- One contribution of the thesis: the creation and design of an OpenAI Gym environment for Anki Cozmo Environment with connection to chapter 4.

4.1.1 The importance of a Framework

4.1.2 Main features

4.1.3 How to create an environment

4.2 Anki Cozmo

TODO (Macaluso P.):

- General description of Cozmo and Anki
- General information about the mechanics and features of Cozmo (LINK)

- Discussion about the selection of Cozmo instead of other solutions
  - Amazon Deep Racer: not available at the start of the thesis. It provides a simulator to train the agent. Using AWS for computation which can be a benefit, but also a drawback because it is a lock-in solution.
  - Building a Car: one of the best path to follow because of the personalization available. Main drawbacks are the length of the car construction process but also the time to spend in the creation of interfaces between the car and Python.
  - In the end, Cozmo is the best trade-off between functionalities and fast-developing. It provides plain and straightforward control of the car and a rich Python SDK to use with OpenAI Gym.
- Discussion about the on-board or off-board computation

4.2.1 Features of Cozmo

4.2.2 Cozmo SDK

4.3 PyTorch

TODO (Macaluso P.):

- General description of Pytorch framework

4.3.1 Tensor and Gradients

4.3.2 Building a Convolutional Neural Network

4.3.3 Loss function and Optimizers

4.3.4 TensorboardX

Chapter 5

Design of the Control System

5.1 The main concept

5.1.1 Related Work

5.2 The track

The design and training of a good driving model cannot go beyond the construction and design of the road. For this reason, some time was spent searching the better way to build a path where to train Cozmo. This section aims to present the central concept and decisions made about the design of the track for the experiments, starting from the materials used, up to the description of the dimensional choices applied.

5.2.1 Track requirements

It is essential to explain the primary needs of the road before proceeding with the description of the choices made.

Firstly, the track needs to be easily transportable to allow various attempt with different locations and environmental conditions that could affect the training phase. In particular, it is necessary to use a material less reflective as possible to avoid problems during the learning process. Another crucial factor is the dimension of the lane, which must reproduce an environment similar to the real one. It can be done analyzing the ratio of the size of a vehicle to the width of a road. On average, a family car is about 160-170cm large, while a lane width can vary between 275cm and 375cm. Cozmo width is about 5.5cm, which results in a ratio of 1/30. Therefore, the scaled lane must be between 9cm and 12.5cm.

5.2.2 Track design and materials

The first choice to make is the one about the type of material to use as terrain for the track. The first choice was the black floor of the Data Science laboratory of Eurecom. It was useful only during the initial design and development of the control system to build small pieces of track in which testing functionalities. This solution had numerous drawbacks such as the impracticality to transport and high light reflection.

The following list provides a brief report of the various solutions taken into account during the thesis, together with an analysis of advantages and drawbacks.

- *Covering fabric*: this material is easily transportable, but it has a high light reflection, and its structure is prone to make wrinkles and dunes challenging to remove.
- *Tar paper*: this solution slightly diminished the reflection problem compared to the previous choice, but the material was fragile and with the same drawbacks of the covering fabric.
- *Cotton fabric*: this solution offers an easily transportable material with reduced light reflection where it is easy to remove wrinkles and dunes.

Summing up, it is noticeable from this analysis that the cotton fabric provides the right trade-off among all requirements reported before.

The structure of the road is also composed by the lane. The implementation of this part was done using a simple paper tape of width equal to 2.5cm. As described in the beginning of this section, the width of the lane must be between 9cm and 12.5cm to provide a context similar to the real one. Because of the narrow and limited angle of vision provided by Cozmo camera, 10cm-width was set: positioning the tape with a distance greater than 10cm would result in a great part of the tape outside the view of the camera.

5.2.3 Problems and solutions

5.3 Cozmo Control System

5.3.1 Formalization as an Markov Decision Process

5.3.2 Design of the OpenAI Gym Environment

5.3.3 Main setup of the system

Chapter 6

Experiments

Suspendisse vitae elit. Aliquam arcu neque, ornare in, ullamcorper quis, commodo eu, libero. Fusce sagittis erat at erat tristique mollis. Maecenas sapien libero, molestie et, lobortis in, sodales eget, dui. Morbi ultrices rutrum lorem. Nam elementum ullamcorper leo. Morbi dui. Aliquam sagittis. Nunc placerat. Pellentesque tristique sodales est. Maecenas imperdiet lacinia velit. Cras non urna. Morbi eros pede, suscipit ac, varius vel, egestas non, eros. Praesent malesuada, diam id pretium elementum, eros sem dictum tortor, vel consectetuer odio sem sed wisi.

6.1 Results of the experiments

Suspendisse vitae elit. Aliquam arcu neque, ornare in, ullamcorper quis, commodo eu, libero. Fusce sagittis erat at erat tristique mollis. Maecenas sapien libero, molestie et, lobortis in, sodales eget, dui. Morbi ultrices rutrum lorem. Nam elementum ullamcorper leo. Morbi dui. Aliquam sagittis. Nunc placerat. Pellentesque tristique sodales est. Maecenas imperdiet lacinia velit. Cras non urna. Morbi eros pede, suscipit ac, varius vel, egestas non, eros. Praesent malesuada, diam id pretium elementum, eros sem dictum tortor, vel consectetuer odio sem sed wisi.

6.1.1 Track A

Suspendisse vitae elit. Aliquam arcu neque, ornare in, ullamcorper quis, commodo eu, libero. Fusce sagittis erat at erat tristique mollis. Maecenas sapien libero, molestie et, lobortis in, sodales eget, dui. Morbi ultrices rutrum lorem. Nam elementum ullamcorper leo. Morbi dui. Aliquam sagittis. Nunc placerat. Pellentesque tristique sodales est. Maecenas imperdiet lacinia velit. Cras non urna. Morbi eros pede, suscipit ac, varius vel, egestas non, eros. Praesent malesuada, diam id pretium elementum, eros sem dictum tortor, vel consectetuer odio sem sed wisi.

6.1.2 Track B

Suspendisse vitae elit. Aliquam arcu neque, ornare in, ullamcorper quis, commodo eu, libero. Fusce sagittis erat at erat tristique mollis. Maecenas sapien libero, molestie et, lobortis in, sodales eget, dui. Morbi ultrices rutrum lorem. Nam elementum ullamcorper leo. Morbi dui. Aliquam sagittis. Nunc placerat. Pellentesque tristique sodales est. Maecenas imperdiet lacinia velit. Cras non urna. Morbi eros pede, suscipit ac, varius vel, egestas non, eros. Praesent malesuada, diam id pretium elementum, eros sem dictum tortor, vel consectetuer odio sem sed wisi.

Appendix A

Reinforcement Learning

A.1 Bellman Equation

The value function is decomposable in the immediate reward r_t and the discounted state value of the next state. It is possible to obtain the result in eq. (A.1) by writing expectations explicitly.

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}[g_t|s_t = s] \\ &= \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots |s_t = s] \\ &= \mathbb{E}[r_{t+1} + \gamma g_{t+1}|s_t = s] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P(s', r|s, a)\Big[r + \gamma \mathbb{E}[g_{t+1}|s_{t+1} = s']\Big] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P(s', r|s, a)\Big[r + \gamma V^{\pi}(s')\Big] \end{aligned} \tag{A.1}$$

This equation expresses the relationship between the value of a state and the values of its successor states. It is further possible to derive the Bellman Equation for Action-Value function using the same procedure described above.

The resulting formulas are shown in eq. (2.7) on page 8.

Furthermore, it is possible to obtain the Bellman Equation solution in eq. (A.2) working with matrix notation.

$$\begin{aligned} V^{\pi} &= \mathcal{R}^{\pi} + \gamma \mathcal{P}^{\pi} V^{\pi} \\ (I - \gamma \mathcal{P}^{\pi})V^{\pi} &= \mathcal{R}^{\pi} \\ V^{\pi} &= (I - \gamma \mathcal{P}^{\pi})^{-1}\mathcal{R}^{\pi} \end{aligned} \tag{A.2}$$

Algorithm A.1: Policy Iteration for estimating $\pi \sim \pi^*$

```
Input: $\pi$ the policy to be evaluated; a small threshold $\theta$ which defines the
       accuracy of the estimation
Initialise $V(s)$ $\forall s \in \mathcal{S}$ arbitrarily, except that $V(terminal) = 0$
$is\_policy\_stable \leftarrow true$
repeat
    /* Policy Evaluation                                              */
    repeat
        $\Delta \leftarrow 0$
        for each $s \in \mathcal{S}$ do
            v $\leftarrow V(s)$
            $V(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P(s', r|s, a)\Big[r + \gamma V(s')\Big]$
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
        end
    until $\Delta < \theta$
    $V_\pi \leftarrow V(s)$
    /* Policy Improvement                                             */
    while true do
        for each $s \in \mathcal{S}$ do
            $old\_action \leftarrow \pi(s)$
            $\pi(s) \leftarrow \arg\max_a \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P(s', r|s, a)\Bigg[r + \gamma V_\pi(s')\Bigg]$
            if $old\_action \neq \pi(s)$ then
                $is\_policy\_stable \leftarrow false$
            end
        end
    end
until $\neg$ $is\_policy\_stable$
Output: $V^*$ and $\pi^*$
```

A.2 Dynamic programming

Policy iteration algorithm

Policy improvement theorem

Let π and π' be any pair of deterministic policy such that

$$Q_\pi(s, \pi'(s)) \geq V_\pi(s) \; \forall s \in S \tag{A.3}$$

Then the policy π' leads to

$$V'_\pi(s) \geq V_\pi(s) \tag{A.4}$$

Therefore, the presence of strict inequality in eq. (A.3) for a state leads to a strict inequality of eq. (A.4).

The proof of this theorem is shown in eq. (A.5).

$$\begin{aligned} V_\pi(s) &\leq Q_\pi(s, \pi'(s)) \\ &= \mathbb{E}[r_{t+1} + \gamma V_\pi(s_{t+1}) | s_t = s, a_t = \pi'(s)] \\ &= \mathbb{E}_{\pi'}[r_{t+1} + \gamma V_\pi(s_{t+1}) | s_t = s] \\ &\leq \mathbb{E}_{\pi'}[r_{t+1} + \gamma Q_\pi(s_{t+1}, \pi'(s_{t+1})) | s_t = s] \;\; \text{(by A.3)} \\ &= \mathbb{E}_{\pi'}[r_{t+1} + \gamma \mathbb{E}_{\pi'}[r_{t+2} + \gamma V_\pi(s_{t+2}) | s_{t+1}, a_{t+1} = \pi'(s_{t+1})] | s_t = s] \\ &= \mathbb{E}_{\pi'}[r_{t+1} + \gamma r_{t+2} + \gamma^2 V_\pi(s_{t+2}) | s_t = s] \\ &\leq \mathbb{E}_{\pi'}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 V_\pi(s_{t+3}) | s_t = s] \\ &\vdots \\ &\leq \mathbb{E}_{\pi'}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \ldots | s_t = s] \\ &= v_{\pi'}(s) \end{aligned} \tag{A.5}$$

Value iteration algorithm

Algorithm A.2: Value Iteration, for estimating $\pi \sim \pi^*$

```
Input: A small threshold $\theta$ which defines the accuracy of the estimation
Initialise $V(s)$ $\forall s \in \mathcal{S}$ arbitrarily, except that $V(\mathit{terminal}) = 0$
repeat
    $\Delta \leftarrow 0$
    for each $s \in \mathcal{S}$ do
        v $\leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P(s', r|s, a)\Big[r + \gamma V(s')\Big]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end
until $\Delta < \theta$
Output a deterministic policy, $\pi \sim \pi^*$, such that

    $$\pi(s) = \arg\max_a \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P(s', r|s, a)\Big[r + \gamma V(s')\Big]$$

Output: $V^*$ and $\pi^*$
```

Bibliography

[1] Anki. Github repository of Cozmo SDK written in python, 2019. https://github.com/anki/cozmo-python-sdk.

[2] Badrinarayanan, V., Kendall, A., and Cipolla, R. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE transactions on pattern analysis and machine intelligence 39*, 12 (2017), 2481–2495.

[3] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research 47* (2013), 253–279.

[4] Bellman, R. E., and Dreyfus, S. E. *Applied dynamic programming*, vol. 2050. Princeton university press, 2015.

[5] Bishop, C. M. *Pattern recognition and machine learning.* springer, 2006.

[6] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym, 2016.

[7] Charafeddine, M. Reinforcement learning in the wild and lessons learned, 2018. https://medium.com/@mohamadtweets/reinforcement-learning-in-the-wild-and-lessons-learned-c4b7aa04c3a5.

[8] Deisenroth, M., and Rasmussen, C. E. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)* (2011), pp. 465–472.

[9] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition* (2009), Ieee, pp. 248–255.

[10] Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning* (2016), pp. 1329–1338.

[11] Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research 12*, Jul (2011), 2121–2159.

[12] Erhan, D., Bengio, Y., Courville, A., and Vincent, P. Visualizing higher-layer features of a deep network. *University of Montreal 1341*, 3 (2009), 1.

[13] Fox, R., Pakman, A., and Tishby, N. Taming the noise in reinforcement learning via soft updates. *arXiv preprint arXiv:1512.08562* (2015).

[14] François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., Pineau, J., et al. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning 11*, 3-4 (2018), 219–354.

[15] Fujimoto, S., van Hoof, H., and Meger, D. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477* (2018).

[16] Gu, S., Holly, E., Lillicrap, T., and Levine, S. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)* (2017), IEEE, pp. 3389–3396.

[17] Haarnoja, T., Tang, H., Abbeel, P., and Levine, S. Reinforcement learning with deep energy-based policies. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70* (2017), JMLR. org, pp. 1352–1361.

[18] Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290* (2018).

[19] Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905* (2018).

[20] Hasselt, H. V. Double q-learning. In *Advances in Neural Information Processing Systems 23*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, Eds. Curran Associates, Inc., 2010, pp. 2613–2621.

[21] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence* (2018).

[22] Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence* (2018).

[23] Ioffe, S., and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[24] Kendall, A., Hawke, J., Janz, D., Mazur, P., Reda, D., Allen, J.-M., Lam, V.-D., Bewley, A., and Shah, A. Learning to drive in a day. In *2019 International Conference on Robotics and Automation (ICRA)* (2019), IEEE, pp. 8248–8254.

[25] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[26] Kingma, D. P., and Welling, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[27] Konda, V. R., and Tsitsiklis, J. N. Actor-critic algorithms. In *Advances in neural information processing systems* (2000), pp. 1008–1014.

[28] Kullback, S. Information theory and statistics. Tech. rep., 1959.

[29] Kullback, S., and Leibler, R. A. On information and sufficiency. *The annals of mathematical statistics 22*, 1 (1951), 79–86.

[30] Lapan, M. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more.* Packt Publishing Ltd, 2018.

[31] LeCun, Y., Bengio, Y., et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks 3361*, 10 (1995), 1995.

[32] LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *nature 521*, 7553 (2015), 436.

[33] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[34] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[35] Lin, L.-J. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning 8*, 3-4 (1992), 293–321.

[36] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (2016), pp. 1928–1937.

[37] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[38] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529.

[39] OpenAI. OpenAI gym, 2016. https://github.com/openai/gym.

[40] OpenAI. OpenAI gym documentation, 2016. https://gym.openai.com/docs/.

[41] OpenAI. Openai five, 2018. https://openai.com/blog/openai-five/.

[42] OpenAI. How to train your openai five, 2019. https://openai.com/blog/how-to-train-your-openai-five/.

[43] OpenAI. OpenAI spinning up, 2019. https://spinningup.openai.com/.

[44] Ort, T., Paull, L., and Rus, D. Autonomous vehicle navigation in rural environments without detailed prior maps. In *2018 IEEE International Conference on Robotics and Automation (ICRA)* (2018), IEEE, pp. 2040–2047.

[45] Plappert, M., Andrychowicz, M., Ray, A., McGrew, B., Baker, B., Powell, G., Schneider, J., Tobin, J., Chociej, M., Welinder, P., Kumar, V., and Zaremba, W. Ingredients for robotics research, February 2018. https://openai.com/blog/ingredients-for-robotics-research/.

[46] Rawlik, K., Toussaint, M., and Vijayakumar, S. On stochastic optimal control and reinforcement learning by approximate inference. In *Twenty-Third International Joint Conference on Artificial Intelligence* (2013).

[47] Research, B. A. I. Soft actor critic—deep reinforcement learning with real-world robots, 2018. https://bair.berkeley.edu/blog/2018/12/14/sac/.

[48] Rezende, D. J., Mohamed, S., and Wierstra, D. Stochastic backpropagation and approximate inference in deep generative models. *arXiv preprint arXiv:1401.4082* (2014).

[49] Rumelhart, D. E., Hinton, G. E., Williams, R. J., et al. Learning representations by back-propagating errors. *Cognitive modeling 5*, 3 (1988), 1.

[50] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. Prioritized experience replay. *arXiv preprint arXiv:1511.05952* (2015).

[51] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[52] Shalev-Shwartz, S., and Ben-David, S. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[53] Silver, D. University college london course on reinforcement learning. http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html, 2015.

[54] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *nature 529*, 7587 (2016), 484.

[55] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815* (2017).

[56] Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. Deterministic policy gradient algorithms.

[57] Stanford University. CS231n: Convolutional neural networks for visual recognition, 2019. http://cs231n.github.io/.

[58] Sutton, R. S., and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

[59] Tieleman, T., and Hinton, G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning 4*, 2 (2012), 26–31.

[60] Toussaint, M. Robot trajectory optimization using approximate inference. In *Proceedings of the 26th annual international conference on machine learning* (2009), ACM, pp. 1049–1056.

[61] Uhlenbeck, G. E., and Ornstein, L. S. On the theory of the brownian motion. *Physical review 36*, 5 (1930), 823.

[62] Van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence* (2016).

[63] Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., and De Freitas, N. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581* (2015).

[64] Watkins, C. J. C. H. Learning from delayed rewards.

[65] WAYVE. Learning to drive like a human, 2019. https://wayve.ai/blog/driving-like-human.

[66] Ziebart, B. D., Maas, A., Bagnell, J. A., and Dey, A. K. Maximum entropy inverse reinforcement learning.