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 TODO (Macaluso P.): 2019-2020

Abstract

TODO (Macaluso P.): Abstract is the last thing to do

Acknowledgements

TODO (Macaluso P.): Acknowledgements must be prepared!

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 [20].

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 [25] or chess [26], to be particularly suited for simulated environments like computer games [17], and to be a promising methodology for simple tasks with robotic manipulators [7].

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). This event opened the doors to function approximation with Neural Network (NN) and Convolutional Neural Network (CNN) 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.

Recently RL has known a remarkable development and interests in video games: it managed to beat world champions at the game of Go [25] and Dota with superhuman results and to master numerous Atari video games [17] from raw pixels. Decisions, actions and consequences make video games a simulated reality on which to exploit and test the power of RL algorithms. It is essential to realise that the heart of RL is the science of decision making. This fact makes it compelling and general for many research fields ranging from Engineering, Computer Science, Mathematics, Economics, to Psychology and Neuroscience.

Before discussing the results of this thesis, it is good to clarify everything that today represents the state-of-the-art in order to understand the universe behind this new 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 RL, then in the second section it moves progressively towards what is Deep RL through a careful discussion of the most essential algorithms paying more attention to those used during the thesis project.

The elaboration of this chapter is inspired by [24], [27], [19], [12] and [6].

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

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 *environment* represents all the things that are outside the agent. At every action received by the agent, it emits a reward, an essential aspect of RL, 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 supervised and unsupervised learning. 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 indipendent and identically distributed (i.i.d.).

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.

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 sequence of states and actions is named *trajectory* (τ): it is helpful to represent an episode in Reinforcement Learning (RL) framework.

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

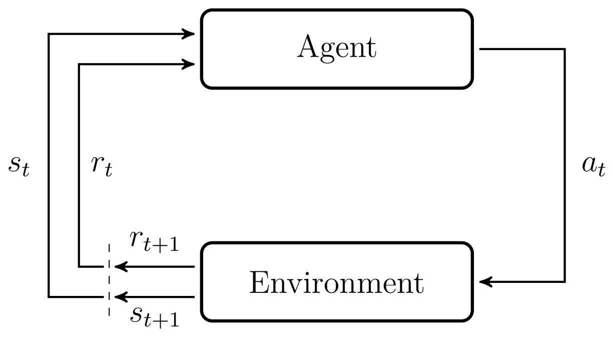

Figure 2.1. Interaction loop between Agent and Environment. The reward and the state resulting from taking an action become the input of the next iteration.

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.

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.

An 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.

In this quick overview of the central unit 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 core of RL because it 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* [2] – shown in eq. (2.7) and demonstrated in appendix A.1 on page 36 – 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.

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 previously.

One of the most crucial aspects of an RL algorithm is the question of whether the agent has access to or learns a model of the environment. A model of the environment 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, because of its on-policy nature, 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 value. 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.

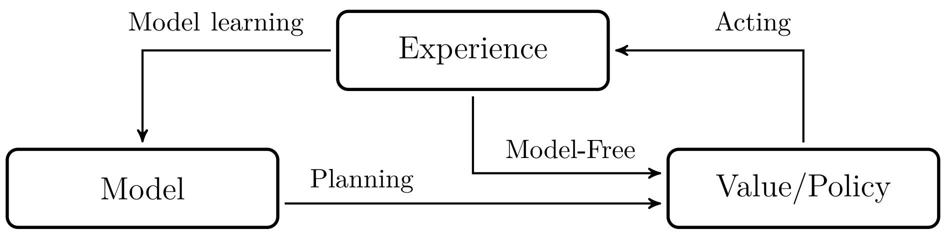

Figure 2.2. Overview of components of an agent with their relation in respect of different approaches of learning. Model-Free methods works with the experience, value functions and the policy, while model-based techniques tries to build up a model to derive value functions and policy to act in the environment.

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. Generalization becomes an important problem in the last approach because the agent is not able to interact anymore with the environment. In the context of this thesis, what matters is *online learning*: in this context, the learning phase is not bound to already 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 important 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) [27, Chapter 4] is one of the approaches used to solve RL problems calculation the optimal policy π^*. Formally, it is a general method to solve complex problems by breaking them into sub-problems that are more convenient to solve. 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 37 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 represents 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 take 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 37 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 38.

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 following 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, 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 the

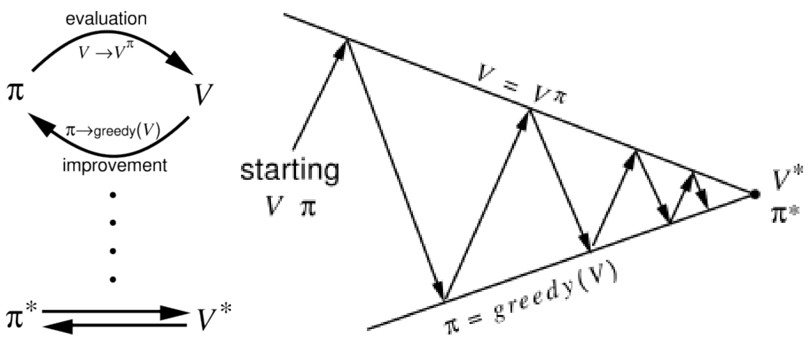

Figure 2.3. Generalized policy iteration schema [27]. Value and policy functions compete and cooperate to reach the joint solution: the optimal value function and an optimal policy.

model 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 [27, 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 summarise its pseudocode.

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

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

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

Q-learning [32] 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 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 2.2: Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

```
   Input: step size $\alpha \in (0,1]$, small $\epsilon > 0$
1  Initialise $Q(s,a)\ \forall\ s \in \mathcal{S}, a \in \mathcal{A}$ arbitrarily, except that $Q(\text{terminal}, \cdot) = 0$
2  foreach episode do
3      Initialise $s_t$
4      Choose $a_t$ from $s_t$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
5      repeat
6          Take action $a_t \rightarrow$ obtain $r_{t+1}$ and $s_{t+1}$
7          Choose $a_{t+1}$ from $s_{t+1}$ using policy derived from Q (e.g. $\epsilon$-greedy)
8          $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$
9          $s_t \leftarrow s_{t+1}$ ; $a_t \leftarrow a_{t+1}$
10     until $s_t$ is terminal
11 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) [27, 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

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

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 $\theta = (\theta_1, \theta_2 . . . , \theta_n)^T$ 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 (DRL). This evolution of traditional reinforcement learning 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 [17] and the consequent [18].

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 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 $\theta \in \mathbb{R}^{n_\theta}(n_\theta \in \mathbb{N})$ such that $y = f(x;\theta)$.

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 $f(\sum_n \theta_i)$ following relation where f is an activation function. The parallel comparison between the biological neuron and the artificial one is shown in fig. 2.4 on the following page. The set of parameter w needs to be adjusted to find a good parameter set: this process is called *learning*.

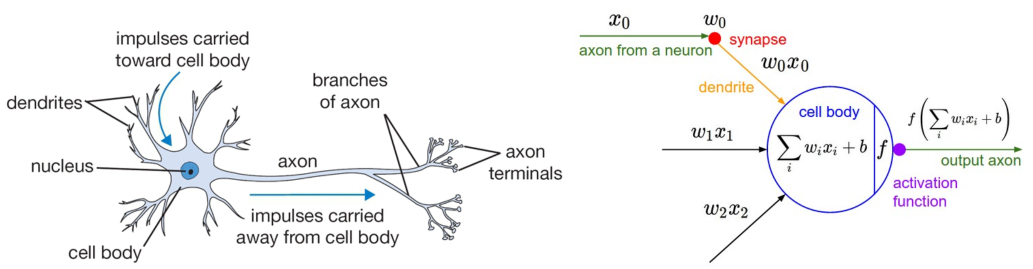

Figure 2.4. Comparison between biological neuron (left) and artificial neuron (right). The artificial neuron designs the dendrites as weighted inputs and returns the sum through an activation function. [29].

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 [5]. 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.

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

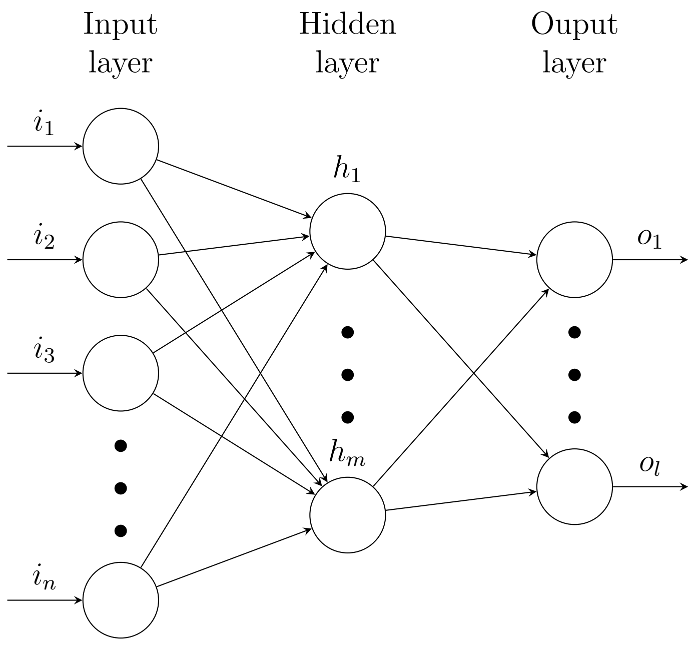

Figure 2.5. An example representation of a deep neural network with one fully-connected hidden layer.

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) [23] 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 [21] 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* [14], 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 calculate 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 greater step towards the minimum, which threatens to overshoot the target.

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

Nowadays, the techniques applied in the majority of research projects is stochastic gradient descent which combines batch learning [29] and gradient descent, but also its various improved extensions and variants, such as ADAM [10] and AdaGrad [4]: 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 [3, 14] 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

TODO (Macaluso P.): RESTART FROM HERE

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

Convolutional Neural Networks

Another crucial step in deep learning was the introduction of convolutional layers [13]. This type of layer revealed is power with images, leading to an increasing interest in image processing field. The parameters of this layer are a set of filters (or kernels) with dimensions smaller than the whole input image: these filters are used to implement a convolution operation over the whole image, and the result serves the input to the next layer in the network. In this context, the network can learn filters that detect specific features in the image such as edges, textures and patterns [5].

Convolutional Layer

Pooling Layer

2.2.2 Value-based methods

The first class of algorithms to explore is the one of value-based methods. 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 [32] shown previously in section 2.1.5 on page 13. The team of DeepMind introduced the Deep Q-Network (DQN) in [17] and in the next cutting-edge paper [18]: 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 [28]. 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, called *target network*, enriches the update phase of this algorithm. Equation (2.23) 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.

$$\begin{aligned} L_i(\theta_i)) &= \mathbb{E}_{s,a\sim\pi}\Big[(y_i - Q(s, a; \theta_i))^2\Big] \\ y_i &= \mathbb{E}_{s'\sim E}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})|s, a] \end{aligned} \tag{2.23}$$

Experience Memory Replay Another crucial introduction in this algorithm is *experience memory replay buffer* [15]. Trajectories sampled from the environment are temporally correlated, and this could lead to over-fitting the parameters of the neural network. 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 over-fitting 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* [9]: 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 [8, 30] can handle the intricacy of overestimation of Q-values caused by the maximisation step in eq. (2.23) 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. (2.24)

$$y_i = \mathbb{E}_{s' \sim E}[r + \gamma Q(s', \operatorname*{argmax}_a Q(s', a; \theta_{i-1}); \theta^-_{i-1})|s, a] \tag{2.24}$$

Prioritized Experience Replay The fundamental idea underlying *prioritized experience replay* [22] is exactly 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 [31] shown in fig. 2.6 on the following page works decoupling the Q-value estimation in two distinct sequences of fully-connected layers right after convolutional layers. 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. (2.25), but [31] also suggests a different approach eq. (2.26) which shown increased stability in practice.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \Big(A(s,a;\theta,\alpha) - \max_{a'\in\mathcal{A}} A(s,a';\theta,\alpha)\Big) \tag{2.25}$$

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \Big(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'\in\mathcal{A}} A(s,a';\theta,\alpha)\Big) \tag{2.26}$$

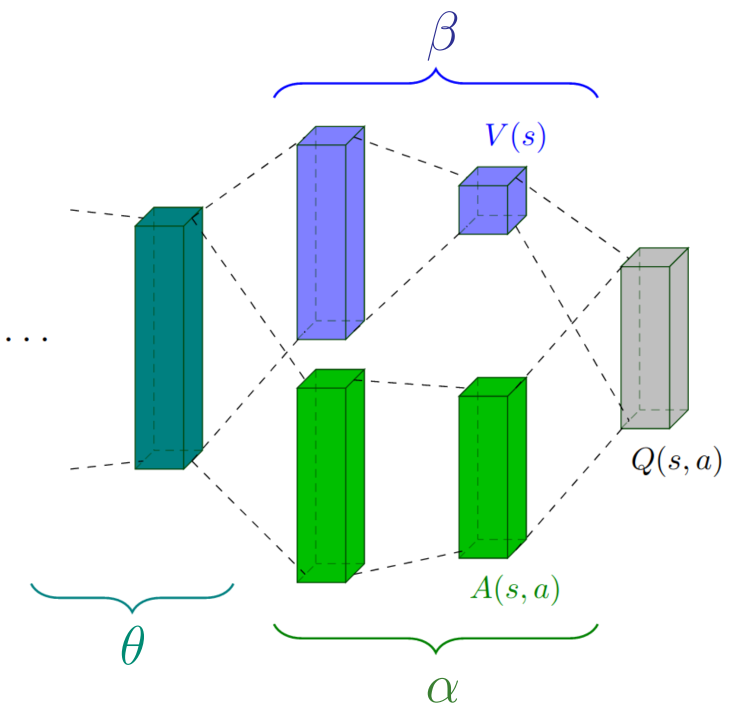

Figure 2.6. Dueling DQN architecture [31] representation taken from [6]. It consists in two stream to estimate state-value parametrized with β and advantages values parametrized with α for each action. The last layer represent the combination of these two types of values.

2.2.3 Policy gradient methods

Policy gradient algorithms aim to optimize the policy's performance measure in eq. (2.27) by finding a good policy $\pi_\theta(s|a)$ capable of generation a trajectory τ that maximizes the expected rewards eq. (2.28) instead of learning a value function. Indeed the objective function in policy gradient methods consists of maximising $J(\theta)$ value by finding a good policy updating θ parameters directly.

$$J(\theta) = \mathbb{E}\Big[\sum_{t=0}^{N} r(s_t, a_t); \pi_\theta\Big] = \sum_{\tau} P(\tau;\theta) r(\tau) \tag{2.27}$$

$$\theta^* = \operatorname*{argmax}_{\theta} J(\theta) \tag{2.28}$$

The parameters of the policy θ are refreshed via stochastic gradient ascent. 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. (2.29) where α is the learning rate which defines the strength of the steps in the direction of the gradient.

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_\theta J(\theta_t) \tag{2.29}$$

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 big 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 important disadvantage: they rather converge to a local maximum than to the global optimum.

Actor-Critic Architecture

Actor-critic architecture, shown in fig. 2.7 on the following page 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* [11]. 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 [16]: 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 13. 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.

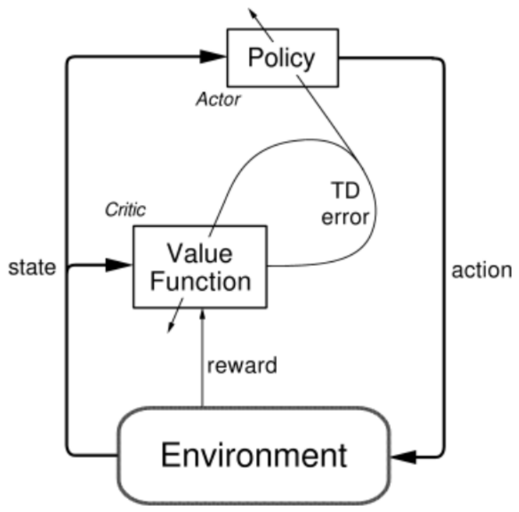

Figure 2.7. Actor-critic architecture schema: the actor represents the policy and maps the input state to an output action, while the critic represents the value function. Both networks can be updated with, e.g. the TD-error, using the contribution of the critic. It is noticeable that the actor uses the critic during the learning process [27].

2.2.4 Deep Deterministic Policy Gradient (DDPG)

2.2.5 Soft-Actor Critic (SAC)

2.3 Related Work

TODO (Macaluso P.):

- Explanation of the state-of-the-art focusing more on Reda's paper, its approach and the related bibliography
- Increasing interest in Reinforcement Learning applied to real-world situations, in contrast with simulated environments experiments

2.4 Summary

Chapter 3

Tools and Frameworks

TODO (Macaluso P.):

- Introduction to the chapter

3.1 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.

3.1.1 The importance of a Framework

3.1.2 Main features

3.1.3 How to create an environment

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

TODO (Macaluso P.): Check correctness and completeness

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.

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

A.2 Dynamic programming

Policy iteration algorithm

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
1  Initialise $V(s)\ \forall s \in \mathcal{S}$ arbitrarily, except that $V(terminal) = 0$
2  $is\_policy\_stable \leftarrow true$
3  repeat
       /* Policy Evaluation                                   */
4      repeat
5          $\Delta \leftarrow 0$
6          for each $s \in \mathcal{S}$ do
7              v $\leftarrow V(s)$
8              $V(s) \leftarrow \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s'\in\mathcal{S}, r\in\mathcal{R}} P(s', r|s, a)\Big[r + \gamma V(s')\Big]$
9              $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
10         end
11     until $\Delta < \theta$
12     $V_\pi \leftarrow V(s)$
       /* Policy Improvement                                  */
13     while true do
14         for each $s \in \mathcal{S}$ do
15             $old\_action \leftarrow \pi(s)$
16             $\pi(s) \leftarrow \arg\max_a \sum_{s'\in\mathcal{S}, r\in\mathcal{R}} P(s', r|s, a)\Big[r + \gamma V_\pi(s')\Big]$
17             if $old\_action \neq \pi(s)$ then
18                 $is\_policy\_stable \leftarrow false$
19             end
20         end
21     end
22 until $\neg\ is\_policy\_stable$
Output: $V^*$ and $\pi^*$
```

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) on the previous page for a state leads to a strict inequality of eq. (A.4).

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

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

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
1  Initialise $V(s)\ \forall s \in \mathcal{S}$ arbitrarily, except that $V(terminal) = 0$
2  repeat
3      $\Delta \leftarrow 0$
4      for each $s \in \mathcal{S}$ do
5          v $\leftarrow V(s)$
6          $V(s) \leftarrow \max_a \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P(s', r | s, a)\big[r + \gamma V(s')\big]$
7          $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
8      end
9  until $\Delta < \theta$
10 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 π^*

Acronyms

CNN Convolutional Neural Network. 4

DL Deep Learning. 4

i.i.d. indipendent and identically distributed. 6

MDP Markov decision process. 1, 7, 9

NN Neural Network. 4

RL Reinforcement Learning. 1, 2, 4, 5, 7, 8

Bibliography

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

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

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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).

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

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

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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).

[18] 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.

[19] OpenAI. Open AI spinning up, 2019. https://spinningup.openai.com/.

[20] 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.

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

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

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

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

[25] 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.

[26] 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).

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

[28] 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.

[29] University, S. Cs231n: Convolutional neural networks for visual recognition, 2019. http://cs231n.github.io/.

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

[31] 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).

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