Faculty of Science and Bio-Engineering Sciences
↪ Department of Computer Science
↪ Artificial Intelligence Laboratory

Model-Free Reinforcement Learning for Real-World Robots

Dissertation submitted in fulfillment of the requirements for the degree of Doctor of Science: Computer Science

Supplementary material available at http://steckdenis.be/thesis_suppl.zip

Denis Steckelmacher

Promotor: Prof. Dr. Ann Nowé (Vrije Universiteit Brussel)
Supervisors: Dr. Peter Vrancx (PROWLER.io)
Dr. Diederik M. Roijers (University of Applied Sciences Utrecht)

Print: Silhouet, Maldegem

2020 Uitgeverij VUBPRESS Brussels University Press
VUBPRESS is an imprint of ASP nv (Academic and Scientific Publishers nv)
Keizerslaan 34
B-1000 Brussels
Tel. +32 (0)2 289 26 50
Fax +32 (0)2 289 26 59
E-mail: info@vubpress.be
www.vubpress.be

ISBN -
NUR -
Legal deposit -

Abstract

Reinforcement Learning allows an artificial agent to learn how to perform a task, given only sensory inputs, and rewards to be maximized. Reinforcement Learning has been successfully applied to several industrial settings, such as energy production and management, transportation, networks, banking, and health care. However, a challenge of Reinforcement Learning, that prevents its wide-spread deployment in many real-world settings, is the need for the agent to learn. When no simulator for a task is available, the agent must learn on the physical system, or the user-facing software. In this case, any mistake made by the agent while learning, due to its still incomplete knowledge or the task, or its inherent need to explore, may have costly effects.

In this thesis, we consider real-world tasks that may benefit from Reinforcement Learning, and for which there is no simulator available. We focus on sample-efficiency, that measures how little interactions with the environment the agent needs before learning the task. We believe that with high sample-efficiency (fast learning), few errors are made, leading to an acceptable cost of training the agent.

We follow a research line that starts with theoretical insights and algorithms, and aims towards practical applications on physical platforms. We review several families of Reinforcement Learning algorithms, then introduce our first contribution, the model-free actor-critic Bootstrapped Dual Policy Iteration algorithm (BDPI). Thanks to its use of off-policy critics, and an explicit actor, BDPI achieves high-quality exploration and sample-efficiency. Our second theoretical contribution is a formalism, based on the Options framework, that allows an agent to learn partially-observable tasks (with a discrete memory) in a sample-efficient way. Fi-

nally, we focus on a real-world robot, a motorized wheelchair controlled with a joystick, and propose extensions to BDPI to make it able to (safely) learn a complex navigation task, directly on the wheelchair, without a model or pre-training, in about one hour of wall-clock time.

We carefully evaluate our algorithms on numerous simulated tasks, and on two robotic platforms, one of them the wheelchair mentioned above. In all our experiments, our algorithms outperform state-of-the-art approaches. Our results demonstrate the applicability of Reinforcement Learning to real-world settings, where sample-efficiency is critical. In addition to our scientific results, we present numerous implementation details, and general introductions to algorithms. We hope that our original algorithms, and the way we present this thesis, will allow a wide range of organizations and companies to deploy Reinforcement Learning in challenging settings, leading to increased value.

Acknowledgments

I would first like to thank **my promotor and mentors**, people who not only helped me doing my research, but also taught me everything I know, and enabled me to pursue my ends. Prof. Ann Nowé, in addition to her extended knowledge of Reinforcement Learning and surrounding domains, also masterfully manages her lab and makes sure that everyone has everything they need. She makes sure that everything is possible, be it material (we received a motorized wheelchair) or immaterial (I was able to give lectures at summer schools, or in companies, even before getting my PhD). Dr. Peter Vrancx was my supervisor before my master thesis, during it, and for the first year of my PhD. He got me interested in Reinforcement Learning, answered my many questions, and pointed me in the right direction on several occasions. Dr. Diederik M. Roijers supervised me at the later stages of my research, taught me how to write and review scientific papers, and discussed many aspects of the algorithms presented in this thesis. His experience abroad was also highly valuable to teach me how the scientific community works on the larger scale.

I would like to thank my jury, Prof. Manuela Veloso, Prof. Robert Babuska, Prof. Bram Vanderborght, Prof. Elisa Gonzalez Boix, Prof. Bernard Manderick and Prof. Ann Nowé, in decreasing order of distance from the lab, for the constructive comments that they provided on this thesis. Their broad range of expertise allowed me to make this manuscript clearer and more complete.

Research is a team effort, and past or present **colleagues** are of utmost importance. Together with Dr. Peter Vrancx, Dr. Anna Harutyunyan greatly helped me during my first year. Her advice allowed me to receive my PhD scholarship

from the Foundations for Scientific Research of Flanders (FWO) on the first try. She formalized my rough ideas to what is now the OOIs framework presented in Chapter 4. She has always been the go-to person for extremely mathematical and formal stuff, that only her can explain clearly enough for me to understand. Other colleagues whose brainstorming sessions were particularly useful, for my research or my sanity, are Timo, Roxana, Pieter, Felipe, Youri, Mathieu, Arno and Kirk (actually Kyriakos). Kirk, especially, is the second colleague I share an understanding the best with.

This leads to the **most important colleague** who I want to thank, and who will appear in this thesis numerous times. Usually, people thank their partner last, for "emotional support". I would like to thank Hélène Plisnier, my dearest, not only for making my life worth living, but for her instrumental, absolutely crucial contributions to my work. She answered numerous fundamental questions that happened when developing Bootstrapped Dual Policy Iteration (Chapter 3). She developed many environments on which algorithms presented in this thesis are evaluated. She is the main reason Invacare Belgium donated a wheelchair to our lab. We had numerous meetings on scientific questions, in the morning, over breakfast. We built hardware together, she helped me reverse-engineer the wheelchair we received, she wrote parts of its controller, and a few software interfaces to joysticks and Bluetooth beacons. Even more importantly, her own line of research provided a formalism and several algorithms that we apply in Chapter 5.

I would now like to thank a few **external organizations**. First, the Foundations for Scientific Research of Flanders (FWO), for having funded my research (grant number 1129319N). Nothing would have happened without them. Then, Invacare Belgium, for having given a motorized wheelchair to the lab.

I thank my non-work friends, not only for making me laugh, or entertaining me with several weddings, but also for sharing their professional experiences with me, which allows me to write this thesis with company life in mind, even though I've only known academic life personally. Finally, let's not forget my family, for having provided me a happy, trouble-free and safe life, on which everything can be built.

How to Read this Thesis

This thesis aims at addressing two audiences at the same time: academics, looking for state-of-the-art research and evaluation, and developers or software architects who want to learn how to apply Reinforcement Learning to their real-world problems. Instead of writing a thesis and a book as two separate entities, with much overlap, we write a single thesis-book that allows each reader to see the two sides of our contributions. We present algorithms and their applications, formalisms and their motivation. We believe that academic-style formalisms and citations are useful and interesting to engineers, and that implementation details and code are also relevant to the scientific community.

As such, this thesis can be read from start to finish by both audiences. We present a single unified story, go from a broad introduction to Reinforcement Learning to our complete contributions. We intertwine intuition and formal details, and discuss practical aspects of how to implement our algorithms in real-world systems. Because a diverse set of readers may have a diverse set of interests, we color-code select sections, hoping to make navigating this thesis-book easier.

Theoretical Sections

Theoretical sections focus on the academic side of our contributions, and use the conventional notations in the field of Reinforcement Learning. After providing an intuition, we use a theoretical section to precisely define the exact setting we are considering, and introduce our notations. We also use theoretical sections to

present proofs and corollaries, or to discuss fundamental aspects of algorithms, environments or results.

Applied Sections

Applied Sections bridge the gap between an algorithm descriptions and actual implementations. In applied sections, we discuss programming languages, frameworks and electronics. By reading applied sections, the reader gains the ability to implement our work, and fully understand the implementation we provide in appendix.

Contributions

Contribution sections highlight our main contributions, and separate them from background information, discussions, corollaries or evaluation.

Most of the sections are not color-coded, thus are not specific to any audience, as we aim at producing a thesis that is as interesting as possible to every reader.

Table of Contents

indicates a contribution section

1 | Introduction

Many systems deployed in the industry, and present in our daily lives, perform actions autonomously. Examples range from the industrial robots that manufacture cars, devices that control how electricity is produced and consumed on the electrical grid [Mihaylov et al., 2016], the management of datacenter cooling solutions [Lazic et al., 2018], to computer-aided decision support for mitigating pandemics [Libin, 2020], thermostats that control heating in increasingly smart ways [De Bock et al., 2017], and robotized devices such as wheelchairs that assist people every day [Feng et al., 2018].

Most of the automated systems currently in use in production are designed by people. An alternative approach, enjoying theoretical research for decades, is Reinforcement Learning. Reinforcement Learning is a family of Machine Learning algorithms that allows an intelligent agent to *learn* how to control a system (a policy), instead of being *programmed* to control the system. The agent observes sensory inputs (states) and produces control signals (actions). After every action, the agent is given a new state and a scalar reward. The objective of the agent is to learn which action to execute in which state to maximize the discounted sum of rewards it receives over episodes (or *trajectories*).

Reinforcement Learning achieved several successes on real-world systems, and is for instance able to learn a policy able to outperform a team of experts at managing the cooling of a datacenter [Lazic et al., 2018]. However, an important challenge of Reinforcement Learning, that reduces the scopes of real-world problems it can be applied to, is its need to learn. When freshly started, a Reinforcement Learn-

ing agent performs mostly random actions, leading to very low-quality control. In real-world systems for which simulators (or models) are challenging or impossible to design, low-quality control may lead to physical damage, or other costly consequences. While most industrial robots can be modeled, we believe that *home robots*, such as motorized wheelchairs and beds, vacuum cleaners, and the future robots we hope this thesis will make possible, cannot be modeled. We motivate our belief in two ways: home robots perform their tasks around people, whose reactions we cannot model; and the robots themselves have to be cheap and sold in large quantities, to a large amount of consumers. Selling to many consumers makes consumer-specific configuration and modeling impractical.

Aiming at increasing the range of human-centric tasks for which robotic assistance is possible, this thesis primarily focuses on increasing the sample-efficiency of model-free Reinforcement Learning agents, leading to faster learning and less low-quality actions being executed. Our hope is that, with increased sample-efficiency and less bad actions, the cost of training a Reinforcement Learning agent in a real-world setting becomes acceptable. For robots being sold to many different consumers, we also hope that training *at home* (or fine-tuning a pre-learned policy), by the consumer, will become possible. In Chapter 5, in which we evaluate our algorithmic contributions on a real-world motorized wheelchair, we also consider the use of backup policies, and their impact on the learning agent, to prevent the agent from executing catastrophic actions. The main result that we present in this thesis, the fact that an off-the-shelf motorized wheelchair is able to learn to navigate in a room in about one hour, without any model, simulator or pre-training, and without ever hitting obstacles, demonstrates that Reinforcement Learning can be successfully applied to real-world tasks where sample-efficiency is critical.

Our main objective of making Reinforcement Learning applicable to real-world robots can be divided in a set of research questions, that we address in this thesis. Each question listed below is accompanied by a brief review of related work, and a reference to the chapter or sections of this thesis that present our answer to the question.

1.1 Main Research Questions

The two main research questions we focus on are related to the sample-efficiency of Reinforcement Learning agents. Each of these research questions, namely sample-efficiency in markov decision processes, and sample-efficiency specific to partially-observable environments, are addressed by two dedicated chapters (3 and 4, re-

spectively). We also present secondary research questions in the next section, that are relevant to the deployment of Reinforcement Learning in the real world, but less fundamental.

1.1.1 Sample-Efficiency in Partially-Observable MDPs

The main Reinforcement Learning formalism considers a Markov Decision Problem (MDP), in which the agent observes the state of the environment before choosing an action. In an MDP, the optimal policy, leading to the highest sum of rewards over episodes, can be expressed as a simple function of the state [Bellman, 1957]. In Partially-Observable MDPs (POMDPs), the agent does not observe the state of the environment anymore, but an observation. The observation may be less informative than the state, with several states leading to a single observation. There is no general way of producing and representing an optimal policy in a POMDP. Current approaches either use the observation to guess what the state of the environment could be [Cassandra et al., 1994; Kaelbling et al., 1998], but are difficult to apply to continuous-valued states [Dallaire et al., 2009]; allow the agent to store bits in a scratch-pad or memory [Peshkin et al., 1999; Graves et al., 2016], but require the agent to spend time learning how to use that memory efficiently; model and reason about sequences of futures observations the agent may see [Littman et al., 2001], a mathematically elegant but computationally challenging approach; or, the most common approach, use recurrent neural networks to map sequences of observations to actions [Bakker, 2001].

The use of recurrent neural networks in Partially-Observable MDPs, detailed in Section 4.1.3, leads to highly encouraging and general results in the recent literature [Mnih et al., 2016, for example]. However, training recurrent neural networks is challenging, and leads to poor sample-efficiency of the resulting agent. We demonstrate that poor sample-efficiency, and propose an alternative approach to addressing partial observability, in Chapter 4.3. Our method builds on the Options framework [Sutton et al., 1999], and allows to efficiently learn in Partialy-Observable MDPs in which discrete pieces of information are unobserved.

1.1.2 Sample-Efficiency in MDPs in General

Increasing sample-efficiency in POMDPs, albeit useful in challenging partially-observable environments, does not increase the range of applications of Reinforcement Learning in fully-observable MDPs. Our second research objective focuses on increasing the sample-efficiency of general model-free Reinforcement Learning,

with as few hypotheses as possible, so that the resulting algorithm can be painlessly applied to as many simulated or physical settings as possible.

Current research in fundamental Reinforcement Learning algorithms focuses on being able to learn high-scoring policies on highly-challenging tasks, such as with pixel-based observations [Hessel et al., 2017], sparse rewards [Ecoffet et al., 2019], or precise dextrous control [Akkaya et al., 2019]. Sample-efficiency is generally considered to be a lower objective. In Chapter 3, we design a highly sample-efficient model-free Reinforcement Learning algorithm, that outperforms various state-of-the-art approaches on many tasks. In Section 5.7.4 later in this thesis, we furthermore show that the sample-efficiency of BDPI does not come at the cost of final policy quality. Finally, BDPI is the algorithm that allows the motorized wheelchair presented in Chapter 5 to learn a navigation task in only one hour, directly on the robot, without a simulator.

The main reason why BDPI is so sample-efficient is its novel combination of *off-policy* critics with a novel actor. Current actor-critic algorithms, even if called "off-policy" (robust to an agent that executes actions that are not chosen by its actor), learn an on-policy critic, tied to the actor. On-policy critics require more care, and are generally less sample-efficient, than the off-policy critics that BDPI uses. We discuss this aspect in Sections 2.8.1 (for the terminology) and 3.2 (for theoretical details).

1.2 Secondary Research Questions

In addition to our primary research questions, focusing on sample-efficiency, the deployment of our algorithms on a motorized wheelchair, a real-world robot, led to many challenges and research questions, that we now describe.

1.2.1 Asynchronous Environments

The Markov Decision Process considers a theoretical setting in which the environment instantly executes an action produced by the agent, and the agent instantly chooses an action to execute given a state. In software implementations of MDPs, the environment is frozen as long as the agent computes which action to execute. Real-world settings, however, cannot be frozen. While the agent processes data, to choose an action or learn from past experiences, the environment continues to change. For instance, motors continue turning, inertia maintains movement, and people around the agent continue with their daily lives. We refer to these

un-freezable environments as *asynchronous*. In Section 5.6, we detail how asynchronous environments may prevent a Reinforcement Learning algorithm from efficiently learning, especially BDPI, that is sample-efficient but compute-intensive. We then propose an extension of BDPI that leverages modern multi-cores machines (for faster compute) and executes its actor asynchronously from its training loops (for robustness to asynchronous environments). Finally, we demonstrate that Asynchronous Parallel BDPI is as sample-efficient as vanilla BDPI, outperforms state-of-the-art Reinforcement Learning algorithms in sample-efficiency on a Virtual Office environment, and allows a motorized wheelchair to learn to navigate in a room.

1.2.2 Safety with Backup Policies

A key challenge in Reinforcement Learning is that the agent may execute bad actions at any time, due to its need to explore, or because of a simple lack of knowledge (in the early stages of learning). In simulated environments, this is not a problem. The agent is punished for its bad actions, and learns not to select them. When learning happens on a real-world platform, any bad action may have costly real-world consequences. In Chapter 5.4.1, we review how backup policies, that detect unsafe states and actions and forcibly move the agent away from them, may influence learning. We propose a simple method of designing a backup policy, that we implement on our motorized wheelchair. Because safety with backup policies is a secondary research question in this thesis, and state-of-the-art methods empirically perform well in our experiments, we do not claim any contribution in this field.

1.2.3 Monocular Obstacle Detection

The particular motorized wheelchair we use in Chapter 5 is available in the market, but comes with no sensor whatsoever. In the pursuit of *cheap robots, smart algorithms*, we refrain from adding expensive but informative sensors on the wheelchair, and instead use a single webcam, attached to the front of the wheelchair, to allow the agent to observe its surrounding.

Detecting the presence of obstacles from a stream of images obtained from a single camera is called *monocular obstacle detection*, and is still an open problem. Existing methods can be divided in methods that look at the color of pixels [Lenser and Veloso, 2003; Ulrich and Nourbakhsh, 2000], recognize obstacles by the way they move in relation to the camera and the floor [Yamaguchi et al., 2006], or

compare the movement of pixels to odometry data [Lee et al., 2016]. This last method can unfortunately not be used on a motorized wheelchair that is unable to measure the rotational speed of its wheels, and modifying every wheelchair that has to be *learning-enabled* before shipping them to potential customers would be unpractical. We therefore propose in Section 5.3.3 an obstacle detection algorithm based on the textures identified in images (movement is ignored). It allows our agent to observe its surroundings in a task-agnostic way (where the agent has to go, and how it is rewarded, does not require changes in how obstacles are seen), and leads to good results in our experiment. We however point out that our main contributions, that focus on sample-efficiency, do not rely on any method for providing observations to a learning agent. Any obstacle detection algorithm can then be used instead of ours, leading to similar (or better) results.

1.2.4 Controlling a Robot

Finally, in Section 5.2.4, we detail how we allow a Reinforcement Learning agent, software running on a computer, to physically control a motorized wheelchair not made to be computer-controlled. This section has only minor theoretical value, but details all the hardware-level engineering and system design we had to perform in order to build our motorized wheelchair experiment.

1.3 Contributions and Plan

This thesis is organized so that our two main contributions, a sample-efficient algorithm for MDPs and a framework based on Options for sample-efficiency in POMDPs, are covered by dedicated chapters. The thesis is organized as follows:

- Chapter 2 introduces Reinforcement Learning and several families of Reinforcement Learning algorithms. It presents background information useful in all the other chapters of this thesis.
- Chapter 3 presents our first contribution, Bootstrapped Dual Policy Iteration, a sample-efficient model-free Reinforcement Learning algorithm. This chapter also positions BDPI regarding related work, and provides an empirical evaluation of BDPI in several challenging environments.
  Paper: Steckelmacher et al. (2020) *Sample-efficient model-free Reinforcement Learning with off-policy critics*, European Conference on Machine Learning, proceedings in Springer Lecture Notes in Computer Science.

- Chapter 4 presents our second contribution, Option-Observation Initiation Sets, a formalism built on Options that allows an agent to reason about a discrete memory in a sample-efficient way. Related work, an empirical evaluation on simulated task, and an empirical evaluation on a robotic platform are also presented in this chapter.
  Paper: Steckelmacher et al. (2017) *Reinforcement Learning in POMDPs with memoryless options and Option-Observation Initiation Sets*, AAAI Conference on Artificial Intelligence, proceedings by AAAI Press.

- Chapter 5 extends BDPI with parallel processing, and applies it to an off-the-shelf motorized wheelchair. This chapter discusses the practical aspects of deploying a Reinforcement Learning algorithm on a physical platform, and details the implementation work that was necessary to port a learning agent to the motorized wheelchair we received from Invacare Belgium.
  Paper: Plisnier & Steckelmacher et al. (equal contribution), *Indoor Navigation on a Wheelchair with Model-Free Reinforcement Learning*, submitted to Machine Learning.

- Finally, in Chapter 6, we summarize our contributions and discuss their applicability to problems more diverse than the motorized wheelchair setting we present in Chapter 5.

The main result of this thesis is that a Reinforcement Learning agent can safely learn a navigation task, directly on a real-world robot, in only about one hour. The agent receives low-level observations and is not helped with domain knowledge. Our contributions increase the applicability of model-free Reinforcement Learning to real-world tasks, and provide a strong base for future research in that direction.

1.4 Future and Related Research Directions

We started a multidisciplinary and multi-teams research line at the VUB AI Lab and partners, that aims at bringing Reinforcement Learning to deployed systems in the real world. Our published fundamental research, such as Option-Observation Initiation Sets for sample-efficient learning in POMDPs [Steckelmacher et al., 2017], and Bootstrapped Dual Policy Iteration for sample-efficient learning in general [Steckelmacher et al., 2019], is extended in Chapter 3 in this thesis with a real-world robotic proof of concept. Ongoing research at the VUB AI Lab, illustrated in Figure 1.1, will in the coming years lead to complete systems, allowing

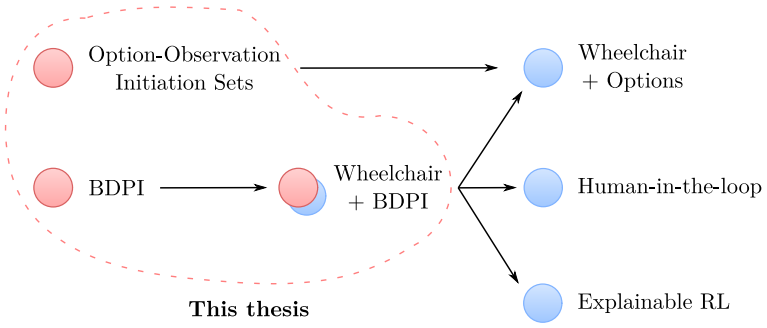

Figure 1.1: The research presented in this thesis serves as the basis of future research at the AI Lab in Brussels. While this thesis starts with theoretical contributions and algorithms, and goes to a proof of concept on a motorized wheelchair, the algorithms we present can be built upon to lead to full systems, amenable for deployment in homes or the industry.

robots to leverage humans when learning, explaining their behavior to them (for increased trust), and learn hierarchical and repetitive tasks. The importance of these research lines is detailed by Carlson and Demiris [2008], who point out that wheelchair users do not want to be passively moved around, but want to remain in control of their robot.

We hope that the work presented in this thesis will allow challenging problems to be solved, thanks to sample-efficient model-free Reinforcement Learning.

2 | Reinforcement Learning

A thermostat is a device we all want to forget about. We want every room of our house or apartment to be at the most comfortable temperature, regardless of the season or particular day, all while minimizing the amount of energy used to heat up or cool down the rooms. We do not want to constantly press buttons, change settings, adjust dials, or discover that the whole house has been heated during a 3-weeks holiday absence. Ideally, the thermostat would masterfully manage the temperature of the house, let it rise or fall when we are away, and have it perfectly comfortable the moment we come back.

The perfect thermostat, something not particularly exciting but definitely useful, does not exist yet. It is the perfect example of an every-day product that has to be intelligent and *adaptive*. Every house is different, the weather changes from day to day, and every user has his or her own preferences. It is impossible to pre-program a thermostat in the factory, and users hate programming one themselves, especially if it has to be done regularly. Ideally, the thermostat should *learn* and *adapt* quickly and automatically, without the user even having to know that it is there. This is where *Reinforcement Learning* shines. In this chapter, we explain what Reinforcement Learning is, how it works, and what are the current challenges in this field. In the next chapters, we present our advances in Reinforcement Learning that make agents more intelligent and better at learning. These algorithms are general, applicable to almost any task. We apply them to a motorized wheelchair, but other researchers at the Vrije Universiteit Brussel apply them to the thermostat described above [De Bock et al., 2017], wind turbines

[Verstraeten et al., 2019], the electrical grid [Mihaylov et al., 2016] or mitigation strategies for pandemics [Libin, 2020].

2.1 Domains and Subfields of AI

Artificial Intelligence is a wide field, that does not have a formal definition. Nilsson [2009], in *The Quest for Artificial Intelligence*, considers that intelligence is anything "that enables an entity to function appropriately and with foresight in its environment", and that artificial intelligence is anything that makes a machine intelligent. We prefer a simpler, albeit less precise, definition of Artificial Intelligence: anything that allows a machine to perform tasks that we would not be surprised to see a person or an animal do. These tasks can be moving and assembling hardware, or thinking, reasoning and creating. To make Artificial Intelligence more tangible, here are three examples of domains of AI that enjoy significant interest from industry and academia:

1. Logic Reasoning and Expert Systems. This part includes the algorithms banks and insurance companies use to compute their rates, computers that answer, understand and process calls in call centers, or manufacturing tools that solve constraint problems.

2. Planning and Optimization, such as scheduling manufacturing jobs, routing goods or people (GPS maps), or optimizing stock bidding.

3. Data Analytics and *Machine Learning*, where a system extracts knowledge from data, for explanation to people or other future use.

In this thesis, we focus on Machine Learning, that can itself be divided in three fields:

1. Supervised Learning, the best-known field, where an agent is given examples of input-output pairs, and learns how to produce new outputs for new inputs. For instance, photos of dogs are given to the agent, annotated with the breed. The agent learns to recognize the breed in new, previously-unseen photos of dogs.

2. Unsupervised Learning, or Clustering, often applied to *anomaly detection*. The agent is given inputs, and learns the structure of its input. Then, new

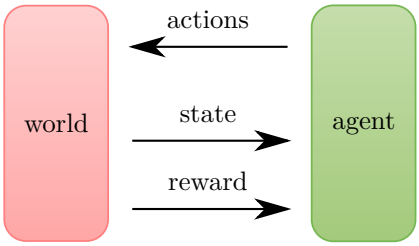

Figure 2.1: The Reinforcement Learning setting. The agent learns which action to execute in which state. For a Reinforcement Learning algorithm, the environment encompasses everything else: the world, a robot if any, what the agent is able to observe, and how the reward is produced.

inputs are given to the agent, that is able to say if it has already seen something similar, and if so, what kinds of already-seen inputs look the most like what it sees. Banks and the industry use Unsupervised Learning to detect when a system diverges from its *usual* behavior.

3. *Reinforcement Learning*, that intuitively lies between Supervised and Unsupervised Learning. The agent is given inputs (called *observations*), is able to execute *actions*, and receives after each action a *reward*. In most settings, the agent receives *sequences* of observations, and executes one action after each observation. The agent learns which actions to execute in which situations in order to obtain the highest sum of rewards along a sequence of observations. The agent is never told *which* action to execute, so it is not supervised. But the rewards guide the agent when learning, so it is not unsupervised either. Because the actions executed by the agent change the observations provided to it, Reinforcement Learning has unique challenges compared to Supervised and Unsupervised Learning, that learn from a fixed pre-defined set of data. This thesis explores these challenges when needed.

To summarize, Reinforcement Learning is a family of Machine Learning algorithms, and Machine Learning is one of the main domains of Artificial Intelligence.

2.2 The Reinforcement Learning Setting

The Reinforcement Learning setting consists of two entities that interact. The *agent* learns and takes decisions. It is an artificial intelligence algorithm, a program that runs on a computer, server, phone or any device. The *environment* is everything else. The environment executes the actions selected by the agent, and produces new states and rewards given to the agent. In practice, the environment is everything that is not the agent: any program or code that processes the actions selected by the agent, the function that rewards the agent, the computer on which the agent runs, the robot itself (if in a robotic setting), sensors, actuators, the room where everything happens, and the physics of the world.

In Reinforcement Learning, time is often considered to be a discrete quantity. There is a subfield of Reinforcement Learning that considers continuous time [Bradtke and Duff, 1995], but it is outside the scope of this thesis. A *time-step* starts when the agent observes the current state of the environment and chooses an action to execute. Then, the action is executed, which may take some time. At the end of the time-step, the agent observes its reward and the next state, and the whole system transitions to the next time-step. Time-steps have an inherent duration, the time needed to execute the action selected by the agent. In simulated environments, the "duration" of an action is the simulated time, for instance one simulated minute. This is distinct from the compute time, the time the simulator takes to simulate the action on an actual computer. In non-simulated environments, such as robots, websites or banking infrastructure, the duration of an action is the real-world, wall-clock time spent executing the action. Accurately reasoning about the duration of a time-step is not important in a purely theoretical or simulated setting. But on physical robots, ensuring that time-steps have a proper duration and meet all the assumptions of Reinforcement Learning is critical. Section 5.6 in Chapter 5 discusses this issue in greater depth.

A sequence of time-steps forms an *episode*. Episodes can be seen like trials. At the beginning of an episode, the agent is put (by the environment) in an initial state, for instance randomly positioned in a room. Agents are assumed to be unable to tell the environment where they want to start an episode. Then, actions are executed in a succession of time-steps. The episode ends when the environment decides it. For instance, an environment can be designed such that the episode ends when a robotic arm either successfully reaches a target position, or spends too many time-steps trying to reach the position. Some Reinforcement Learning algorithms are compatible with *non-episodic* environments, in which the

agent only ever sees one episode of infinite length. The algorithms we present in this thesis are compatible with non-episodic environments, but we only evaluate them on episodic tasks, as obtaining informative statistics, and comparing our agents against a wide variety of other Reinforcement Learning algorithms, is much easier in episodic environments. Most real-world tasks can be designed as episodic without any problem, as shown in Chapter 5.

In Reinforcement Learning, the goal of the agent is to learn an *optimal policy* for the task it solves. There can be several optimal policies for a given task. An optimal policy maps states to actions, so that the behavior of the agent leads it to obtain the highest *(discounted) cumulative reward* possible. The cumulative reward is the sum of all the rewards obtained during all the time-steps of an episode. Discounting gives more weight to rewards obtained sooner than later, which may be important in some settings. Good Reinforcement Learning algorithms not only learn an optimal policy, but do so using as few episodes and time-steps as possible. A thermostat that requires several years to learn how to control the temperature of a room would be useless. Learning this task in only a few time-steps (so, probably a few days), however, is much more useful. We denote as *sample-efficiency* the ability of an algorithm to learn with few time-steps. In Chapter 3, we discuss sample-efficiency in more detail, and present a novel algorithm that significantly improves the sample-efficiency of Reinforcement Learning.

Now that we have introduced what time-steps and episodes are, and what the goal of Reinforcement Learning is, we define the Reinforcement Learning setting more formally.

2.2.1 The Markov Decision Process

In the remainder of this thesis, we formalize the Reinforcement Learning setting as a Markov Decision Process (MDP). An MDP is represented by a tuple $\langle S, A, R, T, \mu_0, \gamma \rangle$, with its six components defined as follows [Bellman, 1957]:

S The state space. In *discrete-state* settings, there is a finite set of states, represented by (possibly labeled) integers from 0 to $|S| - 1$, with $|S|$ the number of states that exist. An example of a discrete-state setting is a task where the agent is able to observe in which if three states a water tank is: mostly empty (0), mostly full (2), or in-between (1). In *continuous-state* settings, common in real-world problems, the set of states is infinite. The state is not an integer anymore, but a vector of real values, such as sensors readings, distance measures, temperatures, pressures, or RGB values of pixels in

an image. With sensors or cameras, it is sometimes difficult to ensure that the agent observes the full and precise state of its environment (cameras do not see through walls). We discuss this *partial-observability* challenge in Chapter 4, and show in Chapter 5 that a sufficiently small amount of partial-observability generally does not prevent a Reinforcement Learning agent from learning good policies.

A The action space. With *discrete actions*, the agent has access to a finite number of actions, $|A|$, and numbers each action between 0 and $|A| - 1$. Executing an action can be compared to pressing a button. With *continous actions*, the action becomes a vector of *several* real values, as if an octopus was riding a car with multiple steering wheels, that can each move with infinite precision. Continuous actions are highly-general, and are an active area of research in Reinforcement Learning. However, high sample-efficiency currently seems challenging with continuous actions, with the most advanced algorithms often being trained on simulated tasks for tens of millions of time-steps [Haarnoja et al., 2018]. In this thesis, we focus on discrete actions, show that they allow highly-complicated tasks to be performed, and leverage their discrete nature to achieve high sample-efficiency.

R The reward function, that maps a state and an action to a single real value.

T The transition function, that maps a state and an action to a next state. The transition function, that defines what happens in the environment when an action is executed, is unknown to the agent. Thanks to that, Reinforcement Learning is applicable to extremely complicated settings, in the real world, for which the transition function may be unknown to Humanity in general.

μ_0 The initial state distribution, unknown to the agent, that defines where the agent may start at the beginning of an episode.

γ The discount factor, a positive value below 1. The closer γ is to 1, the more the agent trades short-term rewards for long-term ones.

The goal of the agent is to learn a policy, usually denoted as the function π that maps any state to an action, that allows the agent to collect the highest-possible discounted sum of rewards, in any episode. Such a policy is considered *optimal*. Some Reinforcement Learning settings, especially in non-episodic environments, consider an alternate problem: learning a policy that collects the highest-possible reward *on average, every time-step* [Mahadevan, 1996].

Keeping in mind the 6 components of an MDP is important both when reasoning formally on Reinforcement Learning problems, but also when designing them. Intuitively, for Reinforcement Learning to be the solution to a problem, it must be possible to define what these 6 components are, as discussed in the next section.

2.2.2 When to use Reinforcement Learning?

Most applications of Reinforcement Learning consist of designing an environment, the MDP described above, and deploying an existing Reinforcement Learning algorithm in this environment. The challenging part of applications of Reinforcement Learning is the design of an environment in which learning happens smoothly. This thesis focuses on the learning algorithm side of the problem, but we regularly discuss properties of the environment that influence learning, and to which we adapt our learning algorithms.

The key idea behind identifying a problem that can be solved with Reinforcement Learning is identifying, in approximate order of importance, what would be the actions, how the agent would be rewarded, and what would it observe. We believe that a long explanation of which kinds of tasks are suited to Reinforcement Learning would be overly convoluted and abstract. We instead discuss in which situations *not to apply* Reinforcement Learning:

There is no reward
: Or the reward function is trivial. If it is impossible to define a reward function, or if the reward function simply rewards the agent for doing exactly the action defined by the designer, then Supervised Learning should be used instead of Reinforcement Learning. Example of Supervised Learning tasks are a robot that learns to imitate the actions executed by a human operator, or a decision system for which we have examples of inputs and expected actions.

There is no action
: A system that predicts the temperature tomorrow, or whether it will rain, is not a Reinforcement Learning agent. The output of a system is a continuous or discrete prediction, that is used as is by human operators (or another system). It is not an action that will cause an environment to change state. Supervised Learning is better suited to this setting.

There is no state

Our personal discussions with businesses sometimes lead to the description of a problem for which there is no state, or no meaningful state. While many *state-less* settings are valid, and can be solved with *Bandit* algorithms, a special family of Reinforcement Learning algorithms that we discuss below, the absence of a state can also prevent any form of Reinforcement Learning from being applied. For instance, a blind agent is given the task to learn the most intimate thoughts of a visitor, or a car without a camera has to avoid pedestrians. Even for tasks such as controlling a wind turbine or a power plant, it is important to think about whether the state that will be given to the agent, such as measurements, is enough for a person to at least be able to imagine an (even low-quality) policy for the task.

The solution is already known

Reinforcement Learning is suited for settings where *super-human performance* is the goal. A good example is given by Lazic et al. [2018], where an agent learns to control the cooling of a datacenter in a significantly more energy-efficient way than a team of engineers. If the problem can already be solved satisfactorily with an existing algorithm, planning in an existing model, or in any other way, Reinforcement Learning will not provide any benefit. Balancing pendulums, routing vehicles or estimating the location of a device from few observations, already have optimal solutions provided by algorithms that are not Reinforcement Learning.

Reinforcement Learning would not be trusted

In the general setting, a Reinforcement Learning agent learns a black-box policy, often in the form of a neural network. The performance or safety of the agent can be empirically measured in simulation or on the real-world task, but only on a finite amount of test-cases. There is currently no general approach at proving that an agent will always execute good actions. Work on Explainable Reinforcement Learning, reviewed by Puiutta and Veith [2020], aims at increasing the trust that can be expressed towards Reinforcement Learning. In Section 5.4.1, we discuss how a backup policy can also increase trust in a Reinforcement Learning agent, but the general guideline remains that Reinforcement Learning should not be used in any system with high risks (lives, expensive damages, ...).

To sequence or not to sequence time-steps

In this thesis, we focus on the setting where the agent executes actions for

a *sequence of time-steps* in an episode. If the problem consists of observing an input, executing an action, then move on to the next (unrelated and not influenced by the action) input, then the problem is a *contextual multi-armed bandit*. If the agent does not observe any state, but has to quickly learn which action to execute to achieve the highest expected reward, the problem is a (not contextual) multi-armed bandit [Berry and Fristedt, 1985]. Bandit algorithms are closely related to Reinforcement Learning algorithms, and algorithms presented in this thesis are applicable to bandits. However, a complete review of every Bandit algorithm is outside the scope of this thesis, even though some of them are mentioned when relevant.

If none of the points above apply to a problem, and it is possible to define states, actions and rewards, then Reinforcement Learning is applicable to the problem.

2.2.3 Designing Reward Functions

An important step in the development of a Reinforcement Learning environment is the design of a reward function. Two opposing objectives have to be balanced when designing the reward function:

1. A complicated but informative reward function, that precisely evaluates most or every action of the agent, has the potential (if done correctly) to allow the agent to quickly learn a good policy. It is like looking for an object in a room, and having someone constantly giving rewards depending on the exact distance between us and the object. Only a small amount of time is required to discover where the desired object is. However, designing complete and informative reward functions takes time and effort, exactly what most people want to spare by using Reinforcement Learning instead of a hand-coded policy. Complicated reward functions also have a higher chance to be erroneous, to mis-guide the agent, or to express an objective that is not exactly the true objective of the task.

2. A simple reward function, that encodes the purest essence of the goal of the agent, such as "reduce mortality on the road" (without giving any hint about how to reduce that mortality), is often highly challenging to optimize for by the agent [Arjona-Medina et al., 2018]. Most of these simple reward functions are *sparse*, meaning that they provide a non-zero reward to the agent only in few rare states. A common form of a sparse reward function is the one that gives a zero reward to the agent every time-step, and a 1 only

when the agent reaches or achieves the goal of the task. With such a reward function, the agent has to try many actions, having no idea of whether it is getting closer to the goal or not, until it reaches it for the first time by chance.

Balancing the two approaches above is an active area of research in Reinforcement Learning. For instance, potential-based reward shaping [Ng et al., 1999; Knox and Stone, 2010] gives as reward to the agent the sum of two functions: the original reward function produced by the environment, that encodes the goal of the agent, and a *shaping function* that gives additional positive and negative rewards to the agent, and is produced algorithmically by the designer. The potential-based aspect of the algorithm comes from the fact that the additional function is constrained to a specific form, having specific mathematical properties, that roughly translate to rewarding the agent when it gets closer to a goal, and punishing it when it gets farther (the *punishing* part is often forgotten when someone designs a shaping function). Recent work proposes a method for transforming any shaping function, that gives rewards and punishments in any way that feels natural to the designer of the system, to a potential-based shaping function having all the properties needed for effective learning [Harutyunyan et al., 2015]. Even when a potential-based shaping function is available, some challenges remain, such as how to best combine the environment reward and the shaping reward signals. We refer the interested reader to Brys et al. [2015] for a discussion of these challenges.

The approach of Arjona-Medina et al. [2018] does not require an additional reward function to be designed. Instead, their algorithm observes the rewards received by the agent, and learns which actions, in which states, may have contributed to those rewards the most. Then, they use that learned information to provide *encouragement* rewards to the agent when it executes these promising actions. The main property of the work of Arjona-Medina et al. [2018] is that the agent *learns* its additional reward function itself. Related approaches include Curiosity-Driven Exploration [Still and Precup, 2012; Pathak et al., 2017], where the agent uses several methods to evaluate how surprised it is to see a state, and gives itself additional rewards when it encounters rarely-seen states. Bootstrapped methods, that we review in Section 2.4.2 and use as part of the algorithm we present in Chapter 3, also allow an agent to discover promising regions of its environment, but do not compute nor give any additional reward to the agent. They focus on allowing the agent to efficiently learn with sparse reward functions *as is*.

Finally, Plisnier et al. [2019a] introduce a method that does not modify the rewards given to the agent, but instead suggests actions (*advises* them). This allows simple, pure but sparse reward functions to be used as is, while still achieving acceptable sample-efficiency by enticing the agent to execute promising actions (as defined by the designer). As time passes, the agent learns when to follow the advice it receives, and when to ignore it, which allows the agent to learn the optimal policy even if the advice it receives is sub-optimal. We use this framework in Chapter 5 to encourage our wheelchair to move forwards, leading to a more efficient discovery of its environment.

2.2.4 The OpenAI Gym

Before presenting one of the most widely used Reinforcement Learning algorithms in the next sub-section, we take a moment to discuss practical details such as code, and the frameworks we use in the rest of this thesis.

The OpenAI Gym [Brockman et al., 2016] is a Python library that implements many simple (and some less simple) environments, commonly used in the Reinforcement Learning literature to compare algorithms with each other. Before the publication of the OpenAI Gym, researchers had to constantly re-implement environments, or give them to each other in the darkest alleys of university campuses. In addition to providing environments, the Gym also defines a standard Python interface between an agent and an environment, that is simpler to use than previous attempts at an unifying framework, such as RL-Glue[1]. This simplicity comes at the cost of language agnosticism, as the Gym requires the agent and the environment to be implemented in Python, and running in the same process, while RL-Glue allowed these two components to be implemented in different languages, running in different processes, possibly on different computers. As such, the Gym changed the Reinforcement Learning research field in two ways:

1. Python became the de-facto programming language for Reinforcement Learning. Before, it was often used when Reinforcement Learning was combined with neural networks, as neural network libraries are predominantly written in Python, but other Reinforcement Learning systems used a variety of other programming languages, such as C++ and Java.

[1] https://sites.google.com/a/rl-community.org/rl-glue/Home

2. A single API has been defined between an agent and an environment. Any environment that follows the Gym API can be used with any learning algorithm that understands the API. It is perfectly possible to download an interesting webcam-based navigation environment from the Internet, and combine it with the latest and greatest Reinforcement Learning algorithm without much effort.

Any environment that follows the Gym API is first *instantiated*. Then, for each episode, the environment is *reset*, then a succession of time-steps are *stepped*. Instantiating an environment can be done in two ways, as shown below:

```
import gym

# First way, a standard environment known to Gym
env = gym.make('LunarLander-v2')

# Second way, a custom environment
class MyEnv(gym.Env):
    def __init__(self):
        super().__init__()

        self.observation_space = gym.spaces.Discrete(16)
        self.action_space = gym.spaces.Discrete(4)
        self.current_state = 0

    def reset(self):
        self.current_state = 0      # Go back to the initial state
        return self.current_state

    def step(self, action):
        # Change state and compute reward according to action
        # ...

        return self.current_state, reward, done, {}

env = MyEnv()
```

The first approach is used for environments shipped with the Gym. The name to give to `gym.make` is defined in the Gym documentation.

The second approach shows how to create a custom environment that follows the Gym API. The resulting environment is instantiated exactly like any other Python class. The API to be followed by a Gym environment consists of two mandatory attributes, and two mandatory methods. The attributes, observation_space and action_space, describe what the agent observes in the environment and how many actions in can execute. In this example, we use a discrete state space, that contains 16 states, numbered from 0 to 15. We also use a discrete action space, of 4 actions. Continuous state-spaces (vectors of floats) and action-spaces are also possible, and will be detailed later in this thesis. The two mandatory methods allow the environment to be reset to its initial state (and returns said initial state), and allow an action to be performed in the environment. The return value of step is a tuple of 4 elements that represent, after the action has been executed, the next state, the reward that has been obtained by executing the action, whether the episode should terminate, and a dictionary in which the environment can add any extra information, in any format it wants. Most current Reinforcement Learning agent implementations ignore this fourth element, that is therefore often left as an empty dictionary, as in the example above.

On the agent side, the environment is created, then two nested while loops respectively iterate over episodes, and over time-steps in the episodes:

```
import gym

env = gym.make(environment_name)

while True:
    # Do an episode
    state = env.reset()
    done = False

    while not done:
        # Execute an action
        action = 0
        next_state, reward, done, _ = env.step(action)

        # Move to the next time-step
        state = next_state
```

The minimal agent shown above always executes the first action (numbered 0), and does not perform any form of learning. We now introduce one of the best-

known Reinforcement Learning algorithm, that will allow this minimal agent to learn how to perform a task.

2.3 Q-Learning

Q-Learning forms the basis of many Reinforcement Learning algorithms. Invented by Watkins and Dayan [1992], its core idea is to learn how promising every action is in every state, so that the most-promising action, the one having the largest *Q-Value*, is the action that should be executed by the agent. More formally, Q-Learning is an algorithm that allows to learn a *Q-Table* of Q-Values, for every state s and every action a, that, under some assumptions [Tsitsiklis, 1994; Munos et al., 2016], converges to the Q-Table $Q^*(s, a)$ of the optimal policy:

$$Q^*(s, a) \equiv \mathbb{E}_{\pi^*}\left[\sum_t \gamma^t r_t \middle| s_0 = s, a_0 = a\right] \tag{2.1}$$

The above equation can be understood as follows: the Q-Value $Q^*(s, a)$ is the expectation of the discounted sum of all the rewards that the agent will get, when being is state s, executing action a, then following the policy π^* until the end of the episode. The π^* notation denotes the optimal policy. This is where the power of Q-Learning is: it progressively learns how good an action would be if the optimal policy were to be followed. Executing the actions with the highest Q-Values therefore leads to executing the optimal policy.

Equation 2.1 provides the mathematical definition of what a Q-Value is. In order to actually learn the Q-Values, a set of update rules (2.2) are used to iteratively compute an updated Q_{k+1} Q-Table based on a previous Q-Table Q_k and a (s_t, a_t, r_t, s_{t+1}) *experience tuple*. The value of k increments every time an *update* is performed, and appears in equations to make them mathematically exact. Practical implementations of Q-Learning, such as presented in Section 2.3.2, usually do not explicitly represent k. In simple cases, an update is performed after every time-step, when an experience tuple is produced and used for learning. In Section 2.3.3, we discuss how updates can be made more or less often than every time-step, on more than one just experience tuple at once.

$$\begin{array}{rcl} y & = & r_t + \gamma \max_{a'} Q_k(s_{t+1}, a') \\ Q_{k+1}(s_t, a_t) & \leftarrow & Q_k(s_t, a_t) + \alpha(y - Q_k(s_t, a_t)) \end{array} \tag{2.2}$$

Our notations slightly differ from the ones used in the Reinforcement Learning literature, such as Sutton and Barto [1998], but these differences allow concepts in the next sections to be introduced without having to change notation on the way. Equation 2.2 computes the updated Q-Value of one action in one state, and is usually executed after every time-step t. During that time-step, the state s_t has been observed, a_t has been executed, which led to the reward r_t and next state s_{t+1}. Before the agent learns, so before the first time-step, the Q-Table is usually initialized either with all zeros, with random values close to zero, or with specific values that the designer may already know (a state can be known to be very bad even before learning starts, for instance). The Q-Table Q_0 is therefore most probably completely different from the optimal Q-Table Q^* that the agent aims to learn. As more Q-Learning iterations are performed, assuming a small learning rate $0 < \alpha < 1$ that decreases over time in the right way, and adequate exploration [Watkins and Dayan, 1992], $Q_{k\to\infty}$ will converge to Q^*.

Equation 2.2 shows how to learn Q-Values from experiences. Which action to execute at every time-step can now be computed from the Q-Values. The general idea is that, in state s, the agent gets the Q-Values $Q_k(s, a)$ for every action a, then executes the action that has the largest value. This produces the *greedy policy*, the policy that consists of always executing the action the agent *believes* is the best at Q-Learning iteration k. In practice, the greedy policy is never used when learning. When the agent has just started to learn, its Q-Values are almost random, highly inexact. The greedy actions are therefore usually sub-optimal. Additional *exploration* has to be introduced to the agent, so that it also tries actions it does not believe to be the best ones, leading to potential good surprises. Exploration strategies can be divided in three families, used in practice about equally as often:

ε-Greedy Every time-step, a random action is executed with probability ε. The greedy action is executed with probability $1 - \varepsilon$. Usually, the ε probability is set to 0.1 or 0.2, but it is possible to change it over time, or adapt it in various ways [Sutton and Barto, 1998].

Softmax A more complicated approach that takes into account the relative quality of every action, instead of just which one is the best one. The probability of executing action a_t in state s_t is set to $\frac{\exp(\tau Q_k(s_t,a_t))}{\sum_{a'}\exp(\tau Q_k(s_t,a'))}$, with τ an *inverse temperature* [Cesa-Bianchi et al., 2017, a recent discusion of Softmax in Reinforcement Learning]. The closer τ is to 0, the more randomly the agent behaves. If τ is larger, for instance 5, then the agent focuses more on

actions that have larger Q-Values. The term *temperature* has been borrowed from the field of physics, and basically relates how large temperatures (small inverse temperatures) lead to randomly-chosen actions, to how large temperatures in the physical world translate to atoms and molecules oscillating and randomly moving faster.

Probabilistic Methods such as UCB [Auer, 2000] and Thompson Sampling [Thompson, 1933; Chapelle and Li, 2011] use additional statistical sources of information to select actions, such as how many times an action has been tried, what the distribution of returns looks like, etc. These methods lead to high-quality exploration. Empirically, Thompson sampling outperforms every other exploration strategy regarding sample-efficiency and the quality of the learned policy [Chapelle and Li, 2011]. However, UCB and Thompson Sampling are challenging to implement. For instance, UCB requires state visits to be counted, which is impossible to do in continuous state-spaces. Thompson Sampling, in a Reinforcement Learning setting, needs per-state statistics about the actions, something that is also impossible to implement with continuous states, as the state can take an infinite amount of values. In Section 2.4.2, we review approximate exploration strategies that lead to results as good as most probabilistic methods, while being applicable to a wide range of settings. Our work, especially in Chapter 3, builds on these methods to achieve an exploration quality comparable to Thompson sampling, yet compatible with continuous state-spaces.

In addition to high-quality exploration as discussed above, other aspects of Q-Learning have been extended over the years to increase its sample-efficiency, critical for real-world applications of Reinforcement Learning. Before introducing these extensions, we discuss the actual implementation of a simple tabular Q-Learning agent in the next section, as the first step of our Reinforcement Learning tutorial.

2.3.1 Numerical Processing with NumPy

Throughout this thesis, we use the NumPy Python library to store numbers, and perform computations on them. NumPy is a numerical processing library, that contains many advanced functions, such as Fourier transforms and spectral decompositions of matrices. Even if considered a Python library, NumPy is implemented with highly-optimized C code. This means that any processing done

with NumPy functions will be much faster than re-implementing the same processing in Python. In this thesis, we mainly use NumPy for its convenient way of representing and managing vectors and tensors of floating-point values.

A *tensor* is a multi-dimensional generalization of a matrix. Basically, a one-dimensional tensor is a vector, and can be seen as a list of numbers. A two-dimensional tensor is a matrix. Higher-dimensional tensors are perfectly possible, with no limit imposed by NumPy. In some specific settings, such as Reinforcement Learning agents that observe images, tensors of four dimensions are used.

```
import numpy as np

t1 = np.array([1, 6, 3])
t2 = np.array([ [1, 2, 3], [4, 5, 6]])
```

Tensors are easily created from Python lists, as shown above. The `np.array` function takes as argument any iterable, and creates a tensor from it. If a list is given to it, it returns a one-dimensional tensor whose elements are the contents of the list. If a list of lists is given to it, a 2-dimensional tensor is returned. If a list of lists of lists of lists is given, a 4-dimensional tensor is returned, and so on.

NumPy tensors offer two main advantages. First, they offer compact notations for many operations that come in handy when implementing Reinforcement Learning algorithms. For instance, `t2[:, 0]` considers the entire first dimension of `t2`, and only the first element of its second dimension. With `t2` defined as in the code snippet above, `t2[:, 0]` is a 1-dimensional tensor that contains 1 and 4. Another example is `t2[0, :]`, that can be shortened to `t2[0]`, and returns the first element of the first dimension of the tensor, in our case a 1-dimensional tensor that contains 1, 2 and 3. While difficult to grasp at first, NumPy tensor operations are useful, and our use of them will always be textually explained in this thesis.

The second advantage of NumPy tensors is that many mathematical operations are defined on them, and apply to all their elements in one function call. For instance, calling `np.sin(t2)` returns a 2-dimensional tensor that contains 6 elements, each element being the sinus of the corresponding element in `t2`. This single function call saves two Python for loops. Moreover, because NumPy is implemented in C and uses efficient representations and storage, calling `np.sin` on a big tensor is several orders of magnitude faster than performing multiple for loops in Python, and repeatedly calling Python's `math.sin` function.

Lastly, NumPy provides useful *reduction* functions, that summarize tensors. Statistical functions such as `np.mean` and `np.std` return the average value and

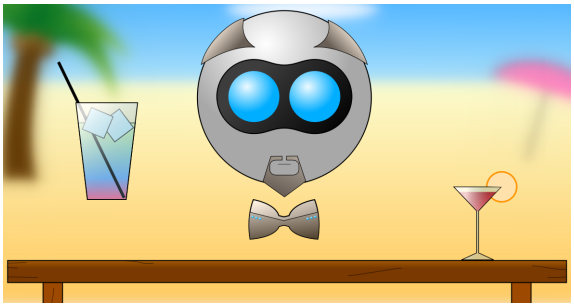

Figure 2.2: A simple virtual bartender that learns to ask people what drink they want, and how to give it to them.

standard deviation of the numbers found in a tensor (np.mean(t2) returns 3.5). Ordinal functions such as np.max, np.min and np.argmax return the minimum or maximum element of a tensor, or the index at which the maximum element is (np.argmax(t1) returns 1). In the next section, we provide an implementation of Q-Learning. We use a NumPy array to store the Q-Value of every action in every state, which leads to clean and short code.

2.3.2 Tabular Q-Learning

The formulas given in the previous section consider states and actions, but do not specify any data type these quantities should have. In practice, the state can be an integer, a vector of floating-point values (such as sensor readings), or an image [Mnih et al., 2015]. Some very specific applications of Reinforcement Learning have other state representations, such as graphs or strings [Graves et al., 2016], but they are less common. The action is usually discrete or continuous. Discrete actions represent specific actions out of a finite set of actions, for instance pressing a particular button. Discrete actions are usually represented with integers, for instance by numbering the action from 0 to the number of available actions (excluded). Continuous actions represent real-valued control signals, such as steering a wheel at any angle out of the infinite set of possible angles. Continuous actions are usually presented with floating-point values, with *the* action executed at time t a possibly-long list of floating-point values. While the type of the state minimally impacts the algorithm, the action has a larger impact, and

having continuous actions requires non-trivial changes to the formulas shown in Section 2.3.

In this section, we illustrate Q-Learning on a simple task where both the state and the action are discrete, and that we represent with integers. Figure 2.2 depicts a virtual bartender, that we regularly use at the VUB Artificial Intelligence Lab in Brussels to explain what Reinforcement Learning is. The bartender is able to pronounce a few pre-defined sentences. The user, who interacts with the bartender, replies by pressing the *yes* or *no* buttons. Two glasses are part of the environment, one on the left, and one on the right. The task is defined as follows:

States 4 states: *Initial (0)*, when agent starts a new interaction with a new customer, *Hello (1)*, to which the agent moves whenever it greets the customer, *Left (2)* and *Right (3)*, that correspond to the desired glass of the customer.

Actions 8 actions: a quick greeting *(0)*, a polite greeting *(1)*, asking whether the user wants the left glass *(2)*, or the right glass *(3)*, saying "a moment please" *(4)*, giving the left glass *(5)*, giving the right glass *(6)*, and apologizing *(7)*.

Transition function The agent starts in *Initial*. When it asks the user for a glass, the answer of the user is used to move the agent to either the *Left* or *Right* states. When in *Initial*, the quick or polite greetings move the agent to the *Hello* state.

Reward function The reward is -0.1 most of the time (the agent is therefore punished for every action, which entices it to finish the episode as quickly as possible). A reward of +2 is given when the user accepts a glass presented to it, +3 if the user also considers the agent to have been polite, -1 when the glass is rejected.

The source-code of this environment, and the entire demo it leads to, is available in the appendix. We can now implement a simple tabular Q-Learning agent for this environment. The *tabular* part of the agent comes from the fact that the states and actions are discrete. It is therefore possible to store Q-Values in a 2-dimensional matrix $Q(s, a)$, whose first dimension has 4 entries (one per state), and second dimension has 8 entries (one per action). We use the Numpy library to work with vectors and matrices in Python, as it makes the code easy to read and compact.

```
import numpy as np

qtable = np.random.random((4, 8)) * 0.01
```

The random function returns floats sampled uniformly in the -1 to 1 range. As a result, the code above creates a 4×8 Q-Table, with every Q-Value initialized to a random value between 0 and 0.01. Randomly initialized Q-Values to values close to zero is a good practice, as it slightly increases exploration during the very first time-steps, compared to an all-zeros initial table. Once the Q-Table exists, the code given in Section 2.2.4 can be extended so that the agent *uses the Q-Values to select actions* and *updates the Q-Values after every time-step*:

```
while True:
    state = env.reset()
    done = False

    while not done:
        # Select the greedy action (the one with the highest Q-Value)
        action = qtable[state].argmax()

        # Implement ε-Greedy, with ε = 0.1
        if np.random.random() < 0.1:
            action = np.random.randint(4) # There are 4 actions

        # Execute the action and learn from it with Equation 2.2
        next_state, reward, done, _ = env.step(action)

        if done:
            y = reward # There is no valid next state
        else:
            y = reward + 0.99 * qtable[next_state].max()

        qtable[state, action] += 0.1 * (y - qtable[state, action])

        # Move to the next time-step
        state = next_state
```

with the learning rate α set to 0.1, and the discount factor γ set to 0.99.

2.3.3 Experience Replay

The basic Q-Learning algorithm detailed in Sections 2.3 and 2.3.2 updates one Q-Value per time-step, using the latest experience obtained from the environment.

An *experience tuple* is a $(s_t, a_t, r_t, s_{t+1}, d)$ tuple, that identifies which action has been tried in a particular state, what reward it led to, to which state the environment transitioned after having executed the action, and whether the episode terminates (d for *done*) at that time-step.

While Q-Learning is able to learn Reinforcement Learning tasks, it does not make efficient use of the experiences collected from the environment, as it has no memory, and discards experiences as soon as they have been used once. Experience Replay [Lin and Mitchell, 1992] extends Q-Learning with a long-term memory of past experiences. The intuition behind Experience Replay is simple: every experience collected by the environment is added to a *buffer*, and, every time-step, *several* experiences from that list are sampled and used to update *several* Q-Values. Usually, the experiences are sampled uniformly at random from the buffer. Recent research shows that prioritizing experiences based on how significant the agent believes they are further improves sample-efficiency [Moore and Atkeson, 1993; Schaul et al., 2015; Zhang and Sutton, 2017]. In this thesis, we use Experience Replay with uniform sampling, to keep our algorithms simple, and to make sure that the high learning performance of our agents comes from our algorithmic contributions, not other components that are not original work.

Implementing Experience Replay in Python is surprisingly simple, as tuples are a native type of the programming language. Storing a $(s_t, a_t, r_t, s_{t+1}, d)$ tuple in a list can therefore be done in one single line. Here is the inner-most loop of Tabular Q-Learning extended with experience replay:

```
experiences = []

while True:
    # [...]
    while not done:
        # Select an action with ε-Greedy
        # (see previous code snippet)

        # Execute the action and store the experience
        next_state, reward, done, _ = env.step(action)
        experiences.append((state, action, reward, next_state, done))

        # Learn from 10 experiences
        # (discount factor set to 0.99, learning rate to 0.1)
        for index in np.random.choice(len(experiences), 10)
```

```
            s, a, r, sn, d = experiences[index]

            if d:
                y = r
            else:
                y = r + 0.99 * qtable[sn].max()

            qtable[s, a] += 0.1 * (y - qtable[s, a])

        # Move to the next time-step
        state = next_state
```

Experience Replay dramatically increases the sample-efficiency of the agent, as each experience is now used *many* times. In our virtual bartender demonstration, we go as far as replaying *every* experience in the buffer every time-step. This allows the bartender to discover that it has to ask which glass the customer wants, and to give him or her that glass and not the other one, in about 10 minutes of actual interaction with a person. Learning a task like this, using only a few dozen interactions with the environment, demonstrates the power of Tabular Q-Learning with Experience Replay.

2.3.4 Eligibility Traces

Eligibility Traces are another method that increases the sample-efficiency of Q-Learning. Instead of storing full experiences in a buffer, the agent remembers the history of states it has visited in the current episode, and keeps on updating them with new experiences. The older a state is, the more weakly it is updated. Eligibility traces enjoy active research, as surveyed by Geist and Scherrer [2014]. While eligibility traces do not increase sample-efficiency as much as experience replay, because they have to clear their memory after every episode, eligibility traces lead to stable algorithms in challenging environments, as discussed by Precup [2000a] and used by Munos et al. [2016] to prove the convergence of numerous variants of Q-Learning.

2.4 Uncertainty and Bootstrapped Methods

In Section 2.3, we discuss simple exploration schemes that allow the agent to sometimes take an action it believes is not optimal, so that new opportunities and

good surprises are possible. We also briefly mention exact statistical methods, such as UCB [Auer, 2000] and Thompson Sampling [Chapelle and Li, 2011; Deisenroth and Rasmussen, 2011], that compute which action to execute, or which probability to associate with each action.

In this chapter, we focus on uncertainty in Reinforcement Learning. The intuition is that if the agent is certain about the accuracy of the Q-Values in the current state, then it can safely execute the action that has the largest Q-Value. If the agent is uncertain, then tending towards a more random action is important [Strens, 2000]. How to measure or evaluate the uncertainty of the agent is highly challenging, and an active area of research. We present two approaches.

2.4.1 Uncertainty-Aware Algorithms

A few recent Reinforcement Learning algorithms build on Q-Learning and explicitly compute and manage the uncertainty of the agent. Distributional Reinforcement Learning [Bellemare et al., 2017] replaces point estimates of Q-Values (one Q-Value per state-action pair, that estimates the expected return to be obtained from that state-action pair) with a full distribution over the returns that the agent obtains by executing action a in state s. For each state and each action, $Q(s, a)$ is a large vector of floating-point values, that, like an histogram, encodes the probability that the return falls into any particular interval. For instance, if 50 *buckets* are used and we know that returns range from 0 to 100, each $Q(s, a)$ is a vector of 50 floats. The first bucket stores the probability that the return is comprised between 0 and 2, the second bucket between 2 and 4, etc. A more complicated version of Equation 2.2 updates all the buckets of the Q-Values being updated.

Bayes-by-Backprop [Blundell et al., 2015] is an uncertainty-aware variant of neural networks. Neural networks, that we discuss in Section 2.5, are used by Reinforcement Learning agents to store Q-Values, much like the Q-Table of Section 2.3.2 but for continuous states, represented as lists of floating-point values. This means that if the neural network that stores the Q-Values is able to express an uncertainty about its output, then the agent has access to uncertainty estimates about its Q-Values, and can explore more when the uncertainty is high. The exact working of Bayes-by-Backprop is outside the scope of this thesis, but mainly consists of replacing every floating-point value that appears anywhere in a neural network (and there are many) with an estimation of their distribution. As in Distributional Reinforcement Learning, presented in the paragraph above, a particular floating-point value is not 0.2 anymore (for example), but a full description of all the values it is likely to take. With distributions of floating-point values used

throughout the network, its output also becomes distributional. Given this distributional output, the agent computes statistical measures, such as the standard deviation of a Q-Value, that map well to the concept of *uncertainty*.

Finally, Plappert et al. [2017] takes a different approach, and focuses on the goal of *exploring efficiently*. Estimating uncertainty is not the goal *per se*, but an efficient means of achieving high-quality exploration. Instead of estimating the uncertainty of the agent, Plappert et al. [2017] introduce noise in the Q-function of the agent, and use that noise to force higher exploration in states that are not visited often. Every time Q-Values for a state have to be accessed, they add a small random value to every floating-point value that encodes the neural network. This noise will propagate to the output of the network (the Q-Values), and change them in a random way. The magnitude of the change depends on how robust the network is to perturbations, for that particular state. Even if exactly characterizing this change is impossible, Plappert et al. [2017] observe that it has a larger magnitude in states that are seldom visited, and smaller magnitude in states that are visited more often. The agent now always executes the action having the largest Q-Value. In well-known states, this will always be the same action, the one that has a high probability of being the best one. In less-often visited states, the ever-changing noise will cause large random changes in the Q-Values, so that which action has the largest Q-Value will constantly change. This leads to high exploration in seldom-visited states.

2.4.2 Bootstrapped Methods

The approach of Plappert et al. [2017], described in the previous sub-section, is very interesting for one reason: it allows high-quality exploration, even if the uncertainty of the agent is not actually computed. This is of capital importance. The objective of Reinforcement Learning is to produce policies that explore well and achieve high returns. The exact method used to achieve these goals is usually not relevant. Even if the motivation behind uncertainty-aware algorithms is sound, other methods also allow for high-quality exploration, such as bootstrapped methods.

Bootstrapped methods build on the observation by Efron and Tibshirani [1994] that learning the same thing (Q-Values for instance) several times, from slightly different sets of data, will lead to several outcomes that follow a probability distribution very close to the real probability distribution of the data. We refer to learning several instances of a *thing* as *bootstrapping* the *thing*. In a Reinforcement Learning setting, learning several Q-functions from different sets of experiences will

lead, in every state, to the Q-Values of every action to be somewhat different according to every Q-function. The more different they are (the less the Q-functions agree, or equivalently the higher the *variance* of the Q-Value is), the more it indicates that the agent does not have a precise idea of what the true Q-Value is.

Osband et al. [2016] propose a method, Bootstrapped DQN, that leverages bootstrapped Q-functions for high-quality exploration, and demonstrate that quality in challenging environments. Bootstrapped DQN learns every Q-function with a variant of Q-Learning with Experience Replay. When replaying experiences, each Q-function sees a different set of experiences. As per the bootstrap, the Q-functions will learn comparable Q-Values in well-known states, and different Q-Values in lesser-known states. Osband et al. [2016] then propose, for each episode, to randomly select one Q-function whose greedy policy will be followed. By changing Q-function every episode, instead of every time-step for instance, they argue that the agent performs *deep exploration*: performing many time-consistent actions that go in the same direction, so that parts of the environment far from the initial state can be explored. In Chapter 3, we propose an algorithm that also learns several Q-functions, but uses all of their Q-Values every time-step, without that notion of *one Q-function has control for an entire episode*. Our good empirical results seem to indicate that deep exploration is either not needed (even in the highly-challenging environments we use), or arises even when all the Q-functions have their output combined.

2.4.3 Bootstrapped Tabular Q-Learning

In Chapter 3, we learn several Q-Functions, and show that they play an important role in the high sample-efficiency and exploration quality of our algorithm. Because of the importance of bootstrapping Q-Functions in our algorithms, and because giving an intuition for it is difficult, we now continue working on our Tabular Q-Learning agent (see Sections 2.3.2 and 2.3.3) and make it able to use several Q-Tables. The general idea is as follows:

At Learning Time Maintain N Q-Tables. Every time-step, for each Q-Table, sample a distinct set of experiences from the experience buffer, and update the table. This adds a for loop to the code, compared to non-Bootstrapped Q-Learning (see below).

At Acting Time Select a Q-Table at random, get the Q-Values of the current state from it, and execute the action with the largest Q-Value (the greedy

action). Bootstrapped DQN [Osband et al., 2016] proposes to choose a Q-Table for every episode, and to stick to it for the entire episode, but we empirically found that choosing a Q-Table every time-step leads to higher sample-efficiency in tasks that do not have an unrealistically sparse reward function.

The agent starts with the initialization of an experience buffer, and a few Q-Tables:

```
N = 16        # We use 16 Q-Tables in this example
qtables = [np.random.random((4, 8)) * 0.01 for i in range(N)]
```

Remember that in our example of the virtual bartender robot, there are 4 states and 8 actions, hence the 4 and 8 numbers appearing in the code above. In Python, a reference-based language, it is important to ensure that the 8 Q-Tables we create are 8 different objects, and not 8 references to the same object. The list comprehension we use above ensures that 8 random arrays are created. Using the [np.random...] * N notation would be incorrect, as would create a single array, and then replicates its reference N times. Any update to one of the elements of this array would therefore update all the other ones, defeating the purpose of bootstrapping, that consists of exploiting the *difference* in what is learned. The time-step loop is given below, and implements the acting and learning time modifications required for Bootstrapped Q-Learning:

```
while not done:
    # Select the greedy action of one of the Q-Tables
    qtable = qtables[np.random.randint(N)]
    action = qtable[s, a].argmax()

    # Execute the action and store the experience
    next_state, reward, done, _ = env.step(action)
    experiences.append((state, action, reward, next_state, done))

    # Update every Q-Table with 10 experiences
    for qtable in qtables:
        for index in np.random.choice(len(experiences), 10)
            s, a, r, sn, d = experiences[index]

            if d:
```

```
                y = r
            else:
                y = r + 0.99 * qtable[sn].max()

            qtable[s, a] += 0.1 * (y - qtable[s, a])

    # Move to the next time-step
    state = next_state
```

The above code updates every Q-Table on 10 experiences, every time-step. While this works well in many settings, updating all the Q-Tables on so many experiences, so often, may in some tasks result in all the Q-Tables learning almost the same values, in almost every state. This defeats the purpose of bootstrapping, that relies on the Q-Tables to learn slightly different Q-Values. In any particular setting where Bootstrapped Tabular Q-Learning seems not to explore enough, we recommend updating only one Q-Table (selected at random) per time-step, or lowering the number of experiences replayed per time-step (10 in this case).

The code above implements Tabular Q-Learning, extended with Experience Replay and Bootstrapped Q-Tables. This is a powerful algorithm, that achieves high sample-efficiency on our virtual bartender domain. However, this algorithm is limited by its requirement for discrete (integer) states, while most real-world tasks produce continuous (floating-point) states. We now introduce neural networks, then explain how they are used in Reinforcement Learning. Neural networks greatly extend the range of algorithms that can be designed, and allow to go beyond Q-Learning variations (see Sections 2.7 and 2.8 for example).

2.5 Neural Networks

Many computer scientists, developers, businessmen and technology experts are fascinated by neural networks. The word appears regularly in the press, and is surrounded by a halo of mystery. To most of the world, the neural network is the secret sauce, the magic ingredient that makes AI *happen*.

This section explains what neural networks are, details how they work and manage to learn, and discusses several of their properties that are highly relevant to Reinforcement Learning applications. This last point is particularly important, even for readers already familiar with neural networks, as Section 2.7 will introduce

the Policy Gradients reinforcement learning algorithm, that interacts with core components of neural networks.

2.5.1 Parametric Functions

Neural networks are a form of parametric functions, inspired by how the animal brain is structured, but only from a distance. At the core, a neural network is just a large set of multiplications and additions, that learns to compute outputs using relatively simple mathematical equations.

A parametric function is a function that takes an input and a parameter, and computes an output from the input and parameter. In the literature, the parameter is usually called θ, and the input is x. The output of the function is $\hat{y}$.

$$\hat{y} = f(x, \theta)$$

The function itself can be any computation, of any complexity. The input and parameters, in a practical setting, can be of any data type. For the particular example of a neural network that recognizes faces, the input x is an image (a set of color pixels), the output is an integer (the identifier of the person in the image), and the parameter is a large set of floating-point values that are involved in the computation of the output.

Parametric functions are often used to solve the *supervised learning* problem: we want to approximate the parameter θ so that the output of $f(x, \theta)$ is as close as possible to the *expected output* y. For instance, we give the function a large set of pictures, along with the identifier of the people in the pictures, and we want to find the parameter that allows to correctly predict the identifier of people in new pictures of these people. More formally, given a large set of (x, y) tuples, we want to find the θ parameter that minimizes the error between $\hat{y} = f(x, \theta)$ and y:

$$\theta = \text{argmin}_\theta \sum_{(x,y)} (y - f(x, \theta))^2 \tag{2.3}$$

In the above equation, the only unknown is θ, which means that the optimal θ we are looking for can be computed using the x and y values provided to the algorithm. Moreover, Equation 2.3 represents a simple minimization problem, for which many solutions exist in the literature, such as Genetic Algorithms [Sexton et al., 1998], Ant Colony optimization [Socha and Blum, 2007], or *gradient*

descent, reviewed and explained in a recent book by Kochenderfer and Wheeler [2019]. Gradient descent is often used in neural networks, as it performs well and is compute efficient.

2.5.2 Gradient Descent

Gradient Descent is a simple algorithm that allows to minimize a function. In the case of neural networks, it allows to minimize the summation in Equation 2.3, which produces the θ parameter we are looking for. Gradient descent builds on derivatives. For functions of a single input, such as $y = f(x)$, the derivative $y' = f'(x)$ is a function itself, that gives the slope of f at the point x. This means that if $f'(x)$ is positive, f is increasing at point x. If $f'(x)$ is negative, f is decreasing. Another way of seeing this is that if we want to make $f(x)$ as high as possible, and $f'(x)$ is positive, we should increase x a bit in order to obtain a larger value of $f(x)$. Conversely, if $f'(x)$ is negative, we should decrease x a bit to make $f(x)$ increase. To summarize, the derivative tells us *in which direction to change x so that $f(x)$ increases*.

From the simple derivative described above, only two steps are required to obtain gradient descent. First, the derivative gives the direction in which $f(x)$ increases. When solving a minimization problem, we want to go in the opposite direction. We therefore simply follow the direction given by $-f'(x)$. Second, the derivative is only defined in accordance to a single variable. For a function that takes several inputs, for instance $f(x, \theta)$, especially if θ is a large list of floating-point values, a single derivative is not enough. The *gradient* of a function is the set of all its *partial derivatives*, and is produced by deriving the function several times, one time per variable we are interested in. The minimization problem can then be solved by independently moving each parameter in the opposite direction of its partial derivative.

For instance, let's consider the parametric function $f(x, \theta_1, \theta_2)$ that takes two parameters. Its gradient ∇_θ according to its parameters is the set $(\frac{\partial}{\partial \theta_1} f(x, \theta_1, \theta_2), \frac{\partial}{\partial \theta_2} f(x, \theta_1, \theta_2))$ of its two partial derivatives according to its two parameters θ_1 and θ_2. How to compute these derivatives depends on the exact form of f and is not relevant to this section. The only important aspects of this discussion are that:

- It is possible to compute the partial derivatives of a function in an automatic way.
- The partial derivatives allow to produce the gradient of the function.

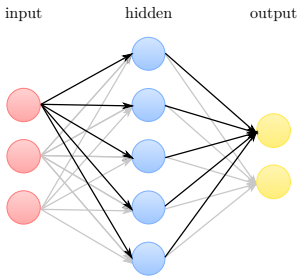

Figure 2.3: A multi-layer feed-forward neural network with 6 inputs, 5 intermediate neurons and 2 output neurons. The arrows indicate connections between neurons. Each connection has its own weight, a floating-point parameter.

- The gradient of the function can be used to algorithmically find the set of parameters θ that minimize the error between the predicted outputs $\hat{y}$ of the function, and the expected outputs y that we want the function to learn.

With the intuition behind gradient descent presented, we now describe the shape and operations performed by f when it is a neural network.

2.5.3 Multi-Layer Feed-Forward Networks

A neural network is a parametric function that takes a large quantity of parameters. The operations performed by the function are inspired by biological networks of neurons in the animal brain [Rosenblatt, 1958]. Basically, Rosenblatt [1958] observed that biological neurons take input stimuli from many neighboring neurons. Each incoming connection has a weight, that defines how sensitive the neuron is to that particular connection. The inputs are added together, and if the resulting sum is high enough, the single output of the neuron fires (and stimulates other neurons to which it connects). This simple description of a biological neuron leads to the definition of an *artificial neuron* that takes many inputs, and computes a single output according to the following formula:

$$o = \sigma(\sum_j w_j i_j)$$

with i_j the j-th input, w_j the j-th *weight* (a parameter that will be learned), and σ the sigmoid function.[2]

Rosenblatt [1958] proposed to organize neurons in *layers*. All the neurons of a layer take the same inputs, but use different weights. This means that they all compute a different function of the same inputs (what changes is their parameters, the weights). Layers can be nested: the outputs of all the neurons of a layer can become the inputs of all the neurons of the following layer. Figure 2.3 depicts such a multi-layer feed-forward neural network.

To summarize, a neural network is a simple collection of artificial neurons organized in layers, as shown in Figure 2.3. Each neuron computes a simple formula based on floating-point inputs, and floating-point weights. The collection of all the weights used by all the neurons of the network form its *parameters*. A neural network is therefore a well-defined computable function that has many floating-point parameters. These parameters can be optimized with gradient descent so that the network computes outputs close to the expected outputs given to it. As such, nothing in a neural network is magic. In the next section, we show how easy it is nowadays to use a neural network in Python.

2.5.4 Practical Implementation of a Neural Network

The previous sections provided a general intuition on how neural networks work, to de-mystify them. The formulas given above, and the fact that gradients have to be computed, may look as if using neural networks would require extensive knowledge about their intricacies. However, neural networks have become so widely used nowadays that a variety of libraries, in many programming languages, implement them. The developer simply has to tell the library how many layers and neurons are required, and to provide (x, y) tuples, and everything else happens automatically.

In this section, we explain how to implement neural networks with the PyTorch[3] Python library. Several other libraries exist, but PyTorch is slightly faster for the particular neural networks we use in Chapter 3, and generally easier to apply to Reinforcement Learning tasks. We therefore implemented all our experiments with PyTorch, and the code attached to this thesis uses PyTorch.

[2] In modern implementations of neural networks, the sigmoid function is replaced by the hyperbolic tangent.

[3] https://pytorch.org

Creating the Network

Before training and using a neural network, we have to create three objects: the network itself, that represents the parametric function and its weights; the optimizer, that specifies the variant of Gradient Descent that will be used to learn the weights; and the loss, in this case the prediction error, that defines the task being solved by the network. The network itself is created layer by layer. The first layer takes as input the input data fed to the network, and the following layers take as input the output of the previous layer:

```
model = torch.nn.Sequential(
        torch.nn.Linear(2, 128),
        torch.nn.Tanh(),
        torch.nn.Linear(128, 4)
)
optim = torch.optim.Adam(model.parameters(), lr=0.001)
loss = torch.nn.MSELoss()
```

The code above creates a simple neural network that takes inputs consisting of 2 floats, has a single hidden layer of 128 neurons with the tanh activation function, and produces outputs consisting of 4 floats. Interestingly, most neural network libraries, PyTorch included, define what happens *between* the layers. In the example above, there is no explicit representation of a hidden layer of 128 neurons. There is one `Linear` connection between the input and the hidden layer, and a second `Linear` connection between the hidden layer and the output. The last two lines of the code above create an optimizer, that uses the state-of-the-art Adam algorithm [Kingma and Ba, 2014], and the Mean Squared Error loss function presented in Equation 2.3. The `lr` parameter of the optimizer is its *learning rate*. The smaller it is, the more stable the neural network tends to be, but the slower it learns. Learning rates between 0.01 and 0.001 are common.

Acquiring the Data

The neural network is now ready to be trained, but we need to provide it with input-output tuples (x, y) of data. Obtaining this data is the most complicated part of working with neural networks. For real-world classification tasks, such as recognizing dog breeds, actual people have to take actual photos of a wide range of dogs, and then annotate every one of them with what breed of dog it is. In Reinforcement Learning, there is fortunately no manual data collection to perform.

The data that will be used to train the network is generated by the Reinforcement Learning agent itself, by interacting with the environment, in the form of states and Q-Values stored in NumPy arrays. For a brief introduction of NumPy, see Section 2.3.1. The code below shows how to create and represent 3 (x, y) pairs, with x a vector of 2 floats and y a vector of 4 floats, suitable for training the neural network presented in the previous section.

```
x_numpy = np.array([
        [1.0, 1.0],
        [0.5, -1.0],
        [2.0, 0.0]
])
y_numpy = np.array([
        [0.0, 1.0, 0.0, 0.0],
        [1.0, 0.0, 0.0, 0.0],
        [0.0, 0.0, 0.0, 1.0]
])
x_pytorch = torch.from_numpy(x_numpy)
y_pytorch = torch.from_numpy(y_numpy)
```

The floating-point values stored in the arrays can be anything that fits the problem to be solved. Neural networks make no assumption about the magnitude or sign of the floats, how many of them are zeros, etc. They can map sensor reading in the thousands to predicted negative temperatures without any problem.

Training the Network

Once input (x) and expected output (y) tuples are available, the neural network can easily be trained. Training a neural network is an iterative process. Each *epoch* consists of presenting the input x to the network, getting its predicted output $\hat{y}$, computing the difference between the predicted output and the actual output y we want it to output, and then performing one gradient step that slightly improves the weights of the network. Because each epoch only slightly changes the weights of the network (by design, rapid changes may cause the network to oscillate and never find the best weights), many epochs are required in order for the network to learn the correct mapping:

```
# Train for 100 epochs
for epoch_number in range(100):
```

```
    optim.zero_grad()
    y_predicted = model(x_pytorch)
    error = loss(y_predicted, y_pytorch)
    error.backward() # This automatically computes the gradient of the network
    optim.step()       # And updates the weights of "model" with Gradient Descent
```

Predicting new Outputs

After a number of epochs, the network has learned a function that maps the inputs (x_pytorch) to the expected outputs (y_pytorch). We can now give the network new inputs, that may be distinct from the inputs that were used for training, and ask the network for predictions. The network will automatically interpolate, or *generalize*, to these new inputs:

```
x_new = np.array([
    [1.1, 1.1],
])
x_new_pytorch = torch.from_numpy(x_new)
y_predicted = model(x_new_pytorch)
```

Will the predicted outputs be *correct*? In a Reinforcement Learning setting, we know that the neural network produces *correct* outputs when the agent learns to solve the task. However, there is no general way to know whether outputs corresponding to previously-unseen inputs are correct. The art of training and evaluating a neural network is a dark and fuzzy one. Fortunately, there are several books on Supervised Learning with neural networks [Loy, 2019; Aggarwal, 2018; Rashid, 2016, the first one being a Python tutorial], that provide an in-depth explanation on how to tune and evaluate neural networks.

2.6 Q-Learning with Neural Networks

The Tabular Q-Learning presented in Section 2.3 is limited to discrete states, that can be represented with a finite number of integer numbers. In most real-world tasks, the state either consists of a vector of floating-point values (that can take a virtually infinite amount of values), or can be represented with integers, but requires so many that maintaining a table of Q-Values would be impractical.

In this section, we discuss how the Q-Table of Tabular Q-Learning can be replaced with a neural network. The neural network observes states, that can be

vectors of floating-point values, and predicts Q-Values. Because neural networks are able to generalize from seen inputs to closeby previously-unseed inputs, environments that never produce the same state twice become possible to learn in. Stochastic environments (with randomness), environments with noisy sensors or that observe complex physical processes are examples of instances where observing the same state twice in the life of the agent is highly unlikely.

2.6.1 Catastrophic Forgetting

When training a neural network, pairs of input and expected outputs are presented to it. The network uses these pairs to optimize its weights so that its predicted output is as close as possible to the expected outputs. This means that even if presented with *a few* input-output pairs, the network will update *all of its weights*. This leads to a phenomenon called Catastrophic Forgetting [French, 1999], that causes the neural network to become good at predicting outputs for inputs it has recently seen, and bad for older ones.

Catastrophic Forgetting is of prime importance in Reinforcement Learning. In Supervised Learning, the whole dataset is typically available to the agent from the start. By presenting many input-output pairs to the network at once, and carefully controlling how and when gradient descent steps are taken, it is relatively easy to ensure that the network learns to be good at the whole task. In Reinforcement Learning, data comes in the form of *experiences*, progressively collected by the agent as it interacts with the environment. If the environment is large, different parts of it may be explored at different times by the agent. Catastrophic Forgetting then becomes relevant: we must prevent the network from losing accuracy in Q-Values of seldom-visited states or states not visited for some time.

One of the first uses of neural networks in a Reinforcement Learning setting is Fitted Neural Q Iteration [Riedmiller, 2005]. NFQ uses a neural network to predict Q-Values. The neural network observes the state-action pairs (a vector of floats concatenated with an integer, or the one-hot encoding of an integer, for instance), and predicts a single floating-point output, the Q-Value $Q(s, a)$ of the state-action pair. When learning, NFQ uses all the experiences from an experience buffer, along with the Q-Learning equation (2.2), to produce target Q-Values y for every state-action pair in the experiences ever seen by the agent. Then, NFQ trains the network on these input (state-action) expected outputs (y) pairs. Because the network always sees all the experiences ever observed by the agent, Riedmiller [2005] point out that the neural network can be aggressively

trained, until it becomes maximally accurate for these experiences. This leads to highly sample-efficient learning, but with two drawbacks:

1. Training a neural network takes an amount of time linear in the amount of input-output pairs presented to it. Constantly re-training the network on all the experiences in the buffer is therefore slow;

2. Worse, the experience buffer grows after each time-step. The training procedure therefore becomes slower and slower as the agent interacts with the environment.

We would also like to mention slightly earlier work, Least-Square Policy Iteration [Lagoudakis and Parr, 2001], that uses linear function approximation instead of neural networks. The idea is the same as NFQ, though: constantly re-use all the past experiences to update the Q-Function. The two drawbacks enumerated above still apply, and we discuss how they have been addressed in more recent work in the next section.

2.6.2 The DQN Algorithm

The DQN algorithm, for Deep Q Networks, made the news in 2015 [Mnih et al., 2015]. They propose a reinforcement learning algorithm that is able to represent its Q-Function with a neural network, while avoiding catastrophic forgetting and maintaining constant-time network training. Mnih et al. [2015] then proceed to apply this reinforcement learning agent to Atari games, video games from the eighties that require the agent to process *images*, a feat well beyond the reach of Reinforcement Learning before.

The core contribution of DQN is the *target network*. The idea is to maintain *two* Q-Functions, Q and the target network Q'. Q is updated with experiences, while Q' slowly follows Q (or is kept fixed, and only set to Q every thousand time-steps, in modern implementations of DQN). The reason why Q' is important can be seen in the equations used to update Q, a slight variation of the Q-Learning equation (2.2):

$$\begin{aligned} y &= r_t + \gamma \max_{a'} \overbrace{Q'_k}^{\text{target network at iteration } k}(s_{t+1}, a') \\ Q_{k+1}(s_t, a_t) &\leftarrow Q_k(s_t, a_t) + \alpha(y - Q_k(s_t, a_t)) \end{aligned} \tag{2.4}$$

In the above equations, the target network produces the Q-Values of the next state, also sometimes called *target values*, hence the name *target network*. Predicting the target values with a dedicated network, that does not change too fast, increases the stability of the agent. Errors made by the neural network Q do not propagate to neighboring states anymore, as Q is not the network used to compute next-state values. This increase in stability also enabled the second contribution of DQN: every time-step, only a relatively small *subset of experiences* are used for training. In the original DQN paper, 32 experiences are sampled every time-step. This allows the agent to learn long and complicated tasks, without becoming slower and slower at learning as with Neural Fitted Q Iteration. However, the DQN algorithm introduces another drawback:

1. Every time-step, 32 experiences are used to compute 32 expected outputs y with Equation 2.4, then the neural network Q performs *only one* gradient step based on these expected outputs.

Performing only one gradient step, in addition to the use of the target network, is required to prevent catastrophic forgetting. Unfortunately, it also dramatically decreases the sample-efficiency of DQN compared to Neural Fitted Q Iteration, that performs many gradient steps per time-step, and as such learns much more accurate Q-Values more quickly. In Chapter 3, we introduce our first contribution, a Reinforcement Learning algorithm that works with neural networks, has all the nice properties of DQN, but does not require target networks, and allows *several* gradient steps to be performed per training iteration.

2.6.3 Extensions of DQN

The DQN algorithm received considerable attention the last few years, and has given birth to a whole family of DQN-like Reinforcement Learning algorithms. Dueling DQN slightly changes the shape of the neural network used by DQN to force it to learn separate *state values* and *advantage values*, using the definition $Q(s,a) \equiv V(s) + A(s,a)$ [Wang et al., 2016b]. This increases the stability of the agent by constraining the values of the actions to have something in common, the value $V(s)$ of the state. Distributional DQN, discussed in Section 2.4.2, does not learn simple Q-Values, but a description of the probabilistic distribution of the Q-Values.

Double DQN [Van Hasselt et al., 2016] replaces the Q and Q' networks with Q^A and Q^B. Q^A uses Q^B as target network, and Q^B uses Q^A as target network. This

maintains the stability of using Q' target networks, without the need to slow down a fundamental part of the agent. Averaged DQN [Anschel et al., 2017] maintains a history of past Q networks. After every update of Q on a batch of experiences (so, usually every time-step), a new Q_k (with k increasing) network is produced. When training Q_k, the average predictions of a number of $Q_{k-1}...Q_{k-N}$ networks is used. This is supposed to mimic the behavior of having a target network, without needing a slow Q' target network, but our personal experiments were unable to validate that claim. Instead, Clipped DQN [Fujimoto et al., 2018] goes back to Double DQN, learns Q^A and Q^B, but uses $\min(Q^A, Q^B)$ as target values for both networks (while Double DQN uses Q^B as the target of Q^A, and Q^A as target for Q^B). Fujimoto et al. [2018] explain that the minimum compensates for the over-estimation errors that neural networks introduce in DQN. Our Bootstrapped Dual Policy Iteration algorithm, presented in Chapter 3, uses Clipped DQN. This variant of DQN, while being simple, led to the best results in our experiments.

Other DQN variants modify other components of DQN. For instance, Prioritized Experience Replay [Schaul et al., 2015] changes how experiences are sampled for learning: instead of uniformly randomly selecting experiences, the experiences having the largest absolute TD-error are more likely to be selected. The absolute TD-error is the absolute value of the difference between $Q(s, a)$ and its updated value, y in Equation 2.4. Because so many variants of DQN exist, the Rainbow algorithm explores their combinations, and shows that using all the bells and whistles of DQN at once leads to very good results [Hessel et al., 2017].

2.6.4 Implementing DQN

In Section 3.4, we provide step-by-step details about how to implement a Reinforcement Learning algorithm that uses neural networks. As such, we do not repeat these instructions here, and refer the interested reader to Section 3.4. In the next sections of this chapter, we review other Reinforcement Learning algorithms, that learn more than just Q-Values, or no Q-Values at all. We provide implementation details for the most important of these algorithms.

2.7 Policy Gradient Methods

In the previous sections, we present Reinforcement Learning algorithms that learn Q-Values, how promising every action is in every state. This family of algorithms

is often referred to as *value-based*, or, somewhat less often but highly relevant to this thesis, as *critic-only*. In the Reinforcement Learning literature, a *critic* is any component that learns Q-Values, because, as in arts, a critic evaluates *how good* an action is.

Another family of Reinforcement Learning algorithms exists. They do not learn Q-Values, but instead directly learn a *policy*, or *actor*, a function that receives a state as input, and directly produces as output the action to be executed by the agent. The advantage of these algorithms is that they are quite simple and make few assumptions. In Chapter 4, we detail why *actor-based* algorithms perform well in environments where the agent has to remember information, instead of purely reacting to the current state. Actor-based algorithms are also able to produce *continuous actions*, as we discuss in Section 2.7.4.[4] However, actor-based methods tend to have poor sample-efficiency, as discussed in Chapter 3, which makes them difficult to apply to real-world settings, compared to value-based methods.

2.7.1 Formalism and Notations

The Policy Gradient algorithm is easy to implement, but more difficult to understand than value-based methods. This comes from its reliance on a number of mathematical concepts, and notations highly specific to the algorithm.

Policy Gradient assumes a *parametric policy* $\pi(s, \theta) \in \mathbb{R}^{|A|}$. π is the policy, s is a state given to the policy and for which we want to obtain an action, θ is the set of parameters of the policy, as defined in Section 2.5, A is the set of actions, $|A|$ is the number of actions available to the agent, and $\mathbb{R}^{|A|}$ represents the set of vectors of real values that have one real value per action. If the agent has access to 4 actions, the policy outputs 4 real values. We usually also consider that the policy produces real values that are all non-negative, and sum to 1. This leads to the actor producing a *discrete probability distribution* over the actions,[5] such as the a-th output of the actor is the probability with which it wants the a-th action to be executed by the agent. We refer to the a-th output of the policy as $\pi(a|s, \theta)$.

Before introducing the Policy Gradient algorithm, we also define the return R_t at time t as the discounted sum of all the rewards obtained by the agent, starting at time t, and until the end of the episode:

[4] Some versions of Q-Learning, such as Fuzzy Q-Learning [Glorennec, 1994], also allow continuous actions.

[5] We discuss continuous actions in Section 2.7.4.

$$R_t \equiv \sum_{k=t}^{T} \gamma^{k-t} r_k \tag{2.5}$$

with T the duration (in time-steps) of the episode, and $\gamma < 1$ the discount factor defined in Section 2.3. Equation 2.5 shown above is comparable to the mathematical definition of a Q-Value, shown in Equation 2.1 on page 22, with one significant difference: the return R_t is a simple sum of what actually happened in the environment, and is computed directly from experiences. It contains no expectation in its definition. A Q-Value $Q(s, a)$ has an expectation in its definition, and can intuitively be seen as measuring, *on average*, after many executions of the policy, what would be the mean return obtained by the agent. This distinction between a return that is a simple sum of what happened, and a Q-Value defined with an expectation, sometimes leads the return being called the *Monte-Carlo return*, as a reference to the Monte-Carlo method used in physics to compute values from the results of a large number of simulations [Eckhardt et al., 1987], instead of mathematical equations.

2.7.2 Intuition Behind Gradient-Following Algorithms

The first well-known description of an actor-based algorithm, that does not learn any Q-Value but still manages to learn how to perform a task, is the REINFORCE family of algorithms [Williams, 1992]. A REINFORCE algorithm is any algorithm that considers a parametric policy $\pi(s, \theta) \in \mathbb{R}^{|A|}$, and progressively adapts its parameter according to a small learning rate, the return received by the agent in state s, and the log-probability $\log(\pi(a|s, \theta))$ according to which the action a was taken in state s.

Intuitively, the parameter is adjusted so that good actions are executed more often, and bad actions less often. More precisely, the inner working of the algorithm, quite difficult to grasp by only looking at the formulas, consists of increasing the probability of every action in every state, with a strength that depends on the return obtained by the agent when executing that action in that state. This means that if, in a particular state s, the action a_1 consistently leads to higher returns than a_2, the probability of a_1 will be pulled upwards with more force than that of a_2. Because the actor produces probabilities, that sum to 1, any increase of a_1 automatically causes a decrease of a_2. There are therefore two processes that compete: the probability of every action is pulled upwards, some stronger and

some weaker, and any increase of probability of one action is at the expense of the other actions. As such, the action leading to the highest return out-competes the other ones, which leads to our intuition that *good actions increase in probability, bad ones decrease in probability.*

While Policy Gradient implements this intuition using a parametric policy whose weights are changed over time, another class of algorithms, the Learning Automata [Narendra and Thathachar, 1989], directly compute how the probabilities output by the policy should change. We discuss Learning Automata algorithms in Section 2.9.

2.7.3 The Policy Gradient Algorithm

While the original REINFORCE family of algorithms is quite broad, and only described theoretically, Sutton et al. [2000] extend it and introduce the Policy Gradient Reinforcement Learning algorithm. At acting time, the state s_t is given to the policy $\pi(s_t, \theta)$, that produces a probability distribution over actions. The action $a_t \sim \pi(s_t, \theta)$ is sampled according to these probabilities, executed in the environment, and produces a reward r_t. The resulting (s_t, a_t, r_t) experience tuple is added to a list.

At the end of the episode, the policy can be updated. Every experience in the experience list is used to compute (s_t, a_t, R_t) tuples, with R_t the return at time t as defined in Equation 2.5. Then, the following *loss* is computed:

$$L_\pi = -\sum_{(s,a,R)} R \log(\pi(a|s,\theta)) \tag{2.6}$$

Most of the time, the policy π is represented by a neural network. It is therefore possible to pass the loss to the neural network, as detailed in Section 2.5.1 (by replacing Equation 2.3 with Equation 2.6), and the neural network will adjust its parameter θ so that the loss is minimized. By minimizing Equation 2.6 (notice the minus sign in the equation), the neural network maximizes $\sum_{(s,a,R)} R \log(\pi(a|s,\theta))$, which matches the description of the algorithm given in the previous section: the probability of every action wants to increase, with a strength that depends on how large the return obtained by that action is.

When implementing Policy Gradient, a few points require attention to ensure convergence:

Single gradient step

The loss is built using all the experiences in the experience list, then its

gradient according to the parameter θ is computed, then θ is adjusted in the direction opposite of the gradient with a *very small learning rate* (0.001 or less) and for *only one gradient step*. Higher learning rates, or more gradient steps, would cause the policy to change too quickly, risking over-shooting and instability.

Discarding experiences

After the parameter has been updated, all the experiences that have been used to compute the loss must be discarded, and a new episode must be executed. This is because Policy Gradient is **strongly on-policy**: the experiences used to compute the returns R_t must have been obtained by following the policy. If experiences are reused across training epochs, they would make the policy move according to outdated information, which causes over-shooting and instability. The *on-policyness* of Policy Gradients makes it incompatible with Experience Replay, a big drawback compared to Q-Learning algorithms when considering sample-efficiency.

Strict on-policy sampling

The actions executed by the agent in the environment must be sampled exactly from $\pi(s, \theta)$. The probabilities output by the policy cannot be modified in any way. Policy Gradient is extremely sensitive to this, and will diverge if any probability is increased, decreased or set to zero after being produced by the policy. In Chapters 4 and 5, **we introduce mitigations to this sensitivity**, that allow to constrain the actions taken by a Policy Gradient agent without harming its convergence.

2.7.4 Policy Gradient with Continuous Actions

The main advantage of Policy Gradient is that it allows the use of continuous actions, actions that are vectors of real values instead of integers. More precisely, Policy Gradient does not make any assumption that prevents continuous actions, while Q-Learning, by requiring a max and an argmax over actions when learning and acting, makes continuous actions intractable.[6]

With discrete actions, the policy outputs a fully-defined *discrete probability distribution*, from which it is easy to sample an action (see Section 2.7.6 for an example). In Equation 2.6, $\pi(a|s, \theta)$ is simply the a-th output of the policy. With

[6] Computing $\max_a Q(s, a)$ with a a real value would require evaluating an infinite amount of actions, or making assumptions about how the Q-Values depend on the action.

continuous actions, it is impossible to output a probability for every possible action, as there is an infinite amount of them. Instead, we must define the general probability *distribution* that the actions will follow, such as a Gaussian probability distribution. Because a Gaussian is defined by its mean and variance, the policy must output two vectors of real numbers, of the same dimension as the continuous action: one for the mean, and one for the variance. From these two vectors, equations allow to sample an action, and to compute what was the probability of this action being sampled. This allows actions to be selected, and Equation 2.6 to be computed. One last challenge is computing the gradient of Equation 2.6, as required by a neural network when learning, when continuous actions are sampled. This is outside the scope of this thesis, but we refer the interested reader to implementations of the PPO algorithm [Schulman et al., 2017] for more details.

2.7.5 Variants of Policy Gradient

Since its introduction by Sutton et al. [2000], Policy Gradient has been built upon several times, to produce algorithms with higher stability, compute efficiency or sample efficiency.

The Asynchronous Advantage Actor Critic [Mnih et al., 2016, A3C] allows several workers to produce experiences in several replicas of the environment, such as several instances of a video game. All these experiences are used for learning in a heavily distributed way. A3C is well-suited to simulated tasks, where compute time is the bottleneck. However, A3C does not improve the sample-efficiency of Policy Gradient, and instead focuses on producing and processing experiences as quickly as possible. This approach is only possible in simulators, as there is no way to make the physical world (robots for instance) move faster, to produces experiences faster.

Two other algorithms, both by mostly the same set of authors, improve the stability and sample-efficiency of Policy Gradient. Trust Region Policy Optimization [Schulman et al., 2015, TRPO] introduces the concept of a *trust region*, the maximum change in action probabilities that is deemed acceptable when updating the policy, and enforces it with linear search. Intuitively, changing the parameters of a policy may change the probabilities it outputs in some states by a large amount, as some parameters of a neural network get multiplied by large values when the output is computed. TRPO forces those *sensitive* parameters to be moved more slowly, to avoid large changes in the policy that are detrimental for learning. Proximal Policy Optimization [Schulman et al., 2017, PPO] achieves the

same goal, but replaces the line search with a clever policy loss, so that the neural network constraints itself automatically. This makes PPO simpler to implement, and also allows the policy to be trained for *several gradient steps* per batch of experiences collected, which increases sample efficiency.

Finally, Kakade and Langford [2002] present Natural Policy Gradient, that computes the gradient in a slightly different way. When the policy has a large number of parameters, some of them may have very small partial derivatives, which leads to slow changes in the policy when it is close to a local optimum, or a saddle point. Natural Policy Gradient uses the second derivative of the policy (more precisely, its Hessian matrix) to correct the gradient, and make it converge faster. A more recent algorithm, ACKTR [Wu et al., 2017], replaces the Hessian matrix (that has a number of entries that is the squared number of parameters in the policy) with its much smaller Kronecker-factored form. This leads to a compute-efficient algorithm that is slightly more robust and sample-efficient than PPO, and is as such the current state of the art in Policy Gradient.

The recent improvements to Policy Gradient allow it to learn highly challenging tasks [Wu et al., 2017], [Haarnoja et al., 2018]. However, the sample-efficiency of Policy Gradient still lags behind value-based methods, as even attempts at allowing Experience Replay with Policy Gradient [Wang et al., 2016a] are limited by the need of Policy Gradient to compute *on-policy* returns or Q-Values, as we discuss in Section 2.8.1. One of our main contributions in this thesis, in Chapter 3, is an algorithm that has many of the positive properties of Policy Gradient, while being as sample-efficient as (if not more) Q-Learning-based methods.

2.7.6 Implementing Policy Gradient

As long as the three points of attention described in Section 2.7.3 are met, implementing Policy Gradient is perfectly possible using off-the-shelf neural network libraries. We start with the policy, a neural network that takes as input the state, and outputs a probability distribution over actions:

```
policy = torch.nn.Sequential(
    torch.nn.Linear(4, 128),
    torch.nn.Tanh(),
    torch.nn.Linear(128, 8),
    torch.nn.Softmax(-1)        # Specific to Policy Gradient
```

```
)
optim = torch.optim.Adam(policy.parameters(), lr=0.001)
```

The neural network described above looks like the one of Section 2.5.4, but introduces an extra layer, Softmax. This layer ensures that the output of the network is a vector of floats that sum to 1, and are all positive and lower than 1 (so, they form a probability distribution). The optimizer we use is Adam [Kingma and Ba, 2014], as usual.

The acting part of Policy Gradient is quite simple, as the policy directly gives the probability with which the actions should be sampled. This removes the need for any kind of artificial exploration strategy, such as ε-Greedy:

```
experiences = []

while True:
    while not done:
        # Sample an action from the policy
        probas_pytorch = policy(torch.from_numpy(np.array([state])))
        probas = probas_pytorch.detach().numpy()[0]
        action = np.random.choice(range(env.num_actions), p=probas)

        # Execute the action and store the experience
        next_state, reward, done, _ = env.step(action)
        experiences.append((state, action, reward))

        # Move to the next time-step
        state = next_state
```

Sampling the action requires a bit of boilerplate code. The state has to be cast to a PyTorch tensor, given to the policy. The probabilities returned by the policy are a PyTorch tensor, that has to be cast to a Numpy array. Because PyTorch neural networks take lists of inputs and predict lists of outputs (one output per input), the code above has to make a one-element list that contains the current state, and extract the first (and only) element of the PyTorch result in probas. Finally, sampling the action is done by np.random.choice. It is important that nothing happens to probas between its prediction by the network and the sampling. No probability can be changed, set to zero or swapped with another one, as it would lead to the complete collapse of Policy Gradient (see Section 2.7.3).

Training the policy happens after every episode, using the experiences collected. First, the returns of every experience has to be computed. Then, the loss given in Equation 2.6 is minimized, leading to an improved policy:

```
while True:
    # [...]
    while not done:
        # [...] (see code above)

    # Compute returns, starting from the last experience
    states = []
    returns = []
    cumulative_ret = 0.0

    for state, action, reward in reversed(experiences):
        cumulative_ret += reward
        ret_mask = np.zeros((env.num_actions,))
        ret_mask[action] = cumulative_ret

        states.append(state)
        returns.append(ret_mask)

    # Clear the experiences, very important
    experiences.clear()

    # Train the policy
    states_pytorch = torch.from_numpy(np.array(states))
    returns_pytorch = torch.from_numpy(np.array(returns))

    optim.zero_grad()
    probas = policy(states_pytorch)
    loss = -torch.sum(returns_pytorch * torch.log(probas)) # Equation 2.6
    loss.backward()
    optim.step()
```

In the example above, we cheat a little bit with the returns. The idea is to produce a *vector* of returns, that contains zeros for every action except for the action that was taken. Then, when the loss is computed, the sum over every

state and action contains zeros for any action that was not taken, and so only takes into account the actions that have been actually executed. This is a simple way to implement the Policy Gradient loss, and concludes the implementation of Policy Gradient. The simple implementation we present here works well, and is able to learn *LunarLander-v2* in about 5000 episodes (in Chapter 3, we show that our BDPI algorithm learns that environment in 200 episodes, illustrating the sample-inefficiency of Policy Gradient).

2.8 Actor-Critic Algorithms

The Policy Gradient loss of Equation 2.6 relies on the Monte-Carlo return obtained by the agent, discussed after Equation 2.5 on page 48. Because the policy changes as the agent learns, and the environment can be stochastic, the Monte-Carlo returns are a high-variance estimate of the expected return to be obtained by executing an action in a state. Konda and Borkar [1999] show that this high variance leads to a poor gradient estimate, and as such slow learning. The proposed solution consists of replacing the Monte-Carlo estimate R with a Q-Value $Q^\pi(s,a)$, leading to the following equation:

$$L_\pi = -\sum_{(s,a)} Q^\pi(s,a) \log(\pi(a|s,\theta))$$

The Q-Values, learned with value-based methods, form a *critic*. Combined with the explicit policy π, or *actor*, this forms an *actor-critic* algorithm. Actor-critic algorithms maintain the positive properties of Policy Gradient, such as the state-specific exploration built into the policy, and add the following properties:

More frequent policy updates
: The removal of the monte-carlo return R means that the agent does not need anymore to wait for an episode to terminate before updating the policy. This allows non-episodic tasks to be learned, or the policy to be updated several times per episode if the episodes are long [Schulman et al., 2017]. Having a critic also generally increases the flexibility of the agent, and for instance allows to distribute the learning of the actor and the critic, in an environment-agnostic way [Mnih et al., 2016].

	ER	Cont.	Smart expl.	Sample-Efficiency	A.	C.
Q-Learning	✓	✗	✗	good	✗	✓
Policy Gradient	✗	✓	✓	very bad	✓	✗
Actor-Critic	✓	✓	✓	bad	✓	✓
BDPI (ours)	✓	✗	✓	excellent	✓	many

Figure 2.4: High-level comparison of the Q-Learning family of algorithms, Policy Gradient, Actor-Critic and the BDPI algorithm we introduce in Chapter 3. ER: Experience Replay possible; Cont.: Continuous actions possible; Smart expl.: state-specific exploration; A./C.: has an actor/critic.

Experience Replay
Related to the point above, experiences can now be stored in an experience buffer, and used for several policy updates [Gruslys et al., 2017; Wang et al., 2016a]. We discuss such actor-critic with experience replay in more details in the next section, as it introduces the challenge of learning an *on-policy* critic from *off-policy* experiences.

Continuous actions
By computing the gradient of $Q^\pi(s, a)$ with regards to the action, it is possible to compute, for continuous actions, how to change the action so that its Q-Value increases. This gradient is at the basis of *deterministic actor-critic* algorithms, discussed in Section 2.8.2.

The main limitation of actor-critic algorithms is that, due to the reliance of the Policy Gradient loss on the *on-policy* return or its estimate, they have to learn an *on-policy critic* Q^π. Before discussing the impact of on-policy critics in the next section, we mention Figure 2.4, that briefly summarizes the main properties of all the Reinforcement Learning algorithms presented in this chapter.

2.8.1 Classification of Actor-Critic Algorithms

Actor-critic algorithms need to learn a critic $Q^\pi(s, a)$ that is accurate for the actor π. As the agent learns, the actor changes, and the critic has to accurately follow it. Pirotta et al. [2013] discuss the importance of accurate on-policy critics in the Conservative Policy Iteration setting, related to actor-critic, that we discuss further in Section 2.9.

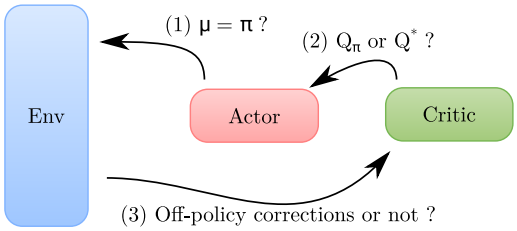

Figure 2.5: In an actor-critic setting, three components of the agent can be implemented in various ways: (1) the behavior of the agent in the environment (μ) may differ from the actions predicted by the actor π; (2) the critic may learn Q-Values that match what the actor does (Q^π), or learn the optimal actor-independent value function Q^*; (3) experience replay may or may not be used, and experiences may or may not be off-policy corrected before being used to train the critic. Such variability in the algorithms cannot be represented by a simple binary classification between *on-policy* and *off-policy*.

We introduce new terminology regarding the on-policyness of actor-critic components. We argue that the current terms of *on-policy* or *off-policy* are not used precisely enough in the literature. A vast amount of papers present "off-policy" actor-critic algorithms [Wang et al., 2016a; Gu et al., 2017a; Haarnoja et al., 2018; Gu et al., 2017b; Degris et al., 2012; Gu et al., 2017c], but they do not have an *off-policy critic*:

1. They allow the behavior of the agent in the environment, $\mu(s) \in A$, to differ from its actor π;
2. But they use *off-policy corrections* [Munos et al., 2016] or importance sampling [Degris et al., 2012] to learn a critic Q^π that is *on-policy with regards to the actor* from samples generated by a policy *that is not the actor*.

In the current actor-critic literature, the critic always learns Q^π (not Q^*), and any algorithm that allows to train the critic using samples not generated by the *latest version* of the actor requires off-policy corrections. ACER and the off-policy actor-critic [Wang et al., 2016a; Degris et al., 2012] use off-policy corrections to learn Q^π from past experiences, DDPG learns its critic with an on-policy SARSA-like algorithm [Lillicrap et al., 2015], Q-prop [Gu et al., 2017b] uses the actor in the critic learning rule to make it on-policy, and PGQL [O'Donoghue et al., 2017]

allows for an off-policy V function, but requires it to be combined with on-policy advantage values.

Figure 2.5 show the three points of variation an actor-critic agent may have. We argue that the proper qualification of actor-critic algorithms requires more terms than just *off-policy*. We introduce the following terminology:

on/off-actor Covers point (2) of Figure 2.5. An agent that is *on-actor*, as most current ones are, learns Q^π. An agent that is *off-actor* learns any other value function, such as Q^*.

on/off-policy As the current terminology defines. An *on-policy* agent executes a behavior policy that is exactly the actor. An *off-policy* agent has a behavior policy that is different from the actor.

Regarding point (3) of Figure 2.5, any on-actor but off-policy agent has to implement off-policy corrections to learn a critic that evaluates the actor.

While almost every actor-critic algorithm is on-actor, notable examples of algorithms without an on-actor critic are AlphaGo Zero [Silver et al., 2017], that replaces the critic with a slow-moving target policy learned with tree search, and the Actor-Mimic [Parisotto et al., 2016]. The Actor-Mimic, presented in a multi-task Reinforcement Learning setting and not as an actor-critic algorithm, learns one critic per task with a variant of Q-Learning. These critics learn Q^* for their task, and do not refer to any actor. Once all the critics are learned, a single actor is trained in a supervised way, to minimizes the cross-entropy between the actor and the Softmax policies of critics. The actor is then shown to be reasonably good at solving all the tasks.

Another example of an off-actor algorithm, that motivates our introduction of the terminology,[7] is our Bootstrapped Dual Policy Iteration (see Chapter 3). BDPI learns critics with a variant of Q-Learning, and regularly trains its actor to approximate the average greedy policy of the critics. We provide a theoretical analysis, and empirical evidence, that BDPI is both off-policy and off-actor.

2.8.2 Deterministic Policy Gradient

The previous sections consider *stochastic* actor-critic algorithms, where the actor outputs a probability distribution over actions, and the loss considers the

[7] About 12 anonymous reviewers for many major conferences mistakenly classified BDPI as already-existing work while citing "off-policy" (actually, off-policy but still on-actor) algorithms.

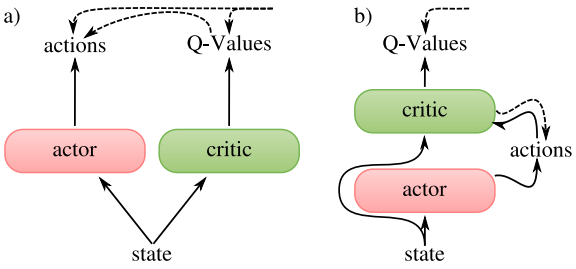

Figure 2.6: Stochastic (left) vs Deterministic (right) actor-critic algorithms. The dotted lines indicate *training feedback*: gradients and returns for actions, rewards for Q-Values. Deterministic actor-critic algorithms use the critic to tell the actor which actions it should perform. Stochastic actor-critic algorithms let the actor improve itself with the help of Q-Values.

$Q(s, a)\pi(a|s, \theta)$ quantity. Deterministic actor-critic algorithms, sometimes called deterministic policy gradient even though they require a critic to work, have an actor that directly outputs an action. The critic is used to compute how the action should change for it to lead to higher returns [Silver et al., 2014]. Figure 2.6 intuitively represents the difference between the two families of algorithms.

In Deterministic Policy Gradient algorithms, the actor parameter θ is updated by minimizing the following loss:

$$L_\pi = -\sum_s \pi(s, \theta) \frac{\partial}{\partial a} Q^\pi(s, a = \pi(s, \theta)) \tag{2.7}$$

In the above equation, we use the same notation as in the stochastic policy gradient equation (2.6). An important point that is emphasized by our notation, and the one of Silver et al. [2014], is that the critic has to be queried for the Q-Values of the action that the actor wants to perform. Practically, the actor is given a state s and produces an action a. The state and action are given to the critic that produces $Q(s, a)$, then the gradient of that value (with regards to a) is taken. Going back to the definition of the gradient, the gradient of a *function* regarding a *parameter* tells in which direction to change the parameter so that the function increases. Therefore, the gradient of $Q(s, a)$ regarding the action a tells in which direction to move a so that the Q-Value increases. By giving this gradient to the actor, the actor moves towards what the critic thinks a better action

is. Bakker [2007] nicely explains the algorithm, presents the figure that inspired our Figure 2.6, and is, to our knowledge, the first description of a deterministic actor-critic algorithm. Lillicrap et al. [2015] also presents a neural-network-based deterministic actor-critic algorithm, and applies it to a variety of continuous-action environments.

2.9 Conservative Policy Iteration

Section 2.8 focuses on actor-critic algorithms that learn their actor with a variant of Policy Gradient. Another family of actor-critic algorithms exist: Approximate Policy Iteration. Instead of Policy Gradient, Approximate Policy Iteration algorithms iteratively update a critic Q^π, then train the actor in a supervised way, so that it approximates $\Gamma(Q^\pi)$, the greedy policy of the critic [Kakade and Langford, 2002]. The greedy policy of a critic outputs, for every state, a probability distribution with a 1 for the action that has the largest Q-Value in the state, and zeros for every other action.

To ensure smoother and more stable learning, Conservative Policy Iteration introduces a policy learning rate α, typically 0.01 to 0.1, so that the actor *slowly pursues* the greedy policy of the critic [Scherrer, 2014]:

$$\pi_{k+1} \leftarrow \begin{array}{ll} \Gamma(Q^{\pi_k}) & \text{(Approximate PI)} \\ (1-\alpha)\pi_k + \alpha\Gamma(Q^{\pi_k}) & \text{(Conservative PI)} \\ (1-\alpha)\pi_k + \alpha\pi'_k & \text{(Dual PI)} \end{array}$$

The above equations do not refer to states and actions. In most of the Conservative Policy Iteration literature, the update equations are assumed to be applied to every state and every action. Safe Policy Iteration [Pirotta et al., 2013] introduces a way for the agent to automatically adjust the policy learning rate to the error in Q-Values. This increases stability in learning, and also paves the way for sampling-based learning (not every state and action, but only some states and actions sampled from an experience buffer). Ensuring monotonic improvement of the policy is a well-researched area, with Thomas et al. [2015] proposing to use statistical tests on the Q-Values to determine in which states to update the actor towards the greedy policy, and in which states to maintain it untouched.

Conservative Policy Iteration algorithms are somewhat difficult to implement, especially when function approximation is used [Wagner, 2011; Böhmer et al., 2016]. This mainly comes from the fact that, much like Policy Gradient-based

algorithms (see Section 2.8.1), CPI algorithms require an *on-actor* (on-policy) critic. Pirotta et al. [2013] show that, to ensure convergence, the actor learning rate has to decrease rapidly as the critic diverges from Q^π. **The core of our main contribution of Chapter 3 is the use of an off-policy critic**, that learns Q^* instead of Q^π. Its greedy function does not match the Conservative Policy Iteration formalism anymore. Instead, we adopt the notations of Dual Policy Iteration (Dual PI in the equations above), that considers an actor trained in a supervised way to pursue *any target policy* π' [Sun et al., 2018]. A popular example of a Dual Policy Iteration algorithm is AlphaGo Zero, that uses a rollout-based planner in a model of the environment to produce target policies [Anthony et al., 2017; Silver et al., 2017].

2.9.1 Pursuit Algorithms

Before ending this introductory chapter and moving to our first contribution, we would like to briefly introduce Pursuit algorithms. While they are superseded by Conservative Policy Iteration algorithms in a Reinforcement Learning setting, the Pursuit literature, that focuses on bandits [Berry and Fristedt, 1985], explores areas of research that may be relevant to reinforcement learning.

Pursuit algorithms maintain a critic $Q(a) \in \mathbb{R}$ and an actor $\pi(a) \in \mathbb{R}$, with the actor producing a probability distribution over actions. At each time-step t, the critic is updated using a simple moving average of the return obtained by action a, and the actor is updated by *pursuing* a set of actions [Narendra and Thathachar, 1989], using Equation 2.8:

$$\pi_{t+1} = (1-\alpha)\pi_t + \alpha \frac{\pi'_{t+1}}{\|\pi'_{t+1}\| + \epsilon}, \tag{2.8}$$

with π_0 a uniform probability distribution, $0 < \alpha < 1$ the policy learning rate, and ϵ positive but very close to zero. The main difference between Conservative Policy Iteration and Pursuit, beyond the fact that Pursuit has no notion of state due to its bandit nature, is that Equation 2.8 uses a vector π'_{k+1} where there may be *several* ones, instead of the greedy function Γ that associates exactly one action to every state. Several *Pursuit schemes* exist, and change how π'_{k+1} is produced:

$$\pi'_{t+1}(a) = \mathbb{1}(a = a^*), \tag{RP}$$
$$\pi'_{t+1}(a) = \mathbb{1}(a = a^* \wedge a = a_t), \tag{RI}$$

$$\pi'_{t+1}(a) = \mathbb{1}(a = a^* \vee Q(a) > Q(a_t)), \tag{GP}$$

with $a^* = \text{argmax}_{a'} Q(a')$ the greedy action, and a_t the action that has just been executed by the agent at time t. Reward-Penalty (RP) pursues the greedy action [Thathachar and Sastry, 1986]. Reward-Inaction (RI) pursues the current action if it is greedy, and does nothing otherwise [Shapiro and Narendra, 1969]. Generalized Pursuit (GP) pursues the greedy action, in addition to every action strictly better than the current one [Agache and Oommen, 2002]. Conservative Policy Iteration implements Reward-Penalty, as only the greedy action is pursued. The other Pursuit schemes rely on a_t in their formulas, which intuitively makes them dependent on the behavior of the agent when learning. We discuss in Chapter 3 why this is undesirable, as we design a fully off-policy algorithm, but other applications of Conservative Policy Iteration may benefit from the use of creative Pursuit schemes.

3 | Bootstrapped Dual Policy Iteration

This chapter is drawn from our publication, Steckelmacher, Plisnier, Roijers and Nowé, *Sample-Efficient Model-Free Reinforcement Learning with Off-Policy Critics*, published in the proceedings of the European Conference on Machine Learning (ECML), 2019.

A simulated robot learns to avoid obstacles in 5 minutes: https://www.youtube.com/watch?v=3jVlRFXafeQ

The universe has a finite lifetime, and so do people and robots. In the previous chapter, we reviewed a variety of Reinforcement Learning algorithms, that can be used to learn policies for a wide range of tasks. However, until now, learning *in a decent amount of time* has been a secondary objective at best.

When we discuss learning speed, we refer to *sample-efficiency*, how efficient a Reinforcement Learning algorithm is at learning a good policy with few interactions with the environment. Most current algorithms are optimized for tasks in which sample-efficiency is irrelevant: the environment can be simulated at thousands of time-steps per second [Mnih et al., 2016; Silver et al., 2017; Schulman et al., 2017], and researchers optimize how to simulate the environment faster and faster, to feed billions of experiences to the agent [Wijmans et al., 2019]. However,

many real-world tasks are unable to execute that many time-steps, for at least one of these two reasons:

1. The time-step takes physical time, such as a robot having to actually move in the real world. It is impossible to make the robot move faster, and there exist many robotic tasks for which a simulation is not available. We provide an example of such as task in Chapter 5.

2. The time-step costs real money. This can be energy consumed, money spent renting equipment, money given to actual people carrying-out the action (such as in medical trials), or money not earned because a bad decision has been taken regarding a customer.

In this chapter, we address sample-efficiency at a fundamental level. We introduce a new Reinforcement Learning algorithm, that has sample-efficiency as its main objective. In Chapter 5, we demonstrate that this algorithm is able to learn highly-complicated tasks on real-world robots, without access to any simulator or pre-training. The tasks being solved were previously considered outside the reach of Reinforcement Learning, and even of most model-based planning methods.

3.1 Outline

We introduce Bootstrapped Dual Policy Iteration (BDPI), a sample-efficient model-free Reinforcement Learning algorithm for discrete, continuous or pixel-based state spaces, and discrete actions. BDPI differs from other Reinforcement Learning algorithms in two key areas:

- It learns an actor and *several* critics. Moreover, BDPI learns *off-policy* critics, that approximate the optimal Q^* function instead of the on-policy Q^π function. We explain in Section 2.8.1 that almost every existing "off-policy" actor-critic algorithm still learns an on-policy critic, even if the behavior of the agent may be different from its actor. Because much (critic-only) research focuses on algorithms that learn Q^*, instead of Q^π, highly sample-efficient algorithms are available to efficiently train the BDPI critics.

- The actor itself does not use Policy Gradient, contrary to almost every current actor-critic algorithm. Removing Policy Gradient from the algorithm is what relieves BDPI from the need of on-actor critics (see Section 2.7.3).

Our novel actor learning rule, and the aggressive critic learning scheme it makes possible, leads to a highly sample-efficient algorithm that we evaluate on many simulated environments in Section 3.5, and on a motorized wheelchair in Chapter 5 (along with additional simulated environments).

We first review a range of reinforcement learning algorithms in Section 3.2. Then, we introduce BDPI in Section 3.3, and review some of its theoretical properties. In Section 3.4, we provide and describe a full implementation of BDPI in a tabular setting, as a continuation of our tutorial of Section 2.4.3.

3.2 Motivation and Related Work

State-of-the-art stochastic actor-critic algorithms, used with discrete actions, all share a common trait: the critic Q^π they learn directly evaluates the actor [Konda and Borkar, 1999; Schulman et al., 2017; Wu et al., 2017; Mnih et al., 2016]. Some algorithms allow the agent to execute a policy different from the actor, which the authors refer to as off-policy, but the critic is still on-policy with regards to the actor [Haarnoja et al., 2018, for instance]. ACER and the Off-Policy Actor-Critic [Wang et al., 2016a; Degris et al., 2012] use off-policy corrections to learn Q^π from past experiences, DDPG learns its critic with an on-policy SARSA-like algorithm [Lillicrap et al., 2015], Q-prop [Gu et al., 2017b] uses the actor in the critic learning rule to make it on-policy, and PGQL [O'Donoghue et al., 2017] allows for an off-policy V function, but requires it to be combined with on-policy advantage values. Notable examples of algorithms without an on-policy critic are AlphaGo Zero [Silver et al., 2017], that replaces the critic with a slow-moving target policy learned with tree search, and the Actor-Mimic [Parisotto et al., 2016], that minimizes the cross-entropy between an actor and the Softmax policies of critics (see Section 3.5.2). The need of most actor-critic algorithms for an on-policy critic makes them incompatible with state-of-the-art value-based algorithms of the Q-Learning family [Arjona-Medina et al., 2018; Hessel et al., 2017], that are all highly sample-efficient but off-policy. In a *discrete-actions* setting, where off-policy value-based methods can be used, this raises three questions:

1. Can we use *off-policy* value-based algorithms (more precisely *off-actor*, see Section 2.8.1) in an actor-critic setting?

2. Would these off-actor algorithms enable higher sample-efficiency?

3. Does the actor increase sample-efficiency and/or exploration compared to a critic-only approach such as Q-Learning?

In this chapter, we provide a positive answer to these three questions:

1. Our actor learning rule, inspired by Conservative Policy Iteration (see Sections 2.9 and 3.3.2), is robust to off-policy critics.

2. Because we lift the requirement for on-policy critics, the full range of value-based methods can now be leveraged by the critic, such as DQN-family algorithms [Hessel et al., 2017], or exploration-focused approaches [Arjona-Medina et al., 2018; Burda et al., 2018]. Off-policy value-based algorithms are state-of-the-art in sample-efficiency, and an active research field. To keep BDPI simple and better isolate the properties arising from its actor-critic structure, we use simple DQN-family critics.

3. We learn several Q-Functions, as suggested by Osband et al. [2016], with a novel extension of Q-Learning (see Section 3.3.1). Unlike other approaches, that use the critics to compute means and variances [Nikolov et al., 2019; Chen et al., 2017], BDPI uses the information in each individual critic to train the actor. We show that our actor learning rule, combined with several off-policy critics, can be compared to bootstrapped Thompson sampling (Section 3.3.6). In Section 3.5.4, we empirically demonstrate that BDPI's actor strongly contributes to its sample-efficiency. To the best of our knowledge, this is the first time that, in a discrete-action setting, the benefit of having an actor can be clearly identified.

Finally, and perhaps most importantly, BDPI is highly robust to its hyperparameters, which mitigates the need for endless tuning (see Section 3.5.5). In situations where sample-efficiency matters, being able to learn a task in a few episodes is useless if millions or time-steps have to be spent tuning the parameters of the algorithm.

3.3 Bootstrapped Dual Policy Iteration

Our main contribution, Bootstrapped Dual Policy Iteration (BDPI), consists of two original components. In Section 3.3.1, we introduce an aggressive off-policy critic, inspired by Bootstrapped DQN and Clipped DQN [Osband et al., 2016; Fujimoto et al., 2018]. In Section 3.3.2, we introduce an actor that leads to high-quality

exploration, further enhancing sample-efficiency. We discuss several theoretical aspects of BDPI in Sections 3.3.3 to 3.3.6.

In Section 3.4, we detail a tabular implementation of BDPI. This complements our pseudocode to provide a clear yet compact description of the algorithm. In the Appendix and at https://github.com/vub-ai-lab/bdpi, we provide a neural-network-based BDPI implementation, that we use in Sections 3.5 and Chapter 5.

3.3.1 Aggressive Bootstrapped Clipped DQN

BDPI is an actor-critic algorithm, which means that we have to describe both its actor and critics. We start by introducing Aggressive Bootstrapped Clipped DQN (ABCDQN), the algorithm used to train the critics of BDPI. Like Bootstrapped DQN [Osband et al., 2016], ABCDQN learns $N_c > 1$ critics, instead of just one. All the critics are part of the same agent, and share a single experience buffer, but see distinct samples of experiences every time they are trained. They are also each randomly initialized independently, which makes them predict slightly different Q-Values even when learning has just started. In Section 2.4.2, we discuss how learning several critics participates in making the agent able to explore the environment more efficiently.

Each critic of ABCDQN is trained with an algorithm loosely inspired by Clipped DQN and Double Q-Learning [Fujimoto et al., 2018; van Hasselt, 2010], to ensure stable learning even when the Q-function is approximated by a neural network. Each critic maintains two Q-functions, Q^A and Q^B. Every *training iteration*, Q^A and Q^B are swapped, then Q^A is trained with Equation 3.1 on a set of experiences sampled from an experience buffer, shared by all the critics. We use k in our equations to distinguish between the original Q_k values, that are used to compute updated Q_{k+1} values. After an update, k is implicitly incremented. Contrary to Clipped DQN, an on-policy algorithm that uses $V_k(s_{t+1}) \equiv \min_{l=A,B} Q^l_k(s_{t+1}, \pi(s_{t+1}))$ as target value, ABCDQN removes the reference to $\pi(s_{t+1})$ and instead uses the following formulas:

$$\begin{aligned} Q^A_{k+1}(s_t, a_t) &= Q^A_k(s_t, a_t) + \alpha\big(r_{t+1} + \gamma V_k(s_{t+1}) - Q^A_k(s_t, a_t)\big) \\ V_k(s_{t+1}) &\equiv \min_{l=A,B} Q^l_k\big(s_{t+1}, \operatorname{argmax}_{a'} Q^A_k(s_{t+1}, a')\big) \end{aligned} \tag{3.1}$$

Because we introduce an actor in the next section, that smooths the behavior of the agent and increases exploration quality, we can make the critics more *aggressive*, more sample-efficient, but also susceptible to lack finesse in the actions being selected. We increase the aggressiveness of ABCDQN by performing several *training iterations* per *training epoch*. Every *training epoch*, every critic is updated using a different batch of experiences, for $N_t > 1$ *training iterations*. As mentioned above, a training iteration consists of applying Equation 3.1 on the critic, which produces Q^A_{k+1} values. If Q^A is parametric, such as a neural network, it is denoted as Q^A_θ with θ the parameters. The mean-squared-error loss is used to optimize the parameters of the network with gradient descent, for *several gradient steps*, minimizing $\sum_{(s,a)}(Q^A_\theta(s,a) - Q^A_{k+1}(s,a))^2$, with the sum over every state an action in the batch of experiences for this training epoch.

ABCDQN used alone achieves high sample-efficiency (see Section 3.5), but its exaggerated aggressiveness, key to its sample-efficiency, makes it prone to overfitting, which prevents it from learning particularly difficult tasks. We now introduce an actor, that alleviates this problem and leads to high-quality exploration and high sample-efficiency at the same time. In Section 3.5, we show that both our critic (ABCDQN) and actor, the combination of whose forms BDPI, are required to achieve top performance.

3.3.2 An Actor Robust to Off-Policy Critics

Almost every actor-critic reinforcement learning algorithm learns its actor with a variant of Policy Gradient, as discussed in Section 2.8. The problem with Policy Gradient, and actors using it, is that the critic must precisely evaluate the actor. When experience replay is used, such as in ACER [Wang et al., 2016a], this means that off-policy corrections have to be used to learn an on-actor critic Q^π from experiences that come from some other policy.

Our second contribution is an actor learning rule that does not rely on on-actor critics. Instead of Policy Gradient, our actor draws inspiration from Conservative Policy Iteration [Scherrer, 2014]. While some analyses of CPI still assume an on-policy critic for convenience [Pirotta et al., 2013], the foundation of the algorithm and its equation do not break down when an off-policy critic is used, as would happen with Policy Gradient. We can therefore use an *off-policy* critic, while maintaining stability. Moreover, we show in Section 3.3.6 that off-policy critics lead to an interesting exploration behavior, that we empirically confirm leads to high sample-efficiency in our experiments.

Our actor learning rule consists of training the actor towards the average greedy policy of all the critics. Every *training epoch*, each critic i is updated on its batch of experiences $E_i \subset B$ uniformly sampled from the experience buffer, then the actor is updated towards its greedy policy on that batch:

$$\pi(s) \leftarrow (1-\lambda)\pi(s) + \lambda\Gamma(Q^{A,i}_{k+1}(s,\cdot)) \qquad \forall\ i, \forall\ s \in E_i \tag{3.2}$$

with π the actor, a neural network that maps states to a probability distribution over actions in our experiments, $\lambda = 1 - e^{-\delta}$ the actor learning rate, computed from the maximum allowed KL-divergence δ defining a *trust-region* (see Section 3.3.5), and Γ the greedy function, that returns a policy greedy in $Q^{A,i}$, the Q^A function of the i-th critic. Because the actor learning rate is small, and the actor pursues the greedy policies of the critics in random order, it learns the average greedy policy of the critics. We summarize BDPI in Algorithm 1, and provide an example implementation in Section 3.4.

While expressed in the tabular form in Equations 3.1 and 3.2, the BDPI update rules produce Q-Values and probability distributions that can directly be used to train any kind of function approximator, on the mean-squared-error loss, and for as many gradient steps as desired.

3.3.3 Distilling Big Critics into Small Actors

The Policy Distillation literature [Rusu et al., 2015] suggests that implementing the actor and critics with neural networks, with the actor having a smaller architecture than the critic (less neurons), may lead to good results. Large critics reduce bias [Fu et al., 2019], and a small-network policy has been shown to outperform and generalize better than big-network policies [Rusu et al., 2015].

Algorithm 1 Learning with Bootstrapped Dual Policy Iteration

for every critic $i \in [1, N_c]$ in random order **do**
  $E \leftarrow$ N experiences sampled from the buffer
  for N_t training iterations **do**
    Swap $Q^{A,i}$ and $Q^{B,i}$
    Update $Q^{A,i}$ of critic i on E with Equation 3.1
  end for
  Update actor on E with Equation 3.2
end for

We performed experiments on the *LunarLander* environment, and discovered that having an actor smaller than the critics impairs the learning ability of the agent, and prevents it from learning policies as good as when the actor is represented by a larger network. In Section 3.5, we use actors and critics that have a single hidden layer of 256 neurons for *LunarLander*. We tried actor-critic sizes of 256-256, 128-256 and 256-512, and found that 256-256 works marginally better than the other combinations. We have yet to find a conclusive explanation for this phenomenon, this is left as future research.

3.3.4 BDPI and Conservative Policy Iteration

The standard Conservative Policy Iteration update rule (see Section 2.9) updates the actor π towards $\Gamma(Q^\pi)$, the greedy function according to the Q-Values arising from π. This slow-moving update, and the inter-dependence between π and Q^π, allows several properties to be proven [Kakade and Langford, 2002], and the optimal policy learning rate α to be determined from Q^π [Pirotta et al., 2013].

Because BDPI learns off-policy critics, that can be arbitrarily different from the on-policy Q^π function, the Approximate Safe Policy Iteration framework [Pirotta et al., 2013] would infer an "optimal" learning rate of 0. Fortunately, a non-zero learning rate still allows BDPI to learn efficiently. In Section 3.3.6, we show that the off-policy nature of BDPI's critics makes it an approximation of Thompson sampling, which CPI's on-policy critics do not do. Our experimental results in Section 3.5 further illustrate how BDPI allows fast and robust learning, even in difficult-to-explore environments.

3.3.5 BDPI and its Easy to Implement Trust Region

A trust-region, successfully used in a reinforcement-learning algorithm by Schulman et al. [2015], is a constrain on the Kullback-Leibler divergence between a policy π_k and an updated policy π_{k+1}. In BDPI, we want to find a policy learning rate λ such that $D_{KL}(\pi_k||\pi_{k+1}) \leq \delta$, with δ the *trust-region*.

While a trust-region is expressed in terms of the *KL-divergence*, Conservative Policy Iteration algorithms naturally implement a bound on the *total variation* between π and π_{k+1}:

$$\begin{aligned} \pi_{k+1}(s) &= (1-\lambda)\pi_k(s) + \lambda\pi'(s) && \text{see (3.2)} \\ D_{TV}(\pi_{k+1}(s)||\pi_k(s)) &= \sum_a |\pi_{k+1}(a|s) - \pi_k(a|s)| \\ &\leq 2\lambda \end{aligned} \tag{3.3}$$

The total variation is maximal when $\pi'(s)$, the target policy, and $\pi_k(s)$, both have an action selected with a probability of 1, and the action is not the same. In CPI algorithms, the target policy is a greedy policy $\pi'(s)$, that selects one action with a probability of one. The condition can therefore be slightly simplified: the total variation is maximized if $\pi_k(s)$ assigns a probability of 1 to an action that is not the greedy one. In this case, the total variation is 2λ (2 elements of the sum of Equation 3.3 are equal to λ).

The Pinsker inequality [Pinsker, 1960] provides a lower bound on the KL-divergence based on the total variation. The inverse problem, upper-bounding the KL-divergence based on the total variation, is known as the Reverse Pinsker Inequality. It allows to implement a trust-region, as $D_{KL} \leq f(D_{TV})$ and $D_{TV} \leq 2\lambda$, with $f(D_{TV})$ a function applied to the total variation so that the reverse Pinsker inequality holds. Upper-bounding the KL-divergence to some δ then amounts to upper-bounding $f(D_{TV}) \leq \delta$, which translates to $\lambda \leq \frac{1}{2}f^{-1}(\delta)$.

The main problem is finding f^{-1}. The reverse Pinsker inequality is still an open problem, with increasingly tighter but complicated bounds being proposed [Sason, 2015]. A tight bound is important to allow a large learning rate, hence higher sample-efficiency while maintaining convergence, but the currently-proposed bounds are almost impossible to inverse in a way that produces a tractable f^{-1} function. We therefore propose our own bound, designed specifically for a CPI algorithm, slightly less tight than state-of-the-art bounds, but trivial to inverse.

If we consider two actions, we can produce a policy $\pi_k(s) = \{\varepsilon, 1-\varepsilon\}$ and a greedy target policy $\pi'(s) = \{1, 0\}$. The epsilon value must be positive but can be arbitrarily close to zero, and prevents an indefinite value from appearing in the following formulas. The updated policy $\pi_{k+1} = (1-\lambda)\pi_k + \lambda\pi'$ is, for state s, $\pi_{k+1}(s) = \{\lambda, 1-\lambda\}$. We omit ε in $\pi_{k+1}(s)$ for brevity, as this value tends to zero. The KL-divergence between π_k and π_{k+1} is:

$$\begin{aligned} D_{KL}(\pi_k||\pi_{k+1}) &= \varepsilon \log \frac{\varepsilon}{\lambda} + (1-\varepsilon)\log\frac{1-\varepsilon}{1-\lambda} \\ &= \log \frac{1}{1-\lambda} && \text{for } \varepsilon \to 0^+ \end{aligned} \tag{3.4}$$

with $\lim_{\varepsilon \to 0^+} \varepsilon \log \varepsilon = 0$. Based on the reverse Pinsker inequality, we assume that if the two policies used above assign a probability of 1 to two different actions, and therefore have a maximal total variation, then their KL-divergence is also maximal. We use this result to introduce a trust region:

$$
\begin{aligned}
D_{KL}(\pi_k || \pi_{k+1}) &\leq \delta && \text{trust region} \\
\log \frac{1}{1-\lambda} &\leq \delta \\
\frac{1}{1-\lambda} &\leq e^{\delta} \\
\lambda &\leq 1 - e^{-\delta}
\end{aligned}
$$

Interestingly, for small values of δ, as they should be in a practical implementation of BDPI, $1-e^{-\delta} \approx \delta$. The trust-region is therefore implemented by choosing $\lambda = \delta$, which is much simpler than the line-search method proposed by Schulman et al. [2015].

3.3.6 BDPI and Thompson Sampling

In a bandit setting, Thompson sampling [Thompson, 1933] is regarded as one of the best ways to balance exploration and exploitation [Agrawal and Goyal, 2012; Chapelle and Li, 2011]. Thompson sampling consists of drawing actions from the posterior probability distribution of which action is optimal. In a reinforcement-learning setting, Thompson sampling consists of selecting an action a according to $\pi(a|s) \equiv P(a = \text{argmax}_{a'} Q^*(s, a'))$, with Q^* the optimal Q-function.

BDPI learns off-policy critics, that produce estimates of Q^*. Sampling a critic and updating the actor towards its greedy policy is therefore equivalent to sampling a function $Q \sim P(Q = Q^*)$ [Osband et al., 2016], then updating the actor towards $\Gamma(Q)$, with $\Gamma(Q)(s, a) = \mathbb{1}[a = \text{argmax}_{a'} Q(s, a')]$, and $\mathbb{1}$ the indicator function. Over several updates, and thanks to a small λ learning rate (see Equation 3.2), the actor learns the expected greedy function of the critics, which (intuitively) folds the indicator function into the sampling of Q, leading to an actor that learns $\pi(a|s) = P(a = \text{argmax}_{a'} Q^*(s, a'))$, the Thompson sampling equation for reinforcement learning.

The use of an explicit actor, instead of directly sampling critics and executing actions as Bootstrapped DQN does [Osband et al., 2016], positively impacts

BDPI's performance (see Section 3.5). Nikolov et al. [2019] discuss why Bootstrapped DQN, without an actor, leads to a higher regret than their Information Directed Sampling, and propose to add a Distributional RL [Bellemare et al., 2017] component to their agent. Osband et al. [2018] presents arguments against the use of Distributional RL, discuss the difference between aleatoric uncertainty (originating from the randomness of the environment, captured by Distributional RL) and epistemic uncertainty (originating from lack of exploration or function approximation, in the agent, captured by Bootstrapped DQN), and instead combines Bootstrapped DQN with prior functions. In the next section, we show that BDPI largely outperforms Boostrapped DQN, along with PPO and ACKTR, without relying on Distributional RL nor prior functions. We believe that having an explicit actor changes the way the posterior is computed, which may positively influence exploration compared to actor-less approaches.

3.4 Tabular BDPI

In Section 2.3.2, we started our tutorial on Reinforcement Learning by introducing our virtual bartender environment, in which an animated virtual robot has to learn how to serve glasses to people. Because the bartender interacts with real people (using a keyboard), people take time to make decisions, and people do not want to interact for an agent for too long, sample-efficiency is crucial for the bartender. Our tutorial progressively improves the Reinforcement Learning agent that controls the bartender, starting with simple Tabular Q-Learning in Section 2.3.2, adding Experience Replay in Section 2.3.3, and finally introducing multiple critics for better exploration in Section 2.4.3.

In this section, we continue our tutorial by presenting a Tabular BDPI implementation for the virtual bartender. Tabular BDPI allows the bartender to learn to give glasses to people in only 5 minutes, taking into account that people are still getting familiar with the demonstration during those first minutes, and as such interact slowly with the agent. Tabular Q-Learning with Experience Replay allows the task to be learned in about 15 minutes.[1]

Because the previous parts of tutorial already present Q-Learning with multiple critics, the main part of this section is the introduction of the BDPI actor, and

[1]The COVID-19 crisis that occured during the writing of this thesis unfortunately prevented additional experiments with non-expert people to be carried out, which would have allowed to measure the precise impact of the actor, aggressive critics and bootstrapped critics.

the design of aggressive critics. Because Tabular Q-Learning does not suffer from the approximation bias of Q-Learning with function approximation van Hasselt [2010], we dot not implement Clipped DQN (with the Q^A and Q^B Q-Tables for each critic), but instead use only a single Q-Table per critic.

We first assume an environment env that has n_states states and n_actions actions. We define a few constants that configure how BDPI learns, and create the actor, critics and experience buffer:

```
GAMMA = 0.99        # Discount factor
ALR = 0.01          # Actor learning rate, λ in Equation 3.2
CLR = 0.1           # Critic learning rate, α in Equation 3.1
BATCH_SIZE = 32     # Experiences replayed by each critic every time-step
NUM_CRITICS = 16    # Number of critics, 16 is more than enough
Q_LOOPS = 4         # Training iterations per training epochs

actor = np.ones((n_states, n_actions)) / n_actions # Uniform probability distribution
critics = np.random.random((NUM_CRITICS, n_states, n_actions)) * 0.01
transitions = collections.dequeue([], 1000)
critic_indexes = list(range(NUM_CRITICS))
```

The structure of the agent is still the same as in the previous parts of our tutorial, with a loop over episodes and a loop over time-steps. In the code snippet below, we show that the actor is used to select actions executed by the agent. The learning part of the algorithm is shown later.

```
for i in range(1000): # 1000 episodes
    state = env.reset()
    done = False
    cumulative_reward = 0.0

    while not done:
        # Select an action with the actor
        probas = actor[state]
        action = np.random.choice(range(n_actions), p=probas)

        # Store the experience in the buffer
        next_state, reward, done, _ = env.step(action)
        transitions.apppend((state, action, reward, next_state, done))
```

```
            # Learn
            # [...shown later...]

            # Move to the next state
            cumulative_reward += reward
            state = next_state

        print('Episode', i, 'return', cumulative_reward)
```

Every time-step, the actor and all the critics are updated. Each critic sees a different batch of experiences, is trained with Q-Learning for Q_LOOPS training iterations, then produces a greedy policy to be pursued by the actor. We found that increasing Q_LOOPS increases sample-efficiency, at the cost of compute time, and with slightly diminishing returns. However, with a neural network implementation of BDPI, it is important to keep Q_LOOPS relatively small (maximum 4), to prevent the network from overfitting the particular batch of experiences it is trained on (see Section 2.6.1). In all cases, it is important that the critics are trained in random order, so that it is not always the same critic that trains the actor last, and therefore influences it the most.

```
# Learn (insert at [...shown later...] in the previous snippet)
random.shuffle(critic_indexes)

for critic_index in critic_indexes:
    # Q-Table to update and batch of experiences for this critic
    qtable = critics[critic_index]
    batch = sample_wr(transitions, BATCH_SIZE)

    # For every training iteration, apply Q-Learning on every experience
    for qloop in range(Q_LOOPS):
        for s, a, r, sn, d in batch:
            if d:
                y = r
            else:
                y = r + GAMMA * qtable[sn].max()

            qtable[s, a] += CLR * (y - qtable[s, a])
```

```
# Update the actor towards the greedy policy of the critic
for s, a, r, sn, d in batch:
    greedy_action = qtable[s].argmax()

    # Pursue the greedy action
    target_probas = np.zeros((n_states))
    target_probas[greedy_action] = 1.0

    # Apply Equation 3.2
    actor[s] = (1. - ALR) * actor[s] + ALR * target_probas
```

This concludes the implementation of Tabular BDPI. We performed numerous experiments on the virtual bartender task, with many visitors of our lab interacting with the agent over the years, and found that a simple Tabular Q-Learning with Experience Replay, as described in Section 2.3.3, learns to correctly give glasses in about 15 minutes. With Tabular BDPI, far less mistakes are made, far less bad actions are repeatedly tried, and the agent learns the task in about 5 minutes. We must keep in mind that this is a demonstration task, with a presenter talking while the task is performed. The person interacting with the agent is therefore slow at answering questions in the beginning of the demonstration, as they get used to the bartender. The 5-vs-15 minutes difference between BDPI and a single critic with experience replay is therefore larger than it seems, as these first 5 minutes are mostly dedicated to explaining what the demo is about, and as such see less interactions between the person and the agent.

We now move on to an in-depth empirical evaluation of BDPI with a neural-network-based actor and critics. This more complete implementation of BDPI is compatible with real-valued and pixel-based environments, and is available at https://github.com/vub-ai-lab/bdpi. This is also the version of BDPI we apply to a motorized wheelchair, as described in Chapter 5.

3.5 Experiments

To test the effectiveness of BDPI on a wide range of problems, we compare the performance of BDPI to its ablations[2] and a wide range of reinforcement learning algorithms, in four environments with completely different state-spaces and

[2] An ablation consists of removing a part of an algorithm, for instance BDPI's actor, to assess the impact and contributions of individual components of the algorithm.

dynamics. Our results demonstrate the high sample-efficiency and exploration quality of BDPI, crucial in real-world settings where every time-step costs time or resources. Moreover, these results are obtained with the same configuration of critics, experience replay and learning rates across environments, which demonstrates that BDPI does not need to be fine-tuned for every environment, which dramatically increases its ease of deployment. In Section 3.5.5, we carry out further experiments, that demonstrate that BDPI is more robust to its hyper-parameters than other algorithms. This is key to the application of reinforcement learning to real-world settings, where vast hyper-parameter tuning is often infeasible.

3.5.1 Algorithms

We evaluate the algorithms listed below:

BDPI	*this thesis*
ABCDQN, BDPI without an actor	*this thesis*
BDPI w/ AM, see Section 3.5.2	*this thesis*
BDQN, Bootstrapped DQN	Osband et al. [2016]
PPO	Schulman et al. [2017]
ACKTR	Wu et al. [2017]

Except for the *Hallway*[3] environment, described in the next section, all algorithms use feed-forward neural networks to represent their actor and critic, with one (2 for PPO and ACKTR) hidden layers of 32 neurons (256 on *LunarLander*). The state is one-hot encoded in *FrozenLake*, and directly fed to the network in the other environments. The neural networks are trained with the Adam optimizer [Kingma and Ba, 2014], using a learning rate of 0.0001 (0.001 for PPO, ACKTR uses its own optimizer with a varying learning rate). Unless specified otherwise, BDPI uses $N_c = 16$ critics, all updated every time-step on a different 256-experiences batch, sampled from the same shared experience buffer, for 4 applications of our ABCDQN update rule. BDPI trains its neural networks for 20 epochs per training iteration, on the mean-squared-error loss (even for the policy).

Hallway being a 3D environment, the algorithms are configured differently. Changes to BDPI are minimal, as they only consist of using the standard DeepMind convolutional layers described by Mnih et al. [2016], a hidden layer of 256 neurons, and optimizing the networks for 1 epoch per training iteration, instead of 20. PPO and ACKTR, however, see much larger changes. They use the DeepMind

[3] https://github.com/maximecb/gym-miniworld

layers, 16 replicas of the environment (instead of 1), a learning rate of 0.00005, and perform gradient steps every 80 time-steps (per replica, so 1280 time-steps in total). These PPO and ACKTR parameters are recommended by the author of *Hallway*. We point out that the use of 16 replicas of the environment, crucial to the learning performance of PPO and ACKTR, is impossible in real-world settings where only one robot is available.

3.5.2 BDPI with the Actor-Mimic Loss

To the best of our knowledge, the Actor-Mimic Parisotto et al. [2016] is the only actor-critic algorithm, along with BDPI, that learns critics that are off-policy with regards to the actor. The Actor-Mimic is designed for transfer learning tasks. One critic per task is trained, using the off-policy DQN algorithm. Then, the cross-entropy between the actor and the Softmax policies $S(Q_i)$ of all the critics is minimized, using the (simplified) loss of Equation 3.5.

$$\mathcal{L}(\pi_\theta) = -\sum_{s\in S, a\in A, i<N} S(Q_i)(a|s)\log(\pi_\theta(a|s)) \tag{3.5}$$

Applying the Actor-Mimic to a single-task setting is possible. We implemented an agent based on BDPI, that retains its ABCDQN critics, but replaces our actor learning rule of Equation 3.2 with the Actor-Mimic loss of Equation 3.5. Because we only change how the actor is trained, and still use our aggressive critics, we ensure the fairest comparison between our actor learning rule and the cross-entropy loss of the Actor-Mimic. In our experiments, the Actor-Mimic loss with Softmax policies fails to learn efficiently, even after extensive hyper-parameter tuning, probably because the Softmax prevents the policy from becoming deterministic in states where this is necessary. This indicates the necessity of replacing the Softmax with the greedy function, which led to the much better results that we present in Section 3.5.4.

3.5.3 Environments

Our evaluation of BDPI takes place in four environments, that illustrate a variety of challenges in Reinforcement Learning. We first list the challenges we consider, and in which environments they appear, then provide a detailed description of every environment.

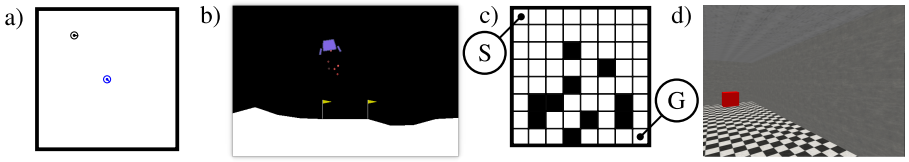

Figure 3.1: The four environments. a) *Table*, a large continuous-state environment with a black circular robot and a blue charging station. b) *LunarLander*, a continuous-state task based on the Box2D physics simulator. c) *Frozen Lake*, an 8-by-8 slippery gridworld where black squares represent fatal pits. d) *Hallway*, a 3D pixel-based navigation task.

1. Sparse rewards. In *Table* and *FrozenLake*, the agent receives a reward of 0 every time-step, and a positive reward only when reaching a target location in the environment. In *Table*, the agent needs about 100 time-steps before reaching the goal. In *FrozenLake*, the agent needs only 14 time-steps to reach the goal. Sparse rewards are challenging for exploration, as the agent receives no informative guidance until it reaches the goal for the first time.

2. High dimensionality. In *LunarLander*, the agent observes vectors of 8 floating-point numbers, that all have a precise meaning (such as a velocity or an angle). In *Hallway*, the agent directly observes 40-by-40 pixel color images, so 4800 floating-point values. High dimensionality stresses the ability of the agent to produce data (states, probabilities, Q-Values) that allows efficient training of a neural network.

3. High stochasticity. In *FrozenLake*, the environment moves the agent to a random neighboring cell with a probability of 0.66, every time-step, and executes the action that the agent wants to perform only with a probability of 0.33. High stochasticity complicates directed exploration (the agent wants to go precisely somewhere), and introduces noise in the data computed by the agent as it learns.

Three of the environments we consider in this section are widely-accepted benchmarks, available online. *Table* is a new environment that we introduce, made to be challenging from an exploration perspective. It is available as supplementary material. The environments are described as follows:

Table simulates a tiny robot on a large table that has to locate its charging station and dock (see Figure 3.1a). The table is a 1-by-1 square. The goal is located at $(0.5, 0.5)$, and the robot always starts at $(0.1, 0.1)$, facing away from the goal. A fixed initial position makes exploration more challenging, as the robot never spawns close to the goal. The robot observes its current (x, y, θ) position and orientation, with $\theta \in [-\pi, \pi]$. Three actions allow the robot to either move forward 0.005 units, or turn left/right 0.1 radians. A reward of 0 is given every time-step. The episode finishes with a reward of -50 if the robot falls off the table, 0 after 200 time-steps, and 100 when the robot successfully docks, that is, its location is $(0.5 \pm 0.05, 0.5 \pm 0.05, \frac{\pi}{4} \pm 0.3)$. The slow speed of the robot and reward sparsity make *Table* more difficult to explore than most Gym tasks [Brockman et al., 2016].

LunarLander is a high-dimensional continuous-state physics-based simulation of a rocket landing on the moon (see Figure 3.1b). The agent observes the location and velocities of various components of the lander, and has access to four actions: doing nothing, firing the left/right side engines for one time-step, and firing the main engine. The reward signal for this task is quite complicated but informative, as it directly maps the distance between the rocket and the landing pad to a reward, on every time-step. The environment is considered solved when a cumulative reward of 200 or more is achieved per episode [Brockman et al., 2016].

FrozenLake is a 8×8 grid composed of slippery cells, holes, and one goal cell (see Figure 3.1c). The agent can move in four directions (up, down, left or right), with a probability of $\frac{2}{3}$ of actually performing an action other than intended. The agent starts at the top-left corner of the environment, and has to reach the goal at its bottom-right corner. The episode terminates when the agent reaches the goal, resulting in a reward of $+1$, or falls into a hole, resulting in no reward.

Hallway is a 3D pixel-based environment, that simulates a camera-based robotic task in the real world. *Hallway* consists of a rectangular room with a target red box, and the agent. The size of the room, location of the goal and initial position of the agent are randomly chosen for each episode. Four discrete actions allow the agent to move forward/backward and turn left/right. Movement is slow, and the amount of movement is stochastic for each time-step. The reward signal is sparse: 0 every time-step, and 1 when the goal is reached. The episode ends with a reward of 0 after 500 time-steps. This sparse reward function heavily stresses the ability of a reinforcement-learning algorithm to train deep convolutional neural networks on small amounts of reward data.

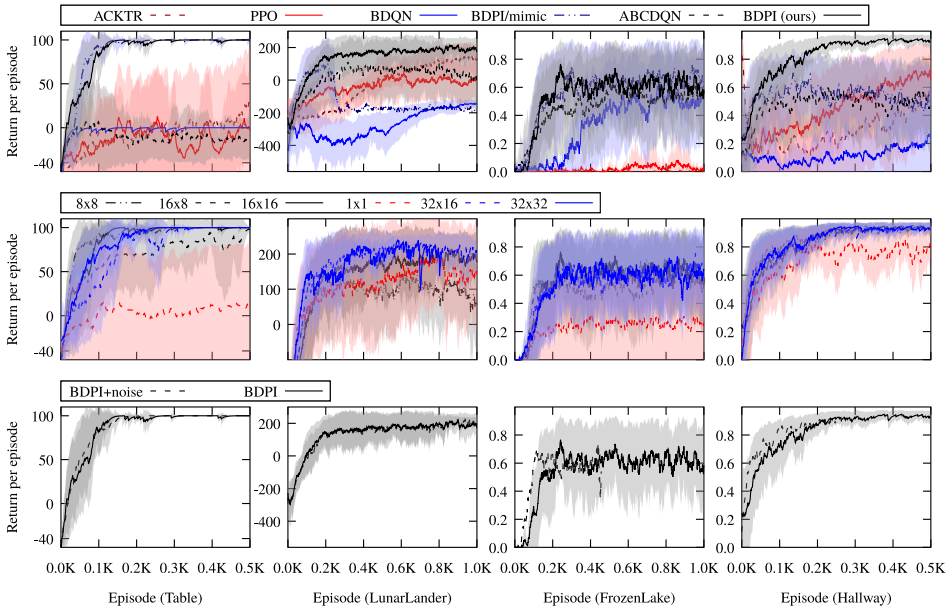

Figure 3.2: *Top:* BDPI (16 critics, updated for 4 iterations per time-step) outperforms all the other algorithms in every environment. *Middle:* Varying the number of critics $\times$ how often they are trained per time-step only has minimal impact on BDPI's performance. *Bottom:* Adding off-policy noise (see text) does not impact BDPI on any of the environments.

3.5.4 Results

Figure 3.2 shows the cumulative reward per episode obtained by various agents in the four environments described in the previous section. These results are averaged across 8 runs per agent, with the shaded regions representing the standard error. The plots compare BDPI to the algorithms detailed in Section 3.5.1, and display the effect of varying key hyper-parameters of BDPI.

Algorithms

Across all the environments, BDPI leads to the highest returns and exhibits the highest sample-efficiency:

Table is particularly difficult to explore, with its sparse reward. In this environment, only BDPI (with our actor) and BDPI with the Actor-Mimic actor loss manage to learn the task in the 500 episodes window shown in our plots. PPO and ACKTR also eventually learn the task, but only after more than 1K episodes (not shown in the figure). ABCDQN, BDPI without an actor, fails to learn *Table*, which indicates that BDPI's actor, more specifically its influence on exploration discussed in Section 3.3.6, actively contributes to the quality of its exploration. We point out that ABCDQN uses several critics (as does BDPI). In the figure above, the critic responsible for choosing an action is chosen every time-step. We also tried, in ABCDQN, to select the critic every episode, as suggested by Osband et al. [2016], but this did not change the results on *Table*.

LunarLander has a more complex state-space than *Table* (8 real values that are not the one-hot encoding of a discrete state), and more complex dynamics. BDPI still outperforms all the other algorithms in this environment, even if the simpler reward functions may explain why the different algorithms perform more similarly on this environment than on *Table*. Interestingly, BDPI with the Actor-Mimic loss sees its performance degrade after about 200 episodes, in every of the 8 runs performed. BDPI with our actor does not exhibit this behavior. *LunarLander* is therefore the environment that shows that our actor loss is necessary for stable and top performance, like *Table* shows that having an explicit actor is necessary for good exploration.

FrozenLake has a high stochasticity. Intuitively, there is not much the agent can do in an environment that is so stochastic. We observe in the Figure 3.2 that all the algorithms that have critics and use experience replay (BDPI, BDPI with the Actor-Mimic loss, ABCDQN and Bootstrapped DQN) learn policies of roughly the same quality, at about the same time. BDPI is still learning a better policy earlier than the other algorithms. PPO and ACKTR, after extensive tuning and with several implementations tested, are not as sample-efficient as BDPI and Bootstrapped DQN. They need about 5K episodes before learning *FrozenLake* (not shown on the figure). *FrozenLake* therefore appears to be an environment where exploration quality is not particularly

relevant, but sample-efficiency is. We note that it is the complete opposite from *Table*, where exploration quality was key, and point out that BDPI outperforms all the other algorithms in these two environments, without any change in its hyper-parameters.

Hallway is a pixel-based task, on which BDPI performs surprisingly well (highest curve, low standard error). PPO and ACKTR also perform much better on *Hallway* than on the other tasks. We believe that it may be due to the fact that, because pixel-based environments are highly common in modern Reinforcement Learning literature, current algorithms and hyper-parameters may focus more on the representation learning problem than on the reinforcement learning aspect of tasks. Also note that on *Hallway*, PPO and ACKTR use 16 replicas of the environment (instead of 1 for BDPI, and PPO/ACKTR on the other environments). This setting greatly stabilizes the algorithms, but cannot be applied to real-world physical robots.

Number of Critics

Increasing the number of critics leads to smoother learning curves in every environment, at the cost of sample-efficiency in *Table*, where a higher variance in the bootstrap distribution of critics seems to help with exploration. Having only one critic seriously degrades BDPI's performance, and having less than 16 critics is detrimental on *LunarLander*, where the environment dynamics are complex. This indicates that more critics are beneficial in complex environments, but may slightly reduce pure exploration.

Aggressiveness

The number of training iterations per training epochs (Q_LOOPS in Section 3.4) positively impacts sample-efficiency in some environments, especially *Table* (where Q-Values have to "travel" a long distance from the goal to the initial state, due to the sparse reward function), but not in environments that have a more informative reward function. In order to keep Figure 3.2 concise, we perform 4 training iterations per training epochs in all the plots. Raw experimental results for varying numbers of training iterations are available in the appendix.

	Sparse *Table*	Dimension *LunarLander*	Stochastic *FozenLake*	Pixels *Hallway*	Run-time 1T	32T
ACKTR	✗	✓	✗	✓	34 min	
PPO	✗	✓	✗	✓	38 min	
BDQN	✗	✗	✓	✗	2 h	
ABCDQN	✗	✓	✓	✗	27 h	
BDPI	✓	✓	✓	✓	29 h	1 h 11

Figure 3.3: Comparison of BDPI to other algorithms, regarding the environments in which they perform well. BDPI is the only algorithm to perform well in every environment, at the cost of compute-efficiency (1T and 32T: run-time of a single run of BDPI on *LunarLander*, with 1 or 32 processes, see Section 5.6).

Off-Policy Noise

BDPI's actor learning equations do not refer to any behavior policy or on-policy return, and its critics are learned with a variant of Q-Learning. This hints at BDPI being an off-policy algorithm. We now empirically confirm this intuition. In this experiment, training episodes have, at each time-step, a probability of 0.2 that the agents executes a random action, instead of what the actor prescribes (0.05 on *Table*, where docking requires precise moves). The agent learns only from training episodes. Testing episodes do not have this noise, so that the learning curves produced with off-policy noise (testing episodes) can be compared to the learning curves of BDPI with no off-policy noise. Such off-policy noise does not negatively impact BDPI's learning performance. Robustness to off-policy execution is an important property of BDPI for safety-critical tasks with backup policies.

Compute Efficiency

In Figure 3.3, we summarize the Reinforcement Learning challenges exhibited by the four environments we evaluate BDPI on, and compare how the algorithms we evaluated in this section cope with these challenges. Our general conclusion is that BDPI outperforms every algorithm in every environment. Our ablative study shows that this performance comes from the combination of high sample-efficiency in the critics, and our actor learning rule. However, we acknowledge that there is no free lunch: BDPI's performance, again demonstrated in Section 5.7.4 on a highly-challenging simulated navigation task, comes at the cost of compute efficiency. On an AMD Threadripper 2990WX (32 cores, 3.6 Ghz under load) with

32 GB of DDR4-2966 Mhz memory, a single-threaded version of BDPI can take a day to learn a task that only takes a few minutes for PPO to learn. The multi-threaded version of BDPI we present in Section 5.6 addresses this issue, but still cannot compete with PPO in terms of compute efficiency. We designed BDPI to make the most out of few experiences generated by the environment, extensively and repeatedly learning from each of them. With BDPI, we focus on settings, physical machines with no simulator for instance, where experiences are generated so slowly that sample-efficiency is paramount, and a significant amount of compute time is available between experiences.

The performance of BDPI, arising from its several off-policy critics and novel actor, has been obtained in our experiments using a single set of hyper-parameters for all the environments[4]. Being able to reuse hyper-parameters across widely different environments illustrates BDPI's strong robustness to hyper-parameters, that we now rigorously demonstrate.

3.5.5 Robustness to Hyper-Parameters

Hyper-parameters often need to be tweaked depending on the environment. Therefore, it is highly desirable that an algorithm provides good performance even if not optimally configured, as BDPI does. To objectively measure an algorithm's robustness to its hyper-parameters, we draw inspiration from sensitivity analysis. Thousands of runs of the algorithm are performed on randomly-sampled configurations of hyper-parameters, with each configuration evaluated on the total reward obtained over 800 episodes on *LunarLander*. Then, we compute the average absolute difference of total reward between random pairs of configurations, weighted by their distance in configuration space. This measures how much changing hyper-parameters affects performance. The appendix gives more details, and lists the hyper-parameters we consider for each algorithm.

We evaluated numerous algorithms available in the OpenAI baselines. The algorithms, sorted by ascending sensitivity, are DQN with Prioritized ER (930), BDPI (1167), vanilla DQN (1326), A2C (2369), PPO (2452), then ACKTR (5815). Figure 3.4 shows that the apparent robustness of DQN-family algorithms comes from them performing equally badly for every configuration. 35% of BDPI's configurations outperform the best configuration among all the other algorithms.

[4]Only the number of hidden neurons changes between some environments, which is related to the state representation and not the actual Reinforcement Learning algorithm.

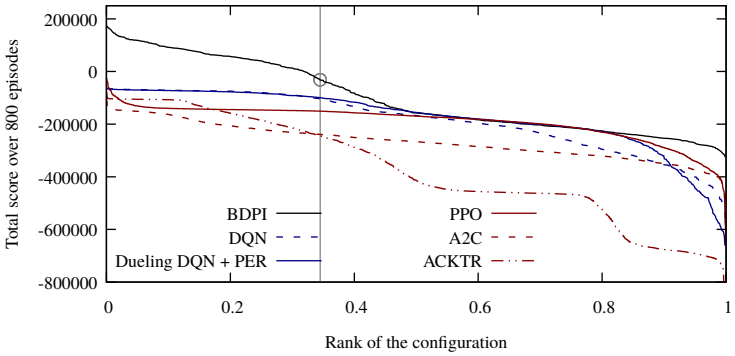

Figure 3.4: Total reward per configuration, sorted by descending total reward, for 800 episodes on *LunarLander*. This plot shows that more than 35% of BDPI's randomly-sampled configurations perform better than the best (PPO) configuration. The worst BDPI configuration is also better than most of the configurations of the other algorithms. On LunarLander, for 800 episodes, the random policy achieves a total reward of about -240K.

3.6 Conclusion

In this chapter, we presented Bootstrapped Dual Policy Iteration (BDPI). Instead of Policy Gradient, fundamentally incompatible with critics that learn Q^*, and as such limiting the sample-efficiency achievable by conventional actor-critic algorithms, our actor learning rule is inspired from Conservative Policy Iteration and is compatible with our off-policy critic learning rule. We compare the exploration behavior of BDPI to Thompson sampling, then empirically validate our claims of high sample-efficiency in several simulated environments. BDPI outperforms every algorithm in every environment, both in sample-efficiency and the quality of the learned policy. We believe that BDPI finally enables the use of model-free Reinforcement Learning in real-world settings, where sample-efficiency is key. For instance, in Chapter 5, we deploy BDPI on a real-world motorized wheelchair, leading to an agent able to learn complex policies in the real world in only one hour. BDPI also starts to attract attention in the Reinforcement Learning community, with numerous researchers now applying BDPI to highly-challenging tasks, such as shunting trains, or preventing diseases to spread in a population of indi-

viduals [Libin, 2020, the latest version of these works, that use BDPI, are not yet published].

4 | Option-Observation Initiation Sets

This chapter is drawn from our publication, Steckelmacher, Harutyunyan, Roijers, Vrancx, Plisnier and Nowé, *Reinforcement Learning in POMDPs with Memoryless Options and Option-Observation Initiation Sets*, published in the proceedings of the AAAI Conference on Artificial Intelligence (AAAI), 2018.

Video of our robotic experiment: https://www.youtube.com/watch?v=VprJZEOD5NE

Cameras do not see through walls. In Chapters 2 and 3, we focus on Reinforcement Learning settings where the agent observes everything it needs to make a decision. Some real-world tasks are more difficult, though. The agent may be unable to see behind obstacles, or has to move from office to office without knowing on which floor it is, or may not observe its current speed in a task that requires precise speed control. All these settings are considered *partially-observable*, as the agent does not observe everything it needs to make a decision based only from its *current* observation. Recognizing that a task is partially-observable is often deemed challenging, even for experts. We propose an extremely simple criterion for partial observability, that directly comes from the definition of a POMDP (see Section 4.1.1):

$$r_t = f(s_t, a_t) \qquad \text{(An MDP)}$$
$$r_t = f(s_t, a_t, \text{something else}) \qquad \text{(A POMDP or harder)}$$

Intuitively, one just has to look at the reward function of the environment. If the entirety of the reward function, special-cases included, can be computed *only from what the agent sees*, then the task is Markovian. If there is anything else needed, even just a boolean variable that sometimes changes and is not observed by the agent, then the task is *partially-observable*. We further distinguish two families of partially-observable environments:

1. The agent does not observe *something that can be counted*, for instance the ID of the office it is in, the current floor, whether a distant basket is full or empty, etc. A finite number of bounded integer values are not observed.

2. The agent does not observe *a real-valued quantity*. For instance, it observes its acceleration, but not its speed. Or it does not know its position in a room.

The second family of partially-observable environments is harder to address than the first one. Fortunately, many real-world environments fall in the first family, mainly because it is often possible to design the environment so that every real-value the agent needs is observed or well estimated, and the only missing pieces of information are discrete. In this chapter, we present our second contribution, a simple method that allows partially-observable tasks of the first kind to be solved with high sample-efficiency.

4.1 Addressing Partial Observability

The main challenge we focus on in this chapter is partial observability. In this section, we review current approaches to tackle partial observability, then focus on the use of recurrent neural networks in Reinforcement Learning, and present an actual implementation of Q-Learning with a recurrent neural network, able to solve a variety of partially observable tasks. In Section 4.2, we briefly discuss another challenge in Reinforcement Learning, complicated tasks that take a long time to solve, and present solutions to it. Finally, in Section 4.3, we consider both challenges at the same time, and propose an elegant algorithm that addresses

partial observability in long-running tasks. The elegance of the algorithm comes from the fact that addressing two challenges does not make the algorithm twice as complicated. Instead, we show that Options, used to address long-running tasks, go 80% of the way towards addressing partial observability, even if not originally designed in that direction.

4.1.1 Partially-Observable MDPs

Until now, we discussed partially-observable environments in an intuitive way. We now define the Partially-Observable Markov Decision Process, that formalizes the most common kind of partially-observable environment considered in the Reinforcement Learning literature.

A Partially Observable MDP (POMDP) $\langle \Omega, S, A, R, T, O, \gamma \rangle$ is a Markov Decision Process (see Section 2.2.1) extended with two components: the possibly-infinite set Ω of observations, and the $O : S \to \Omega$ function that produces observations x based on the unobservable state s of the process. Two different states, requiring two different optimal actions, may produce the same observation. The reward function and transition function still depend on the actual state of the environment. The difficulty of learning in a POMDP therefore comes from two of its properties:

1. The agent does not observe the data used to compute its rewards. It therefore becomes impossible for it to learn a *memoryless* policy, that takes a single observation as input and produces an action, that maximizes the cumulative reward.

2. Exploration becomes even more challenging than in an MDP, as *information-gathering* actions become necessary. The agent has to explicitly learn to perform actions that do not lead to a reward, but allows it to better evaluate what the state of the environment is, in order to make better decisions later on.

Because many real-world tasks are POMDPs, much research has been dedicated to them. We now review existing algorithms that have been shown to work well in POMDPs.

4.1.2 Literature Review

The main challenge with POMDPs is that they are not all *possible*. A degenerate task, that consists of the agent observing an uninformative observation $x_\emptyset$ every time-step, while still having to drive a car in dense traffic, perfectly matches the POMDP framework. However, driving a car without observing anything is impossible. The existence of impossible POMDPs intuitively shows that POMDPs range in shape and form, with some of them being quite easy to solve, some of them much harder, and some of them impossible. This translates into a very large range of algorithms designed for POMDPs, each family of algorithms tailored for a specific subset of tasks, that are partially-observable in a different way, or present different challenges. In this section, we review 5 well-known families of algorithms, and detail the kinds of POMDPs they are best suited for.

Belief States

The first family of algorithms relies on the definition of the POMDP, that assumes that the environment is always in a single state s, not observed by the agent. By observing x, the agent tries to infer in which state s the environment may be, and selects actions according to this *belief of what s is* [Cassandra et al., 1994]. Practically, the state-space S is assumed to be discrete, with $|S|$ states. The *belief state* is a vector of probabilities in $\mathcal{R}^{|S|}$, with one probability per state. At the beginning of the episode, the belief is uniform. Every time-step, the current belief and current observation are used to compute a new belief. The belief is then used as a continuous state by the agent. The policy therefore maps a belief state, a vector of real values, to an action [Kaelbling et al., 1998]. Belief states are robust to noise, and easily explainable (they do not compute any magical value, or behave like a black box), but they almost always require discrete states. Dallaire et al. [2009] review a few approaches to POMDPs with continuous states, that mainly consist of discretizing the state in smart ways, then propose a Bayesian approach that maintains the continuous nature of the POMDP.

External Memory

Some POMDPs are *almost Markovian*, which means that only a small amount of information is hidden from the agent. An example of such as task is given in Section 4.3.1, where an agent has to fetch objects from terminals, that may become empty. That a terminal is empty can be observed only when the agent is close to it. The agent then has to remember that information during the remaining

of the episode, so that it does not go to that terminal again. An agent with external memory observes the current observation x_t, along with the contents of some finite-size memory m_t. The action set of the agent is extended, so that it can not only execute actions in the POMDP, but also modify its memory. A simple external memory is memory bits [Peshkin et al., 1999], that allows the agent to set and clear a few bits. This easily allows the agent to remember whether something is full or empty, whether a flag was set, or on which floor it is. The Neural Turing Machine [Zaremba and Sutskever, 2015] adds a tape to the agent, with symbols on it. The agent is able to observe x_t and the symbol at the current *head location* on the tape. Actions allow the agent to move the head and write symbols on the tape. Neural Turing Machines are potentially able to learn any program. In practice, they have been shown to be able to learn to count and perform mathematical operations [Zaremba and Sutskever, 2015; Graves et al., 2016].

Predictive State Representations

Predictive State Representations (PSRs) consists of maintaining a belief over what the agent will observe in the future, instead of over what the state of the environment is [Littman et al., 2001]. The main argument for PSRs is that learning and representing what the agent will observe in the future given the current observation is easier than recovering a hidden state. Intuitively, the state of the environment may be highly complex, and difficult to recover, while most of the information it conveys is useless for the agent to solve the task. By instead reasoning about future observations, the agent not only relies more on data it actually observes (time passing, and observations, instead of guessing a state it can never see), but can also focus its learning effort on information that allows the task to be learned.

Predictive State Representations are usually used as models of an environment. A *test* is a sequence of actions and observations, in the form of $t \equiv (a_1 o_1 a_2 o_2 ... a_N o_N)$. The probability p_t is the probability that the test t is true given a history of past actions and observations. *Learning* the PSR consists of finding a small set of *core tests*, and learning a function that computes p_t for every core test and every history [Littman et al., 2001]. From the p_t probabilities of every core test, the probability of any other test the agent may need can be computed [Littman et al., 2001; Rudary and Singh, 2004].

Most research on PSRs limits itself to learning the PSR, as a way to model and encode a partially-observable task. The use of this model to learn a policy is successfully demonstrated by Boots et al. [2011], where experiences collected by a Reinforcement Learning agent allows the PSR to be learned, and a policy to

be produced from it. However, the algorithm presented by Boots et al. [2011] is extremely difficult to understand and to implement. Both tests and histories need to be represented by features, vectors of real values, computed in an environment-specific way and for which only a few examples are given in the paper. The algorithm also repeatedly inverts large matrices, and performs *tensor multiplications*, an operation that is not widely available in software packages. For this reason, research on PSR seems to have slowed down in the late 2010s, as progress has been made in recurrent neural networks, that we now discuss.

Recurrent Neural Networks

Recurrent neural networks (RNNs) are a family of neural networks that are able to map a *sequence of inputs* to an output, instead of just a single input to an output. The sequence of inputs can be of arbitrary length, but all the inputs must have the same shape. Examples of inputs can be sequences of images sharing the same resolution, sequences of vectors of floats that have the same length, etc. The neural network consumes one input after the other, without any limit on how many inputs are fed to it, then predicts an output. In some setting, the network is able to predict a *candidate output* after every time an input is consumed.

The most common form of RNNs is the Long Short-Term Memory (LSTM) architecture, described in Section 4.1.3. [Bakker, 2001] propose to use an LSTM network to learn Q-Values. The network is therefore able to map sequences of observations to Q-Values, which makes the agent able to make use of *past observations* when selecting actions. [Bakker, 2001] show that this allows various partially-observable environments to be solved optimally. However, RNNs have difficulties learning the long-term impact of some individual inputs [Mikolov et al., 2014]. In a Reinforcement Learning setting, this means that RNN-based agents have a tendency to be good at remembering observations for a few time-steps, but are usually unable to select actions depending on observations that happened earlier than a dozen time-steps. Our contribution of Section 4.3 addresses this problem in an elegant way, for several kinds of POMDPs. Despite their long-term memory issues, RNNs are often used in POMDPs, as they are extremely easy to implement [Mnih et al., 2016], as we discuss in Section 4.1.3.

4.1.3 Recurrent Neural Networks

Most of the research on neural networks consists of varying their structure, that is, the amount, shape and kind of layers that make up the network. Once the

parametric function, that a neural network is, is built, optimizing its parameter can be done in a generic way that does not depend on the shape of the network. In Section 2.5, we introduce *feed-forward* neural networks. They take an input on one side, then progressively morph it into the output through a finite set of layers. The output of one layer feeds the layer after it. Some feed-forward networks have slightly more advanced architectures, where the output of a layer can feed *several* layers after it [He et al., 2016, the very famous ResNet]. However, all feed-forward neural networks share one property: the output of a layer *never* feeds a layer that comes *before it*. There is no loop in the network.

Recurrent neural networks, on the other hand, have loops. The first recurrent neural networks simply had a layer, somewhere in the network, whose output was fed back to its input [Werbos et al., 1990]. However, training these networks was difficult, as computing their gradient with backpropagation had a tendency of being unstable. Every loop of the loop either increased the magnitude of the gradient, leading to gradient blow-up, or decreased it, leading to gradient vanishing [Hochreiter and Schmidhuber, 1997]. Designing recurrent neural networks that are able to ingest more than a few inputs before producing an output therefore requires care. Hochreiter and Schmidhuber [1997] propose a new, more complicated recurrent architecture: the Long Short-Term Memory *cell*, that acts as an integrator. It takes an input, that is added to an internal accumulator (a simple floating-point value), that then produces the output. *Several* recurrent connections allow the LSTM cell to learn when to clear its accumulator, by which value to multiply it when producing the output, and how to scale the input before adding it to the accumulator. Careful design of activation functions ensures that the gradient that flows from the output to the input of the cell maintains its magnitude, so that it neither blows up or vanishes. LSTM networks are now commonly used, and are available in many neural network libraries. Gated Recurrent Units [Chung et al., 2014] also allow efficient learning of recurrent neural networks with long input sequences, and even appear to slightly outperform LSTM units in a Reinforcement Learning setting [Steckelmacher and Vrancx, 2015].

4.1.4 Q-Learning with LSTM Networks

Even though our contribution of Section 4.3 does not rely on recurrent neural networks, and neither the rest of this thesis, we believe that learning Q-Values (or a policy) with a recurrent neural network is both easy and useful in several tasks. As such, in this section, we show how to implement a basic Q-Learning agent that

is able to learn partially-observable tasks. We base our code on the Tabular Q-Learning agent with Experience Replay of Section 2.3.3, and replace the Q-Table with a recurrent neural network implemented with the PyTorch library (presented in Section 2.5.4). We start with some imports, so that all the snippets provided in this section, combined one after the other, form a fully-working Python script:

```
import gym
import numpy as np
import torch

import random
import collections
```

The OpenAI Gym [Brockman et al., 2016] provides the environments. NumPy efficiently and compactly processes arrays of floating-point values, such as returned as observations by many Gym environment. PyTorch (torch) builds and trains neural networks. The two standard library packages, random and collections, are used to sample actions and to instantiate an efficient store of experiences.

We now define a simple Python class that represents a recurrent neural network. In Section 2.5.4, we used Sequential, provided by PyTorch, to very compactly make a neural network. Unfortunately, this class only supports feed-forward neural networks. Recurrent ones require a small amount of special-casing, that goes beyond the abilities of Sequential.

```
class RNN(torch.nn.Module):
    def __init__(self, state_shape, hidden, num_actions):
        super().__init__()

        # Create the LSTM and output layers
        self.hidden = hidden
        self.lstm = torch.nn.LSTM(state_shape, hidden)
        self.fc = torch.nn.Linear(hidden, num_actions)

    def forward(self, states):
        # Pass the sequences of states through the LSTM
        hidden = torch.zeros(1, states.shape[1], self.hidden)
        x, _ = self.lstm(states, (hidden, hidden))
        x = x[-1]  # Only the last output of the LSTM matters
        x = torch.relu(x)
```

```
        x = self.fc(x)

        return x
```

The recurrent neural network contains *hidden* LSTM units, that are then connected to *num_actions* outputs. The constructor creates the LSTM units (between the input and the hidden layer) and the fully-connected layer, between the hidden layer and the output. The forward method is called when the network has to predict values. Intuitively, it passes every element of the input to the recurrent LSTM layer. For each input element, the LSTM layer produces a *candidate output*. We take the last candidate output (x[-1]), and pass it through the fully-connected layer to produce the actual output of the layer. states is a 3-dimensional tensor of floating-point values, whose dimensions are: number of input elements per input line, number of input lines, number of floats per input. If we want to predict the Q-Values for 32 states, with each state being a sequence of 5 observations (so, the agent gives the neural network the current observation and 4 previous observations), states will have a shape of $(5, 32, hidden)$, with $hidden = 128$ in our example (a good starting number for most environments). We now proceed to instantiate the environment, and the neural network:

```
HORIZON = 5

# Environment
env = gym.make('LunarLander-v2')
state_n = env.observation_space.shape[0]
action_n = env.action_space.n

# Neural network that stores Q-Values
qvalues = RNN(state_n, 128, action_n)
optim = torch.optim.Adam(qvalues.parameters(), lr=0.001)
loss = torch.nn.MSELoss()

# Experience buffer
experiences = collections.deque([], 5000)
```

HORIZON is the number of observations given to the neural network when predicting Q-Values. The larger this number is, the longer-term memory the agent will have. However, increasing HORIZON also makes the learning problem much harder, as more data has to be made sense of by the network. We can now

implement the general training loop of the agent, that contains its acting phase and learning phase:

```
while True:
    state = env.reset()
    done = False
    cumulative_ret = 0.0
    seen_states = [np.zeros_like(state)] * (HORIZON - 1) + [state]

    while not done:
        # Select action
        # [...]

        # Execute the action and store the experience
        next_state, reward, done, _ = env.step(action)
        experiences.append((seen_states, action, reward, next_state, done))

        # Learn from experiences
        # [...]

        # Move to the next time-step
        state = next_state
        seen_states = seen_states[1:] + [state]
        cumulative_ret += reward
```

Compared to the code presented in Section 2.3.2, the main change is the introduction of seen_states, a list of past observations maintained by the agent. This list is given to the neural network for it to predict $Q\big((x_t, x_{t-1}, ..., x_{t-N}), a\big)$. At the beginning of each episode, this list is filled with zeros. As the episode progresses, the list is shifted, so that its length remains constant, and the latest observation is the last element of the list:

```
# Select action
qv = qvalues(torch.from_numpy(np.array(seen_states).reshape(HORIZON, 1, -1)))[0]
action = int(qv.argmax())

if np.random.random() < 0.1:
    action = np.random.randint(4)
```

It is a bit unfortunate that PyTorch and NumPy do not follow the same conventions regarding sequences. The first line of the snippet above transforms seen_states, a list of past observations, represented by NumPy arrays, to a 3-dimensional PyTorch tensor of dimensions: horizon, 1 (we predict Q-Values for only one "state" when selecting an action), number of floats in an observation. Once that data has been shuffled around properly, in can be passed to the neural network (qvalues), that returns the Q-Values we use to select an action with ε-Greedy. This code snippet is not much more complicated than with Tabular Q-Learning (see Section 2.3.2). The complicated part is learning, that we divide in two steps: preparing the input-output pairs on which the network will be trained, and performing the actual parameter optimization of the network.

```
# Sample experiences and make PyTorch arrays from them
batch = random.choices(list(experiences), k=32)
QN = torch.arange(len(batch))

s = torch.from_numpy(np.array([e[0] for e in batch]).swapaxes(0, 1))
a = torch.tensor([e[1] for e in batch]).long()
r = torch.tensor([e[2] for e in batch])
sn = torch.from_numpy(np.array([e[0][1:] + [e[3]] for e in batch]).swapaxes(0, 1))
dones = torch.tensor([e[4] for e in batch]).float()

# y = current Q-Values, qv = updated Q-Values
y = qvalues(s)
qv = y.detach().clone()
qv[QN, a] = r + 0.99 * (1. - dones) * qvalues(sn).detach().max(1)[0]

# Make the network minimize the difference between y and qv
optim.zero_grad()
l = loss(y, qv)
l.backward()
optim.step()
```

Most of the boilerplate consists of sampling and processing experiences, returned as a list of $(x..., a, r, x', d)$ tuples, with $x...$ a list of HORIZON observations and d a boolean set to True when an episode finishes with the experience. This list of tuples has to be morphed into a set of PyTorch tensor, one for all the observations, one for all the actions, etc. Once these tensors are ready, the

second step computes qv, updated Q-Values, using the Q-Learning equation (see Equation 2.2). Here is one place where libraries like PyTorch and NumPy shine: all the tensors that appear in the equation, that is, a, r, dones and sn, contain data coming from many experiences. PyTorch automatically iterates over every entry of the tensors it performs operations on, which allows the developer to write one single compact expression, closely resembling the Q-Learning equation, and have it applied to every state and action corresponding to the experiences sampled above.

The last step consists of optimizing the neural network, so that it better approximates the Q-Function. The 4 last lines of the snippet above do that, and are part of standard PyTorch boilerplate. Basically, the difference between the currently-predicted Q-Values and updated Q-Values is computed, then the neural network is told to minimize this difference. An important note is that we perform *a single gradient step*, with a small learning rate of 0.001. This makes the neural network slowly follow the updated Q-Values. This slow learning is necessary for stable learning, and has the nice benefit of removing the need for the α learning rate of Equation 2.2.

This completes our implementation of a simple recurrent neural-network based agent. It is able to learn reasonably well in a few MDPs and POMDPs available in the OpenAI Gym. In the Appendix of this thesis, we provide a slightly larger version of this agent, that is also able to use a feed-forward neural network for settings that are known to be fully-observable. With partial-observability now discussed, we move on to the other challenge we consider in this chapter, complex long-running tasks.

4.2 Addressing Complex Tasks

After having explored the challenges of partial observability, we now consider a different challenge in Reinforcement Learning, complex tasks. By complex tasks, we refer to tasks, such as assembling a car, that take many time-step to execute, require precision, but also exhibit some structure and repetition. In the case of making a car, thousands of bolts have to be tightened. Learning to assemble a car becomes much easier when the agent is able to learn to tighten a bolt once, and reuse this skill a thousand times, instead of having to learn to tighten a bolt a thousand times from scratch.

Complex tasks may seem completely unrelated to partial observability, and were indeed so until recently. In Steckelmacher et al. [2017] and Section 4.3, we

show that the most common way of addressing complex tasks is also a strong basis on which to build Reinforcement Learning algorithms capable of working in various partially-observable environments. As such, we review in this section how complex tasks are addressed in Reinforcement Learning, and then combine both the complex-task and partially-observable settings in Section 4.3.

4.2.1 Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning consists of exploiting the structure of the task to be learned by the agent. When the task is repetitive, or the state-space contains repetitions, sample-efficiency and generalization can be improved by decomposing the task into *sub-tasks*, such as opening a door or leaving a room. Two main approaches exist:

Hierarchical Value Functions

Based on Q-Learning, the MAXQ algorithm [Dietterich, 1999, 2000] decomposes the Q-Function into hierarchy of nodes, that allow to compute the expected return of performing a subtask, and the expected return expected once the subtask is finished. Equations allow to learn those two separate quantities, for every subtask in the task graph, so that the agent is able to learn *which subtask to perform when*.

Hierarchical Policies

Except few exceptions, such as Kulkarni et al. [2016] who propose to integrate intrinsic motivation with a hierarchical value function, most current work in Hierarchical RL focuses on hierarchies of policies. The Options framework [Sutton et al., 1999] decomposes a task into a set of subtasks. Each subtask ω, or *option*, is represented by a policy $\pi_\omega(s) \in A$, an initiation set $I_\omega \in S$, and a termination function $\beta(s) \in \mathcal{R}$. The policy defines the behavior of the agent when performing the subtask. The initiation set identifies the set of task where it is valid to start the option. The termination function gives the probability, in each state, to terminate the option. When an option terminates, the *top-level policy* $\hat{\pi}(s) \in O$ observes the current state, and chooses the next option to execute. Usually, the options are fixed, provided by the system designer. Several algorithms, such as the Option-Critic, allow to learn the policies of options [Bacon et al., 2017]. These algorithms still assume that the number of desired options is provided to the agent, along with their initiation sets, and often also a reward function. Learning *how many*

and which options are required for a task can be done by observing which states appear to be bottlenecks [Stolle and Precup, 2002], or by identifying simple *skills* from demonstrations [Konidaris and Barto, 2009; Konidaris et al., 2012].

4.2.2 To Learn or not to Learn Option Policies

As part of our research, we implemented Options in many agents and settings, and read a large amount of literature on the subject. This led us to observe a few trends in the Options literature, that we would like to discuss now. This section is more philosophical than the other ones, and reflects our experience with Options. The main questions that arises with options are the following:

- Is it acceptable that the designed provides the option policies, and the agent only learns the top-level policy?
- Does learning both the option policies and top-level policy lead to increases in sample-efficiency?

These questions seem to divide the community. A large amount of literature focuses on pre-defined options. Notable work includes learning the value of an option even if another action is being executed [Sutton et al., 1998], allowing options to observe the state of the environment using their own option-specific encoding [Jonsson and Barto, 2000], learning models of options (for planning) given a reward function that can change at any time [Yao et al., 2014], or learning the top-level policy using options that execute for a long time (sample-efficient learning), but taking into account that the deployed policy will have access to much more fine-grained and short options [Harutyunyan et al., 2018].

An equally large amount of literature considers that providing the option policies is too much domain knowledge, and should be replaced with learning the option policies. The Option-Critic architecture allows to learn the top-level policy, option policies and termination functions at once [Bacon et al., 2017]. Better guidance about what the options should do is introduced by Harb et al. [2018]. Tessler et al. [2016] show that learning options allows to solve a 3D task in Minecraft.

Despite its recent successes, learning option policies exhibits two limitations:

1. Option-specific reward functions must often be provided, so that the options know what they have to achieve, or:

2. All the policies of all the options tend to converge to the same policy, that simply solves the overall task.

The second point is particularly interesting to us, as it leads to the intuition that learning N options may become equal to learning the complete task N times. We follow this intuition and extend it by reminding the reader that the agent executes actions in a Markov Decision Process. We then make the following observations:

1. The optimal policy maps every state to *one* optimal action. There is no need to have alternate policies that do different things in the same state.
2. The reward function depends only on the current state. Per-option reward functions can therefore trivially be combined into a single task-wide reward function. This means that decomposing the reward function into subtasks provides no benefit.
3. Learning options also requires their termination function to be learned, which, in our opinion, makes the learning problem strictly more difficult than learning a flat policy.
4. We observed that learning good option policies is difficult when the top-level policy is bad. Learning a good top-level policy is impossible when the option policies are bad (or change). In fact, having two layers of interacting policies that learn *at the same time* fits the definition of a multi-agent system [Littman, 1994], for which specific algorithms compatible with multi-agent learning are required [Bu et al., 2008]. These algorithms are outside the scope of this thesis.

We therefore argue that *learning top-level and option policies at the same time does not improve sample-efficiency*, and only distracts the agent. Obtaining option policies can still be important for explainability [Andrychowicz et al., 2017; Tessler et al., 2016], but does not align with our goal of increasing sample-efficiency. To be complete, we point out that learning options that observe the state differently from the top-level policy [Jonsson and Barto, 2000; Konidaris and Barto, 2007] may allow sample-efficiency improvements, but in the general case, *learning a flat policy with a good function approximator that generalizes well* is, in our opinion, the best approach. The use of options is still justified, though, in settings where the option policies are provided. We also now introduce another setting where learning options is beneficial, as the presence of options allows partially-observable tasks to be solved.

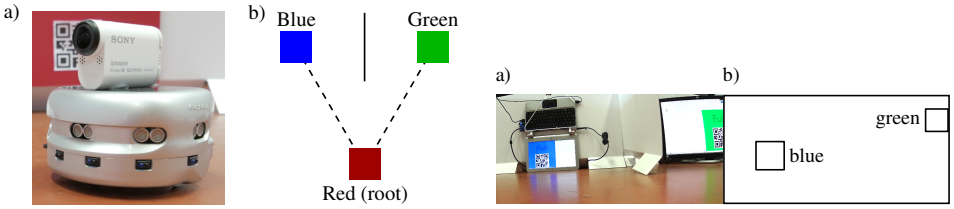

Figure 4.1: *Left:* Robotic object gathering task. a) Khepera III, the two-wheeled robot used in the experiments. b) The robot has to gather objects from two terminals separated by a wall, and to bring them to the root. *Right:* Observations of the Khepera robot. a) Color image from the camera. b) Color blobs detected by the vision system, as observed by the robot. QR-codes can only be decoded from a few centimeters away.

4.3 Complex Partially-Observable Tasks

After having discussed the challenges of partial observability in Section 4.1, and complex long-running tasks in Section 4.2, we now introduce our main contribution, an Options-based formalism that allows partially-observable complex long-running tasks to be learned efficiently. Our contribution is simple and elegant, nicely integrates in the Options framework, is easy to implement (we discuss two implementation directions in Sections 4.4.1 and 4.4.2), and leads to high sample-efficiency.

4.3.1 Gathering Objects from Terminals

We propose a method that allows to solve complex partially-observable tasks that can be decomposed into a set of fully-observable sub-tasks. For instance, a robot with first-person sensors may be able to avoid obstacles, open doors or manipulate objects even if its precise location in the building is not observed. We now describe such an environment, on which our robotic experiments of Section 4.5.2 are based.

A Khepera III robot[1] has to gather objects from two terminals separated by a wall, and to bring them to the root (see Figure 4.1). Objects have to be gathered one by one from a terminal until it becomes empty, which requires many journeys between the root and a terminal. When a terminal is emptied, the other one is automatically refilled. The robot therefore has to alternatively gather objects from both terminals, and the episode finishes after the terminals have been emptied

some random number of times. The root is colored in red and marked by a paper QR-code encoding 1. Each terminal has a screen displaying its color and a dynamic QR-code (1 when full, 2 when empty). Because the robot cannot read QR-codes from far away, the state of a terminal cannot be observed from the root, where the agent has to decide to which terminal it will go. This makes the environment partially observable, and requires the robot to remember which terminal was last visited, and whether it was full or empty.

The robot is able to control the speed of its two wheels, by defining a target velocity setpoint for each individual wheels. Once the target velocity is defined, the wheel quickly accelerates or slows down to reach it, then maintains that speed regardless of imperfections in the floor, battery voltage, etc. A wireless camera mounted on top of the robot detects bright color blobs in its field of view, and can read nearby QR-codes. Such low-level actions (setting speeds, instead of planning paths for instance) and observations, combined with a complicated task, motivate the use of hierarchical reinforcement learning. Designer-provided options allow the robot to move towards the largest red, green or blue blob in its field of view. The options terminate as soon as a QR-code is in front of the camera and close enough to be read. The robot has to learn a policy over options that solves the task.

4.3.2 Related Work

Several solutions to the combined challenge of partial observability and complex tasks have already been proposed in the literature.

In *planning*, options are already used to increase the planning horizon under uncertainty [He et al., 2011; Lim et al., 2011] and to construct policies robust to partial observability [Omidshafiei et al., 2017], but *learning* algorithms for hierarchical partially observable tasks are not yet ideal. HQ-Learning decomposes a task into a *sequence* of fully-observable subtasks [Wiering and Schmidhuber, 1997], which precludes cyclic tasks from being solved. Using recurrent neural networks in options and for the top-level policy [Sridharan et al., 2010] addresses both challenges, but brings in the design complexity of RNNs [Józefowicz et al., 2015; Angeline et al., 1994; Mikolov et al., 2014], and the challenges we discuss in Section 4.1.3. RNNs also have limitations regarding long time horizons, as their memory decays over time [Hochreiter and Schmidhuber, 1997], which makes them inapplicable to long-running tasks.

[1] http://www.k-team.com/mobile-robotics-products/old-products/khepera-iii

In her PhD thesis, Precup (2000b, page 126) suggests that options may already be close to addressing partial observability, thus removing the need for more complicated solutions. In this chapter, we prove this intuition correct by:

1. Showing that standard options do not suffice in POMDPs;
2. Introducing Option-Observation Initiation Sets (OOIs), that make the initiation sets of options conditional on the previously-executed option;
3. Proving that OOIs make options at least as expressive as Finite State Controllers (Section 4.3.4), thus able to tackle challenging POMDPs.

In contrast to existing HRL algorithms for POMDPs Wiering and Schmidhuber [1997]; Theocharous [2002]; Sridharan et al. [2010], OOIs handle repetitive tasks, do not restrict the action set available to sub-tasks, and keep the top-level and option policies memoryless. A wide range of robotic and simulated experiments in Section 4.5 confirm that OOIs allow partially observable tasks to be solved optimally, demonstrate that OOIs are much more sample-efficient than a recurrent neural network over options, and illustrate the flexibility of OOIs regarding the amount of domain knowledge available at design time. In Section 4.5.4, we demonstrate the robustness of OOIs to sub-optimal option sets. While it is generally accepted that the designer provides the options and their initiation sets, we show in Section 4.5.3 that random initiation sets, combined with learned option policies and termination functions, allow OOIs to be used without any domain knowledge.

4.3.3 Option-Observation Initiation Sets

Descriptions of partially observable tasks in natural language often contain allusions at sub-tasks that must be sequenced or cycled through, possibly with branches. This is easily mapped to a policy over options (learned by the agent) and sets of options that may or may not follow each other.

A good memory-based policy for our motivating example, where the agent has to bring objects from two terminals to the root (see Section 4.3.1), can be described as "go to the green terminal, then go to the root, then go back to the green terminal if it was full, to the blue terminal otherwise", and symmetrically so for the blue terminal. This sequence of sub-tasks, that contains a condition, is easily translated to a set of options. Two options, ω_{GF} and ω_{GE}, sharing a single

policy, go from the green terminal to the root (using low-level motor actions). ω_{GF} is executed when the terminal is full, ω_{GE} when it is empty. At the root, the option that goes back to the green terminal can only follow ω_{GF}, not ω_{GE}. When the green terminal is empty, going back to it is therefore forbidden, which forces the agent to switch to the blue terminal when the green one is empty.

We now formally define our main contribution, Option-Observation Initiation Sets (OOIs), that allow to describe which options may follow which ones. We define the initiation set I_ω of option ω so that the set $\mathcal{O}_t$ of options available at time t depends on the observation x_t and previously-executed option ω_{t-1}:

$$\begin{aligned} I_\omega &\subseteq \Omega \times (O \cup \{\emptyset\}) \\ \mathcal{O}_t &\equiv \{\omega \in O : (x_t, \omega_{t-1}) \in I_\omega\} \end{aligned} \tag{4.1}$$

with $\omega_0 = \emptyset$, Ω the set of observations and O the set of options. $\mathcal{O}_t$ allows the agent to condition the option selected at time t on the one that has just terminated, even if the top-level policy does not observe ω_{t-1}. The top-level and option policies remain memoryless. Not having to observe ω_{t-1} keeps the observation space of the top-level policy small, instead of extending it to $\Omega \times O$, without impairing the representational power of OOIs, as shown in the next sub-section.

4.3.4 OOIs Make Options as Expressive as FSCs

Finite State Controllers (FSCs) are commonly used in POMDPs. An FSC $\langle \mathcal{N}, \psi, \eta, \eta^0 \rangle$ is defined by a finite set $\mathcal{N}$ of nodes, an action function $\psi(n_t, a_t) \in [0, 1]$ that maps nodes to a probability distribution over actions, a successor function $\eta(n_{t-1}, x_t, n_t) \in [0, 1]$ that maps nodes and observations to a probability distribution over next nodes, and an initial function $\eta^0(x_1, n_1) \in [0, 1]$ that maps initial observations to nodes Meuleau et al. [1999].

At the first time-step, the agent observes x_1 and activates a node n_1 by sampling from $\eta^0(x_1, \cdot)$. An action is performed by sampling from $\psi(n_1, \cdot)$. At each time-step t, a node n_t is sampled from $\eta(n_{t-1}, x_t, \cdot)$, then an action a_t is sampled from $\psi(n_t, \cdot)$. FSCs allow the agent to select actions according to the entire history of past observations Meuleau et al. [1999], which has been shown to be one of the best approaches for POMDPs Lin and Mitchell [1992]. By proving that options with OOIs are as expressive as FSCs, we provide a lower bound on the expressiveness of OOIs and ensure that they are applicable to a wide range of POMDPs.

Theorem 1. *OOIs allow options to represent any policy that can be expressed using a Finite State Controller.*

Proof. The reduction from any FSC to options requires one option $\langle n'_{t-1}, n_t \rangle$ per ordered pair of nodes in the FSC, and one option $\langle \emptyset, n_1 \rangle$ per node in the FSC. Assuming that $n_0 = \emptyset$ and $\eta(\emptyset, x_1, \cdot) = \eta^0(x_1, \cdot)$, the options are defined by:

$$\beta_{\langle n'_{t-1}, n_t \rangle}(x_t) = 1 \tag{4.2}$$

$$\pi_{\langle n'_{t-1}, n_t \rangle}(x_t, a_t) = \psi(n_t, a_t) \tag{4.3}$$

$$\mu(x_t, \langle n'_{t-1}, n_t \rangle) = \eta(n'_{t-1}, x_t, n_t) \tag{4.4}$$

$$I_{\langle \emptyset, n_1 \rangle} = \Omega \times \{\emptyset\}$$

$$I_{\langle n'_{t-1}, n_t \rangle} = \Omega \times \{\langle n'_{t-2}, n_{t-1} \rangle : n'_{t-1} = n_{t-1}\}$$

Each option corresponds to an edge of the FSC. Equation 4.2 ensures that every option stops after having emitted a single action, as the FSC takes one transition every time-step. Equation 4.3 maps the current option to the action emitted by the destination node of its corresponding FSC edge. We show that μ and $I_{\langle n'_{t-1}, n_t \rangle}$ implement $\eta(n_{t-1}, x_t, n_t)$, with $\omega_{t-1} = \langle n'_{t-2}, n_{t-1} \rangle$, by:

$$\mu(x_t, \langle n'_{t-1}, n_t \rangle) = \begin{cases} \eta(n_{t-1}, x_t, n_t) & \langle n'_{t-2}, n_{t-1} \rangle \in I_{\langle n'_{t-1}, n_t \rangle} \\ & \Leftrightarrow n'_{t-1} = n_{t-1} \\ 0 & \langle n'_{t-2}, n_{t-1} \rangle \notin I_{\langle n'_{t-1}, n_t \rangle} \\ & \Leftrightarrow n'_{t-1} \neq n_{t-1} \end{cases}$$

Because η maps nodes to nodes and μ selects options representing pairs of nodes, μ is extremely sparse and returns a value different from zero, $\eta(n_{t-1}, x_t, n_t)$, only when $\langle n'_{t-2}, n_{t-1} \rangle$ and $\langle n'_{t-1}, n_t \rangle$ agree on n_{t-1}. □

Our reduction uses options with trivial policies, that execute for a single time-step, which leads to a large amount of options to compensate. In practice, we expect to be able to express policies for real-world POMDPs with much less options than the number of states an FSC would require, as shown in our simulated (Section 4.5.3, 2 options) and robotic experiments (Section 4.5.2, 12 options). In addition to being sufficient, the next sub-section proves that OOIs are necessary for options to be as expressive as FSCs.

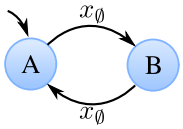

Figure 4.2: Two-nodes Finite State Controller that emits an infinite sequence ABAB... based on an uninformative observation $x_\emptyset$. This FSC cannot be expressed using options without OOIs.

4.3.5 Original Options are not as Expressive as FSCs

While options with regular initiation sets are able to express some memory-based policies [Sutton et al., 1999, page 7], the tiny but valid Finite State Controller presented in Figure 4.2 cannot be mapped to a set of options and a policy over options (without OOIs). This proves that options without OOIs are strictly less expressive than FSCs.

Theorem 2. *Options without OOIs are not as expressive as Finite State Controllers.*

Proof. Figure 4.2 shows a Finite State Controller that emits a sequence of alternating A's and B's, based on a constant uninformative observation $x_\emptyset$. This task requires memory because the observation does not provide any information about what was the last letter to be emitted, or which one must now be emitted. Options having memoryless policies, options executing for multiple time-steps are unable to represent the FSC exactly. A combination of options that execute for a single time-step cannot represent the FSC either, as the options framework is unable to represent memory-based policies with single-time-step options Sutton et al. [1999]. □

4.3.6 Partially-Observable Variable Action Set

The OOIs equation (4.1) defines the set of options available at time t as dependent on the current observation x_t and the option that terminated in $t-1$, ω_{t-1}. If we focus on the top-level policy, that chooses which option to execute given an

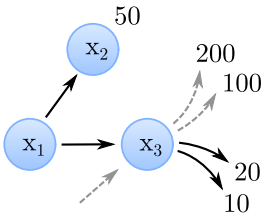

Figure 4.3: The set of actions available in x_3 depends on what the previous action was. When coming through the black arrow, only the black actions are available. When computing the value of x_3, a version of Q-Learning that is unaware of the variable action set would produce 200, the value of the best action emanating from x_3. In this setting, the best action in x_1 becomes going down, to x_3. This is however incorrect, as the 200 action is unavailable in x_3 when coming from x_1. Equation 4.5 must therefore be used, to ensure that the value of going down from x_1 is computed to be 20, leading the agent to prefer going to x_2.

observation, we observe that OOIs lead to a form of variable action set: the set of "actions" (here options) available at a given time-step is not fixed.

Most literature on variable action sets consider that the set $\mathcal{A}(x_t)$ of actions available at time t only depends on the current observation. When combined with Q-Learning, this leads to the following update:

$$Q_{k+1}(x,a) = Q_k(x,a) + \alpha\big(r + \gamma \max_{a' \in \mathcal{A}(x')} Q_k(x',a')\big)$$

The max part of the equation only depends on x', the next observation. The a' variable in the above equation is produced by an enumeration $A(x')$ of the actions available in state x'. In this setting, $A(x')$ only depends on x'. This means that, with variable action sets, the value of an observation $V(x')$ only depends on x'.

With OOIs, the situation is different. Because the set of "actions" (with OOIs, options) available at time t depends on the current observation *and the previous action*, $\mathcal{A}(x_t, a_{t-1})$ does not depend solely on the observation anymore. Figure 4.3 provides an example of a setting where properly computing $\mathcal{A}$ is important. The Q-Learning equation becomes:

$$Q_{k+1}(x,a) = Q_k(x,a) + \alpha\big(r + \gamma \max_{a' \in \mathcal{A}(x',a)} Q_k(x',a')\big) \tag{4.5}$$

The symbols x', a' and a now appear in the computation of the value of x'. The action a' is still produced by an enumeration, but the enumeration itself now depends on x' and a. The value of an observation therefore depends on the observation itself, *and the action that was executed before.*

We stress that OOIs still assume memoryless top-level and option policies, and we show in Section 4.3.4 that these memoryless policies, combined with OOIs, are able to represent memory-based policies. However, the fact that $V(x)$ depends on x and the action executed before it has two implications:

1. Policy Gradient methods, that we present in 4.4.1, do not use the V function, but instead estimate the value of an action in a state using Monte-Carlo returns. As such, Policy Gradient can be combined with OOIs with minimal effort, the only change required is the implementation of variable action sets.

2. Off-Policy value-based methods, that require $V(x)$ to be computed for some observations, need care. We point out that the SARSA algorithm is value-based, but uses the on-policy $\gamma Q_k(x', a')$ return estimate instead of the problematic $\gamma V(x')$ function. The value function arising from the use of OOIs is therefore challenging only to Q-Learning-family algorithms.

In Section 4.4.2, we present our implementation of BDPI with OOIs. Because BDPI uses off-policy critics, trained with Q-Learning, we have to maintain the off-policy nature of the critic as much as possible, while still adding the mandatory dependency of $V(x_{t+1})$ on a_t.

4.4 Implementing Option-Observation Initiation Sets

In Section 4.3.3, we introduce the formalism of Option-Observation Initiation Sets, that consists of defining the initiation set of options such that the set of options available at a given time-step depends on the current observation, and the option that has just finished. We now propose two implementations of OOIs. The first one, that we use in Section 4.5, is based on Policy Gradient. The second one builds on Bootstrapped Dual Policy Iteration, our sample-efficient algorithm introduced in Chapter 3. By providing two implementations, one policy-based and one value-based, we demonstrate that OOIs are applicable regardless of the family of the reinforcement learning algorithm used to learn the task.

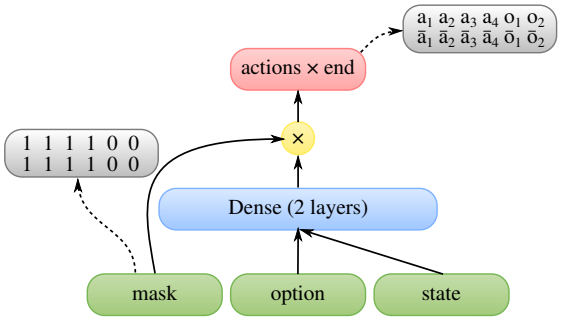

Figure 4.4: Neural architecture that allows Policy Gradient to be robust to disabled options. The mask forces the probabilities of disabled options to zeros, as part of the neural network, so that options can be directly sampled form the output of the policy, which ensures proper gradient propagation and stable learning.

4.4.1 Masked Policy Gradient

Our first implementation of OOIs is based on Policy Gradients, that we extend to be robust to disabled options. By learning a policy and no value function, and using a simple algorithm, we ensure that the performance of the agents we evaluate in Section 4.5 comes from the use of OOIs, and not some magical behavior of a more complicated reinforcement learning algorithm.

As discussed in Section 2.7, stochastic Policy Gradients is strictly on-policy, and any change to the probability distribution output by the policy makes learning unstable. This means that, in the top-level policy, we cannot just set the probability of disabled option to zero before sampling an option. We must ensure that the policy, in our case a neural network, directly produces a probability distribution compatible with the set of available options. A second requirement is that constraining the set of available options must not make the learning problem harder, and so must not increase the amount of learnable weights in the network. Our contribution therefore consists of adding a **mask** input to the network, as shown in Figure 4.4. The mask is element-wise multiplied with the candidate probabilities output by the learnable part of the network, to produce the actual output of the network. By setting **mask** elements to 0 or 1, the set of available options can be given to the network, while preserving the correct gradient flow, and without adding any learnable weight. A generalization of this mask to general *advice* is

presented by Plisnier et al. [2019a], and later developed into a multi-task Transfer Learning framework by Plisnier et al. [2019b].

More specifically, we propose a neural architecture inspired by the Option-Critic [Bacon et al., 2017], where the critic is replaced by Monte-Carlo returns, and a single neural network learns the policies and termination functions of all the options. Our neural network π takes three inputs and produces one output. The inputs are problem-specific observation features $\mathbf{x}$, the one-hot encoded current option $\boldsymbol{\omega}$ ($\boldsymbol{\omega} = \mathbf{0}$ when executing the top-level policy), and a mask, **mask**. The output $\mathbf{y}$ is the joint probability distribution over selecting actions or options (so that the same network can be used for the top-level and option policies), while terminating or continuing the current option:

$$\begin{aligned}
\mathbf{h_1} &= \tanh(\mathbf{W}_1[\mathbf{x}^T \boldsymbol{\omega}^T]^T + \mathbf{b}_1), \\
\hat{\mathbf{y}} &= \sigma(\mathbf{W}_2 \mathbf{h_1} + \mathbf{b}_2) \circ \mathbf{mask}, \\
\mathbf{y} &= \frac{\hat{\mathbf{y}}}{\mathbf{1}^T \hat{\mathbf{y}}},
\end{aligned}$$

with W_i and b_i the trainable weights and biases of layer i, σ the sigmoid function, and $\circ$ the element-wise product of two vectors. The fraction ensures that a valid probability distribution is produced by the network. The initiation sets of options are implemented using the **mask** input of the neural network, a vector of $2 \times (|A| + |O|)$ integers, the same dimension as the $\mathbf{y}$ output. When executing the top-level policy ($\boldsymbol{\omega} = \mathbf{0}$), the mask forces the probability of primitive actions to zero, preserves option ω_i according to I_{ω_i}, and prevents the top-level policy from terminating. When executing an option policy ($\boldsymbol{\omega} \neq \mathbf{0}$), the mask only allows primitive actions to be executed. For instance, if there are two options and three actions, $\mathbf{mask} = \begin{smallmatrix} end \\ cont \end{smallmatrix} \left(\begin{smallmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{smallmatrix} \right)$ when executing any of the options. When executing the top-level policy, $\mathbf{mask} = \begin{smallmatrix} end \\ cont \end{smallmatrix} \left(\begin{smallmatrix} 0 & 0 & 0 & 0 & 0 \\ a & b & 0 & 0 & 0 \end{smallmatrix} \right)$, with $a = 1$ if and only if the option that has just finished is in the initiation set of the first option, and $b = 1$ according to the same rule but for the second option. The neural network π is trained using Policy Gradient, with the following loss:

$$\mathcal{L}(\pi) = -\sum_{t=0}^{T} (\mathcal{R}_t - V(x_t, \omega_t)) \log(\pi(x_t, \omega_t, a_t))$$

with $a_t \sim \pi(x_t, \omega_t, \cdot)$ the action executed at time t. The return $\mathcal{R}_t = \sum_{\tau=t}^{T} \gamma^\tau r_\tau$, with $r_\tau = R(s_\tau, a_\tau, s_{\tau+1})$, is a simple discounted sum of future rewards, and ignores

changes of current option. This gives the agent information about the complete outcome of an action or option, by directly evaluating its flattened policy. A baseline $V(x_t, \omega_t)$ is used to reduce the variance of the $\mathcal{L}$ estimate Sutton et al. [2000]. $V(x_t, \omega_t)$ predicts the expected cumulative reward obtainable from x_t in option ω_t using a separate neural network, trained on the monte-carlo return obtained from x_t in ω_t.

4.4.2 BDPI with OOIs

Because the main arguments behind the use of OOIs are their simplicity and sample-efficiency, we also implement OOIs on top of BDPI, an algorithm that is several orders of magnitude more sample-efficient than Policy Gradient (see Chapter 3). Our implementation simply consists of ensuring that the actor of BDPI *acts* according to the set of available options, and that its critics *learn* Q-Values that take the set of available options into account. We refer the interested reader to Plisnier et al. [2019b] for a more complicated extension of BDPI, that implements the Actor-Advisor instead of simple masked options, and discusses the interaction between a BDPI actor, its critics, and advice.

General Architecture

The architecture BDPI with OOIs follows closely the Option-Critic and our Policy Gradient implementation of Section 4.4.1. A single actor learns a single policy, and a set of critics learn Q-Values. They all take as input an extended state: the current observation and one-hot encoding of the current option. The action set is extended as in Section 4.4.1, to allow a single policy to select options and actions, while terminating or continuing the current option. This means that both the actor and critics produce $2(|A| + |O|)$ outputs. We must therefore ensure that the actor acts according to the set of available options, and does not execute options in options or actions in the top-level policy, and that the Q-Values of the critics are consistent with the environment and OOIs.

Acting

BDPI being a fully off-policy algorithm, no special care is required when the probabilities output by the actor need to be modified before acting. As in Section 4.4.1, a mask is computed at acting time, that disables unavailable options, options

in options, and actions in the top-level policy. This mask is simply element-wise multiplied with the probabilities output by the actor, before an action or option is sampled. The mask is stored in the experiences collected by the agent, as it influences learning.

Training the actor

Contrary to Plisnier et al. [2019b], the actor learning rule is not modified. Because our mask only consists of zeros and ones, and as such only fully allows or forbids actions, the learning correction describe by Plisnier et al. [2019b] is not applicable nor necessary. Because the actor is not Policy Gradient-based, having some greedy policies recommend forbidden options, options in options or actions in the top-level policy, will cause the actor to be different from the (masked) behavior policy, which is not a problem for BDPI. The actor can therefore pursue the greedy policy of the critics, with any constrain applied solely at acting time.

Training the critics

While the critics need no special care in Plisnier et al. [2019b], they do in the case of BDPI with OOIs. As detailed in Section 4.3.6, the Q-Learning equation at the core of the BDPI critic update rule needs to be modified, as the value of observation x' depends on the action that was execute before reaching that observation.

We propose to make the BDPI critics *partially on-policy*. Following the notations used in Section 4.4.1, we extend the BDPI experience buffer to store $(x_t, a_t, r_t, x_{t+1}, m_{t+1})$ tuples. The mask m_{t+1} encodes which actions are available at time $t+1$. With the mask m_{t+1} in the experience buffer, we now modify the BDPI actor update rule to use the value function introduced in Section 4.3.6:

$$V(x_{t+1}, m_{t+1}) \equiv \min_{l=A,B} \max_{a' \in m_{t+1}} Q^l(x_{t+1}, a')$$

with Q^A and Q^B the two Q-Functions of a BDPI critic, x_{t+1} the observation for which the value is computed, and m_{t+1} the mask at time-step $t+1$, as stored in the experience at acting time. We denote the resulting critic learning rule as *partially on-policy* for the two following reasons:

1. Neither the action a_{t+1}, nor anything produced by the actor π, appears in the equation. This is the off-policy aspect of the formula, that does not rely on the knowledge of either the actor or the behavior policy.

2. The mask m_{t+1} depends on the action a_t executed by the behavior policy. There is therefore still an indirect dependency between a_t and the value $V(x_{t+1}, m_{t+1})$. This makes this version of BDPI not completely off-policy, but, interestingly, still fully off-actor (π does not directly or indirectly appear).

An intuition for the small amount of on-policyness required for BDPI with OOIs to learn in an environment that, while OOIs increase expressiveness, remains partially-observable, is provided by Perkins and Pendrith [2002]. They show that on-policy SARSA, and in some settings two-step Q-Learning, are able to find satisfactory memoryless policies in POMDPs, while Q-Learning fails to learn a good policy even if the optimal policy for a POMDP is memoryless.

We show in Section 4.5.5 that BDPI with OOIs is able to learn the optimal policy for our *Terminals* task, with a sample-efficiency that is significantly higher than recurrent neural networks or Policy Gradient with OOIs. This shows that the partially-on-policy nature of BDPI with OOIs does not impair its learning performance, and empirically outperforms several competing approaches at learning in POMDPs.

4.4.3 Tabular BDPI with OOIs

In Section 4.4.2, we detail how we implement Options with extended observations and actions, allowing a regular flat (Options-unaware) agent to learn a top-level and option policies. We then detail how to incorporate OOIs in that agent. In this section, we consider the Tabular BDPI implementation presented in Section 3.4, and extend it with Options and OOIs. We hope that the tangible implementation of BDPI with OOIs of this section would complement Section 4.4.2, and illustrate how OOIs can be implemented in practice.

Extended State and Action Space

We first consider the foundation of any Reinforcement Learning agent: its state and action space. With Options and OOIs, the state-space of the agent must be extended to also encode which option is currently executing, and the action-space

must be extended to allow the agent to execute options (in addition to primitive actions), and terminate options.

We introduce a new file, called terminals_policy.py in this example. It describes how many options the agent has access to, and what they do. The framework we present in this section is general, but we use as running example the *Terminals* environment described in Section 4.5.2.

```
import terminals

ENV = terminals.TerminalsEnv()
NUM_OPTIONS = 12
```

The above file is then imported in the main file, that contains the agent. Its ENV and NUM_OPTIONS attributes are used to prepare the extended state and action spaces:

```
import terminals_policy as options

def main():
    env = options.ENV
    average_cumulative_reward = 0.0

    # Number of discrete states and actions in the environment
    num_env_states = env.observation_space.shape[0]
    num_env_actions = env.action_space.n

    # Each environment state can be see in an option or the top-level policy
    num_states = num_env_states * (options.NUM_OPTIONS + 1)

    # Execute actions or options, while continuing or terminating
    num_actions = (num_env_actions + options.NUM_OPTIONS) * 2

    # Create the actor and critics for the extended state and action spaces
    critics = np.random.random((NUM_CRITICS, num_states, num_actions)) * 0.01
    actor = np.ones((num_states, num_actions)) / num_actions
```

The use of num_states and num_actions, as in Section 3.4, illustrates that our use of a flat policy, with extended state and action spaces, allows the Reinforcement Learning agent to be mostly agnostic to the existence of options. Any learning

algorithm can be combined with Options and OOIs, following our framework. We only need to modify the agent in two places: decode extended actions and map them back to environment actions (or changes in current option), and produce extended states from environment states.

Producing Extended States

We introduce a new function, that maps an environment state (in this example, an integer) to an extended state (also an integer), part of a larger state-space that encodes which action is currently executing. We assume that the options are identified with integers from 0 to NUM_OPTIONS - 1, with -1 representing the top-level policy.

```
def extend_state(s, current_option, num_env_states):
    return s + (current_option + 1) * num_env_states
```

Extending the state can be understood as follows: assume that the environment has 10 states, numbered from 0 to 9, and that the agent has access to 2 options. Extended states 0 to 9 represent environment states 0 to 9 while the top-level policy executes. Extended states 10 to 19 represent environment states 0 to 9 while option 0 executes, and extended states 20 to 29 encode the fact that option 1 is executing.

The environment state must be extended in two places in the agent: when an episode starts and the environment is reset (which produces an initial environment state), and after every time-step, when the effect of an action produces a new environment state:

```
# Loop over episodes
for i in range(EPISODES):
    done = False
    cumulative_reward = 0.0

    current_option = -1
    previous_option = -1

    env_state = de_onehot(env.reset())
    state = extend_state(env_state, current_option, num_env_states)
```

At the beginning of an episode, the agent executes the top-level policy, and there is not yet any previous option (previous_option set to -1). Information

about which option is current, and what was the previous option, will be used to compute OOIs and condition which option the agent can execute after which one. In the code above, de_onehot is a small function that takes a one-hot encoded vector of floating-point values (0, 0, 1, 0 for instance) and maps it back to an integer (2 in this case).

After every time-step, the environment produces a next state, that is extended before being added to the experience buffer. We omit this code in this section, as it will be presented in the next sub-section when we extend the action space.

Executing Extended Actions

The actor of the agent produces a probability distribution over extended actions, with each extended action allowing the agent to execute a primitive action or option, while terminating or continuing the current option. After the actor produces a probability distribution, from which an extended action is sampled, the extended action is decoded before being passed to the environment:

```
action = int(np.random.choice(range(num_actions), p=probas))

# High action numbers terminate the current option
terminates = action // (num_actions // 2)
action = action % (num_actions // 2)  # An action or an option

if action < num_env_actions:
    # Execute an environment action while terminating or continuing
    next_env_state, reward, done, _ = env.step(action)
    next_env_state = de_onehot(next_env_state)

    if terminates:
        # Terminate the current option
        previous_option = current_option
        current_option = -1
else:
    # Execute an option
    next_env_state, reward, done, _ = env_state, 0.0, False, None
    current_option = action - num_env_actions
```

The code above considers three cases: the extended action identifies an option to execute, or a primitive action. When the extended action encodes a primitive

action, it is executed in the environment, then the current option is terminated or not. Some cases are not covered by the code above, that for instance considers that options are only selected when the top-level policy is executing (the *option* case does not consider terminating an option). We now detail how a *mask* allows to constrain which extended actions are available to the agent at every time-step.

The Mask

Options can not recursively execute other options, and the top-level policy can not execute actions or terminate. In order to implement theses constraints, we propose to use a mask, a vector of num_actions numbers that are either 1 (for allowed extended actions) or 0 (for forbidden ones). Every time-step, before an action is selected from the probability distribution output by the actor, the mask is computed and element-wise multiplied with the probabilities, to force forbidden actions to have a probability of zero:

```
# At first, every action is allowed
mask = np.ones((num_actions,), dtype=np.float32)

if current_option == -1:
    # Top-level policy, cannot terminate nor execute actions.
    mask.fill(0.0)

    if previous_option in options.CAN_BE_FOLLOWED:
        for option in options.CAN_BE_FOLLOWED[previous_option]:
            mask[num_env_actions + option] = 1.0
    else:
        # No constrain on which option can follow previous_option
        mask[num_env_actions:num_actions//2] = 1.0
else:
    # Option policy, disable recursively executing options
    mask[num_env_actions:num_actions//2] = 0.0      # Recurse while continuing
    mask[num_actions//2 + num_env_actions:] = 0.0   # Recurse while terminating
```

The code above sets elements of the mask to 0 or 1 (more details in Section 4.4.1). CAN_BE_FOLLOWED is a dictionary that will be introduced in the next sub-section, and that allows to define which option can be followed by which ones, thereby implementing our OOIs. Once the mask is computed, it can be element-wise multiplied with the probabilities output by the actor. This is followed by

a normalization, to ensure that the constrained probabilities still form a valid probability distribution:

```
probas = probas * mask
probas /= probas.sum()

action = int(np.random.choice(range(num_actions), p=probas))
```

Implementing OOIs with the Mask

To allow specifying which option may be followed by which ones, we go back to terminals_policy.py and introduce a CAN_BE_FOLLOWED dictionary, that maps a previous option index to a list of option indexes that can follow it:

```
CAN_BE_FOLLOWED = {
        -1: [0, 1, 2, 3],
        0: [4, 5],
        1: [6, 7],
        2: [8, 9],
        # ...
}
```

The code above tells the agent that, at the beginning of the episode, when there is no previous option, only options 0 to 3 can be executed. Then, option 0 can be followed by 4 and 5, and so on. The complete source code of this file, in Appendix, details the name and meaning of every option. Section 4.5.2 also explains why this agent uses 12 options, and what they do.

Once CAN_BE_FOLLOWED is defined, the code shown in the previous subsection can use it to compute masks every time-step. Two components of the agent still have to be modified to allow learning: in the experience buffer, every experience must know the mast *of the next time-step*; and the critic update rule, a variant of Q-Learning, must be made partially on-policy as described in Section 4.4.2. We begin with the mask of the next time-step:

```
# Put the mask in the next_mask location of the previous experience
if len(transitions) > 0:
        transitions[-1][4] = mask

# [...] (choose and execute action)
```

```
# Add an experience to the experience buffer with a temporary all-ones next_mask
next_state = extend_state(next_env_state, current_option, num_env_states)
transitions.append([state, action, reward, next_state, np.ones_like(mask), done])
```

For the critic update rule, we refer the reader to Section 3.4, and only present in this section the code that updates the Q-Value of one action in one state, as sampled from the experience buffer. This Q-Value needs to consider the reward obtained at time t, along with the value of the next state, $V(s_{t+1})$. As detailed in Section 4.4.2, $V(s_{t+1})$ must consider the mask at $t+1$, to only consider allowed actions when computing its maximum over Q-Values:

```
for (s, a, r, ns, nm, t) in batch:
    # Only consider actions available at t+1 when computing V(s_{t+1})
    vnext = qtable[ns][nm > 0.0].max()

    td_error = r + (GAMMA * vnext if not t else 0.0) - qtable[s, a]
    qtable[s, a] += LEARNING_RATE * td_error
```

The complete source code of Tabular BDPI with Options and OOIs is available in the Appendix, and allows terminals_policy.py to provide a policy function, that defines the option policy. Moreover, any state (in any policy, or the top-level policy) for which policy returns None, the actor of the agent selects an action and will learn from its outcome. Otherwise, the action selected by the policy is executed. This is a general way to partially define the policy of an agent (only in some states, only in some options), and letting the actor learn a good policy for all the other states.

With the implementation of Tabular BDPI with Options and OOIs complete, we now move on to the experiments, and validate our OOIs on various partially-observable environments. In our experiments, we use a Policy-Gradient based algorithm, that allows for continuous state-spaces, that is also available in the Appendix.

4.5 Experiments

The experiments in this section illustrate how OOIs allow agents to perform optimally in environments where options without OOIs fail. Section 4.5.2 shows that OOIs allow the agent to learn an expert-level policy for our motivating example

(Section 4.3.1). Section 4.5.3 shows that the top-level and option policies required by a repetitive task can be learned, and that learning option policies allow the agent to leverage random OOIs, thereby removing the need for designing them. In Section 4.5.4, we progressively reduce the amount of options available to the agent, and demonstrate how OOIs still allow good memory-based policies to emerge when a sub-optimal amount of options are used.

All our results are averaged over 20 runs, with standard deviation represented by the light regions in the figures. The source code, raw experimental data, run scripts, and plotting scripts of our experiments, along with a detailed description of our robotic setup, are available in the appendix. A video detailing our robotic experiment is available at https://youtu.be/VprJZEOD5NE.

4.5.1 Comparison with LSTM over Options

In order to provide a complete evaluation of OOIs, a variant of the π and V networks of Section 4.4.1, where the hidden layer is replaced with a layer of 20 LSTM units Hochreiter and Schmidhuber [1997]; Sridharan et al. [2010], is also evaluated on every task. We use 20 units as this leads to the best results in our experiments, which ensures a fair comparison of LSTM against OOIs. In all experiments, the LSTM agents are provided the same set of options as the agent with OOIs. Not providing any option, or less options, leads to worse results. Options allow the LSTM network to focus on important observations, and reduces the time horizon to be considered. Shorter time horizons have been shown to be beneficial to LSTM Bakker [2001].

Despite our efforts, LSTM over options only manages to learn good policies in our robotic experiment (see Section 4.5.2), and requires more than twice the amount of episodes as OOIs to do so. In our repetitive task, dozens of repetitions seem to confuse the network, that quickly diverges from any good policy it may learn (see Section 4.5.3). On TreeMaze, a much more complex version of the T-maze task, originally used to benchmark LSTM agents Bakker [2001], the LSTM agent learns the optimal policy after more than 100K episodes (not shown on the figures). These results illustrate how learning with recurrent neural networks is sometimes difficult, and how OOIs allow to reliably obtain good results, with minimal engineering effort.

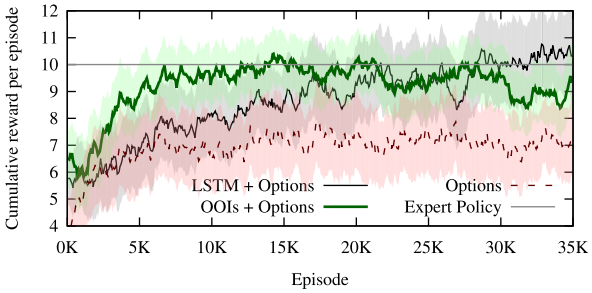

Figure 4.5: Cumulative reward per episode obtained on our object gathering task, with OOIs, without OOIs, and using an LSTM over options. OOIs learns an expert-level policy much quicker than an LSTM over options. The LSTM curve flattens-out (with high variance) after about 30K episodes.

4.5.2 Object Gathering

The first experiment illustrates how OOIs allow an expert-level policy to be learned for a complex robotic partially-observable repetitive task. The experiment takes place in the environment described in Section 4.3.1. A robot has to gather objects one by one from two terminals, green and blue, and bring them back to the root location. Because our actual robot has no effector, it navigates between the root and the terminals, but only pretends to move objects. The agent receives a reward of +2 when it reaches a full terminal, -2 when the terminal is empty. At the beginning of the episode, each terminal contains 2 to 4 objects, this amount being selected randomly for each terminal. When the agent goes to an empty terminal, the other one is re-filled with 2 to 4 objects. The episode ends after 2 or 3 emptyings (combined across both terminals). Whether a terminal is full or empty is observed by the agent only when it is at the terminal. The agent therefore has to remember information acquired at terminals in order to properly choose, at the root, to which terminal it will go.

The agent has access to 12 memoryless options that go to red ($\omega_{R1..R4}$), green ($\omega_{G1..G4}$) or blue objects ($\omega_{B1..B4}$), and terminate when the agent is close enough to them to read a QR-code displayed on them. The initiation set of $\omega_{R1,R2}$ is $\omega_{G1..G4}$, of $\omega_{R3,R4}$ is $\omega_{B1..B4}$, and of ω_{G_i,B_i} is ω_{R_i} $\forall i = 1..4$. This description of the options and their OOIs is purposefully uninformative, and illustrates how little information the agent has about the task. The option set used in this experiment

is also richer than the simple example of Section 4.3.3, so that the solution of the problem, not going back to an empty terminal, is not encoded in OOIs but must be learned by the agent.

Agents with and without OOIs learn top-level policies over these options. We compare them to a *fixed* agent, using an expert top-level policy that interprets the options as follows: $\omega_{R1..R4}$ go to the root from a full/empty green/blue terminal (and are selected accordingly at the terminals depending on the QR-code displayed on them), while $\omega_{G1..G4,B1..B4}$ go to the green/blue terminal from the root when the previous terminal was full/empty and green/blue. At the root, OOIs ensure that only one option amongst *go to green after a full green*, *go to green after an empty blue*, *go to blue after a full blue* and *go to blue after an empty green* is selected by the top-level policy: the one that corresponds to what color the last terminal was and whether it was full or empty. The agent goes to a terminal until it is empty, then switches to the other terminal, leading to an average cumulative reward of 10.[2]

When the top-level policy is learned, OOIs allow the task to be solved, as shown in Figure 4.5, while standard initiation sets do not allow the task to be learned. Because experiments on a robot are slow, we developed a small simulator for this task, and used it to produce Figure 4.5 after having successfully asserted its accuracy using two 1000-episodes runs on the actual robot. The agent learns to properly select options at the terminals, depending on the QR-code, and to output a proper distribution over options at the root, thereby matching our expert policy. The LSTM agent learns the policy too, but requires more than twice the amount of episodes to do so. The high variance displayed in Figure 4.5 comes from the varying amounts of objects in the terminals, and the random selection of how many times they have to be emptied.

Because fixed option policies are not always available, we now show that OOIs allow them to be learned at the same time as the top-level policy.

4.5.3 Modified DuplicatedInput

In some cases, a hierarchical reinforcement learning agent may not have been provided policies for several or any of its options. In this case, OOIs allow the agent to learn its top-level policy, the option policies and their termination functions. In this experiment, the agent has to learn its top-level and option policies to copy

[2] $\frac{2+3}{2} \times (-2 + \frac{2+4}{2} \times 2)$, 2 or 3 emptyings of terminals that contain 2 to 4 objects. Average confirmed experimentally from 1000 episodes using the policy, $p > 0.30$.

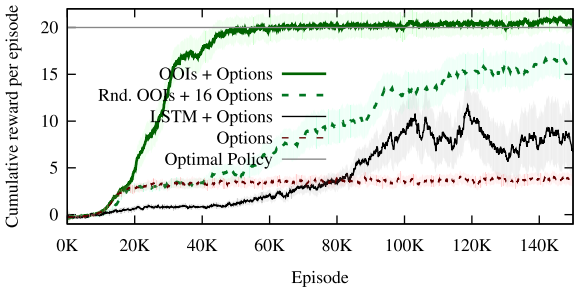

Figure 4.6: Cumulative reward per episode obtained on modified DuplicatedInput, with random or designed OOIs, without OOIs and using an LSTM over options. Despite our efforts, an LSTM over options repeatedly learns then forgets optimal policies, as shown by the high variance of its line.

characters from an input tape to an output tape, removing duplicate B's and D's (mapping ABBCCEDD to ABCCED for instance; B's and D's always appear in pairs). The agent only observes a single input character at a time, and can write at most one character to the output tape per time-step.

The input tape is a sequence of N symbols $x \in \Omega$, with $\Omega = \{A, B, C, D, E\}$ and N a random number between 20 and 30. The agent observes a single symbol $x_t \in \Omega$, read from the i-th position in the input sequence, and does not observe i. When $t = 1$, $i = 0$. There are 20 actions ($5 \times 2 \times 2$), each of them representing a symbol (5), whether it must be pushed onto the output tape (2), and whether i should be incremented or decremented (2). A reward of 1 is given for each correct symbol written to the output tape. The episode finishes with a reward of -0.5 when an incorrect symbol is written.

The agent has access to two options, ω_1 and ω_2. OOIs are designed so that ω_2 cannot follow itself, with no such restriction on ω_1. No reward shaping or hint about what each option should do is provided. The agent automatically discovers that ω_1 must copy the current character to the output, and that ω_2 must skip the character without copying it. It also learns the top-level policy, that selects ω_2 (skip) when observing B or D and ω_2 is allowed, ω_1 otherwise (copy).

Figure 4.6 shows that an agent with two options and OOIs learns the optimal policy for this task, while an agent with two options and only standard initiation sets ($I_\omega = \Omega \ \forall\omega$) fails to do so. The agent without OOIs only learns to copy characters and never skips any (having two options does not help it). This shows

that OOIs are necessary for learning this task, and allow to learn top-level and option policies suited to our repetitive partially observable task.

When the option policies are learned, the agent becomes able to adapt itself to random OOIs, thereby removing the need for designing them. For an agent with N options, each option has $\frac{N}{2}$ randomly-selected options in its initiation set, with the initiation sets re-sampled for each run. The agents learn how to leverage their option set, and achieve good results on average (16 options used in Figure 4.6, more options lead to better results). When looking at individual runs, random OOIs allow optimal policies to be learned, but several runs require more time than others to do so. This explains the high variance and noticeable steps shown in Figure 4.6.

The next section shows that an improperly-defined set of human-provided options, as may happen in design phase, still allows the agent to perform reasonably well. Combined with our results with random OOIs, this shows that OOIs can be tailored to the exact amount of domain knowledge available for a particular task.

4.5.4 TreeMaze

The optimal set of options and OOIs may be difficult to design. When the agent learns the option policies, the previous section demonstrates that random OOIs suffice. This experiment focuses on human-provided option policies, and shows that a sub-optimal set of options, arising from a mis-specification of the environment or normal trial-and-error in design phase, does not prevent agents with OOIs from learning reasonably good policies.

TreeMaze is our generalization of the T-maze environment Bakker [2001] to arbitrary heights. The agent starts at the root of the tree-like maze depicted in Figure 4.7, and has to reach the extremity of one of the 8 leaves. The leaf to be reached (the goal) is chosen uniformly randomly before each episode, and is indicated to the agent using 3 bits, observed one at a time during the first 3 time-steps. The agent receives no bit afterwards, and has to remember them in order to navigate to the goal. The agent observes its position in the current corridor (0 to 4) and the number of T junctions it has already crossed (0 to 3). A reward of -0.1 is given each time-step, +10 when reaching the goal. The episode finishes when the agent reaches any of the leaves. The optimal reward is 8.2.

We consider 14 options with predefined memoryless policies, several of them sharing the same policy, but encoding distinct states (among 14) of a 3-bit memory where some bits may be unknown. 6 partial-knowledge options ω_{0--}, ω_{1--}, ω_{00-}, ..., ω_{11-} go right then terminate. 8 full-knowledge options ω_{000}, ω_{001}, ..., ω_{111} go

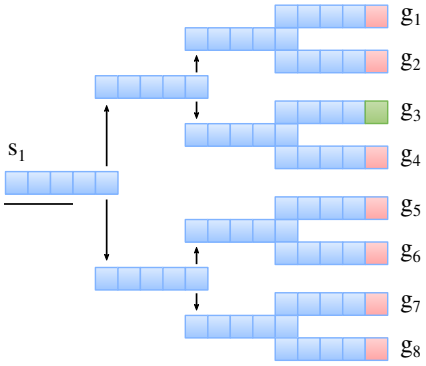

Figure 4.7: TreeMaze environment. The agent starts at s_1 and must go to one of the leaves. The leaf to be reached is indicated by 3 bits observed at time-steps 1, 2 and 3.

to their corresponding leaf. OOIs are defined so that any option may only be followed by itself, or one that represents a memory state where a single 0 or - has been flipped to 1. Five agents have to learn their top-level policy, which requires them to learn how to use the available options to remember to which leaf to go. The agents do not know the name or meaning of the options. Three agents have access to all 14 options (with, without OOIs, and LSTM). The agent with OOIs (8) only has access to full-knowledge options, and therefore cannot disambiguate unknown and 0 bits. The agent with OOIs (4) is restricted to options ω_{000}, ω_{010}, ω_{100} and ω_{110} and therefore cannot reach odd-numbered goals. The options of the (8) and (4) agents terminate in the first two cells of the first corridor, to allow the top-level policy to observe the second and third bits.

Figure 4.8 shows that the agent with OOIs (14) consistently learns the optimal policy for this task. When the number of options is reduced, the quality of the resulting policies decreases, while still remaining above the agent without OOIs. Even the agent with 4 options, that cannot reach half the goals, performs better than the agent without OOIs but 14 options. This experiment demonstrates that OOIs provide measurable benefits over standard initiation sets, even if the option set is largely reduced.

Combined, our three experiments demonstrate that OOIs lead to optimal policies in challenging POMDPs, consistently outperform LSTM over options, allow

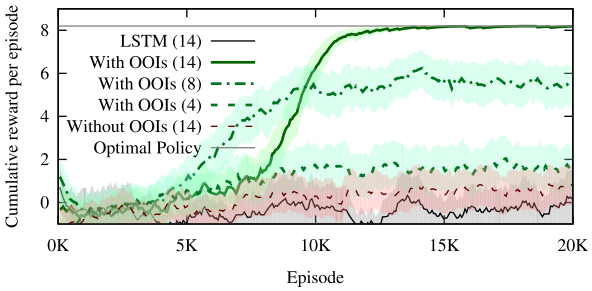

Figure 4.8: Cumulative reward per episode obtained on TreeMaze, using 14, 8 or 4 options. Even with an insufficient amount of options (8 or 4), OOIs lead to better performance than no OOIs but 14 options. LSTM over options learns the task after more than 100K episodes (not shown on the figure).

the option policies to be learned, and can still be used when reduced or no domain knowledge is available.

4.5.5 BDPI with OOIs

Finally, we evaluate BDPI with OOIs on our object gathering task. The BDPI agent uses the same options and actions as the Policy Gradient with OOIs agent (PG + OOIs in Figure 4.9). BDPI is configured as follows: neural networks with a single hidden layer of 64 neurons, trained with the Adam optimizer for 50 gradient steps per 256-experiences batch, with a learning rate of 0.001. There are 32 critics, each trained for 4 training iterations every time-step. The actor learning rate λ is 0.01, and the critic learning rate α is 0.005. Such a small critic learning rate was the only significant change to the default parameters of BDPI we had to apply, as the action-dependent value-function discussed in Section 4.4.2 makes the target Q-Values extremely noisy.

4.6 Conclusion and Future Work

This chapter proposes OOIs, an extension of the initiation sets of options so that they restrict which options are allowed to be executed after one terminates. This makes options as expressive as Finite State Controllers. Experimental results con-

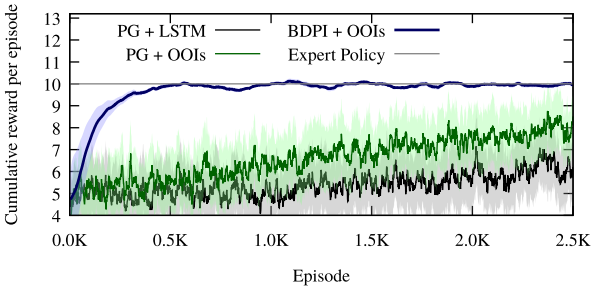

Figure 4.9: Cumulative reward per episode obtained on our object gathering task, with BDPI + OOIs. BDPI allows the agent to learn orders of magnitude faster than Policy Gradient. The horizontal scale of this figure is different from the one of Figure 4.5.

firm that challenging partially observable tasks, simulated or on physical robots, one of them requiring exact information storage for hundreds of time-steps, can now be solved using options. Our experiments also illustrate how OOIs lead to reasonably good policies when the option set is improperly defined. Furthermore, we show that learning the top-level and option policies in parallel allow random OOIs to be used, thereby providing a solution to partial observable problems without requiring engineering work.

Options with OOIs also perform surprisingly well compared to an LSTM network over options. While LSTM over options does not require the design of OOIs, their ability to learn without any a-priori knowledge comes at the cost of sample efficiency and explainability. Furthermore, random OOIs are as easy to use as an LSTM and lead to superior results (see Section 4.5.3). OOIs therefore provide a compelling alternative to recurrent neural networks over options, applicable to a wide range of problems, able to use domain knowledge when available, but not requiring it.

Finally, the compatibility between OOIs and a large variety of reinforcement learning algorithms leads to many future research opportunities. Our implementations of OOIs for Policy Gradient and BDPI are only a few possible implementations. Continuous-action algorithms are also amenable to OOIs, be them on-policy (as in Section 4.4.1) or off-policy (as in Sections 4.3.6 and 4.4.2, where we introduce partial off-policyness).

5 | Reinforcement Learning on a Wheelchair

This chapter has been submitted to the Machine Learning journal special issue on Reinforcement Learning in the Real World. Some sections present (as background information) work mainly carried out by Hélène Plisnier. These sections are explicitly identified

European Commission president Mrs. Ursula von der Leyen being shown the wheelchair: https://www.youtube.com/watch?v=wWpyCYai9O0 (at 4:37)

At some point, agents have to leave the comfort of the lab, and be exposed to the harsh realities of the real world. In this chapter, we discuss the long journey between a Reinforcement Learning algorithm applied to simulated tasks on a computer (as in Chapter 3), and the same algorithm being deployed on a real-world motorized wheelchair that navigates autonomously in an office designed by people.

This chapter considers Bootstrapped Dual Policy Iteration, presented in Chapter 3, and builds a complete Reinforcement Learning system on it. Our system consists of extensions of BDPI to make it even more sample-efficient, a method for ensuring the safety of an embodied agent, and a complete description of how we designed hardware and software components to allow a wheelchair to learn to

navigate in a room. The sample-efficiency and model-free nature of BDPI allows our system to be taught in the field, without any sort of task-specific design or preparation. We believe that, in the real world, it is of utmost importance to be able to bring a self-contained agent on the field, and train it in a short amount of time, which we propose in this chapter. Our system does not need an Internet connection to the Cloud, can be taught a wide variety of tasks in many human-centric environments, does not require maps or models of the environment to be built, and does not require environment or task-specific code to be written, or data to be acquired. We stress the generality of our approach, that requires no task-specific feature engineering, robot-specific adaptations to the environment, expensive sensors, or high-quality precisely-controlled actuators. We believe that our approach of *cheap robots, smart algorithms* will greatly facilitate the deployment of robots in homes and offices, where cost and maintenance need to be minimized. Moreover, our algorithm leads to stable agents, that learn high-quality policies for every random seed and most configurations of hyper-parameters. This is crucial in the real-world, where it is impossible to spend robot time tuning hyper-parameters or restarting *unlucky runs*.

While we deploy our system on a motorized wheelchair, most of our contributions in this chapter also apply to non-robotic systems, such as Reinforcement Learning-based recommender and decision systems, industrial settings [Lazic et al., 2018], or renewable energy infrastructure [Verstraeten et al., 2019], in which sample-efficiency is paramount.

5.1 Learning to Navigate in an Office

The running example of this chapter is a commercially-available motorized wheelchair that learns to navigate in an office space, as shown in Figure 5.1. We believe that this both raises and showcases the applicability of model-free Reinforcement Learning algorithms to real-world settings. Our wheelchair setting is both challenging from a practical point of view (see Sections 5.1.1, 5.4.2, 5.4.4, 5.4.5), and an application of Reinforcement Learning with high social impact, as our method has the potential to largely increase the quality of life of people with reduced mobility and hand control skills [Montemerlo et al., 2002; Atrash and Pineau, 2009; Feng et al., 2018]. In this thesis, the wheelchair performs its navigation task in a stationary office. Furniture does not move during training, and people remain seated. If someone moves in front of the wheelchair, this introduces noise, but this does not appear to perturb learning. However, we acknowledge that Reinforce-

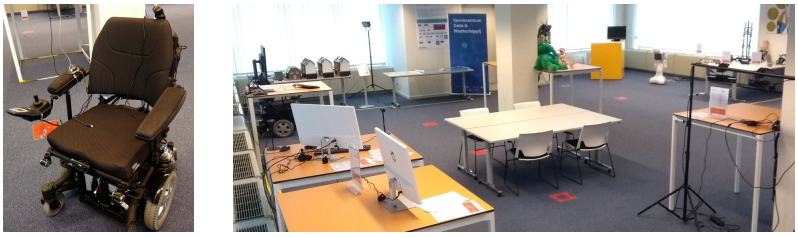

Figure 5.1: The motorized wheelchair (left) learns with model-free Reinforcement Learning, in one hour, to navigate in a large cluttered office environment (right). Regular changes in the configuration of the office, and the impossibility to map it, make it a good motivating example for model-free Reinforcement Learning.

ment Learning, especially our algorithms, consider *stationary environments* and do not actively reason about moving objects. We believe that stationary environments are compatible with many home robots, especially wheelchairs on private property, with the only moving organism being seated in the wheelchair while in use.

We now describe the wheelchair, and the challenges that arise with deploying Reinforcement Learning in the real world. Then, in Sections 5.2 to 5.6, we detail our complete Reinforcement Learning system that allows the wheelchair to learn navigation tasks. Finally, in Section 5.7, we empirically demonstrate the applicability of our system, by teaching the wheelchair a complicated navigation task in only one hour (without pre-training, mapping, or annotating the environment).

5.1.1 The Motorized Wheelchair

We deploy our Reinforcement Learning agent on the Invacare TDX SP2 Ultra Low Maxx[1] motorized wheelchair, depicted in Figure 5.2. The wheelchair is commercially available, focuses on being used by a person, and has not been designed to be controlled by a computer. It has no sensors, and can only be controlled with a physical joystick. Moreover, how the wheelchair physically reacts to movements of the joystick has not been optimized for automatic control, and is highly non-linear and difficult to predict. Any action on the joystick requires a few seconds to take full effect. Rapid conflicting actions, such as going left then right, cancel out

[1] http://www.invacare.eu.com/tdx-sp2-ultra-low-maxx-ma-40tdxsp2ulmen

Figure 5.2: *Left:* Our motorized wheelchair. *Right:* In this wheel configuration, the chair will momentarily go right even if the agent tries to go forwards (by applying the same torque to both large wheels).

without the chair moving. When the joystick is put back in its neutral position, the wheelchair progressively decelerates in a non-linear way.

The motors are connected to two big wheels, whose differential speed allows to steer. Around those wheels are four smaller articulated ones, for balance. Because the wheelchair has no encoder on the wheels, and the small wheels influence steering (see Figure 5.2), an agent is unable to predict how a given joystick position will make the wheelchair move. Observing the state of the small wheels would require additional sensors to be added on the chair, which would increase its cost. Moreover, the speed at which the chair moves, and the aggressiveness of its turning, depends on the battery charge, a quantity that is also unobservable on the wheelchair as available in the market. The combination of the delayed joystick and partially-observable steering makes this wheelchair a highly challenging setting for a Reinforcement Learning agent, or any other control approach.

5.1.2 Related Work on Robot Control

Most current methods for controlling robots, such as the ones reviewed in the excellent book by de Wit et al. [2012], rely on the existence of a model of the robot and its environment. Models are easy to identify for heavy and precise robots, that do not wobble and have accurate ways of measuring their current pose and position. Building these models can be quite labor intensive, but usually pays off in repetitive industrial settings, and are often directly provided by the manufacturer [Rethink Robotics, 2016, for the Baxter robot]. However, in consumer settings (e.g. homes and offices), where robots have the potential to allow physically impaired users to move better or accomplish more everyday tasks, models are much harder

to design. Most consumer robots are provided as is, without a description of their reactions to commands, such as acceleration curves. One example is the motorized wheelchair that we consider in this chapter, that is controlled through a joystick with user-tunable, highly non-linear and time-dependent dynamics.[2] Rapid movements of the joystick are ignored by the wheelchair, and the user can freely configure how strongly the wheelchair accelerates when asked to turn or go forward. Settings in which modeling the robot is highly challenging also arise in the industry. For instance, a new and promising research direction considers *soft robots*, that are soft, malleable, compliant and self-healing [Terryn et al., 2017]. These robots have the potential to solve new tasks, that require minutia and a delicate touch. Their self-healing properties (the rubber material they are made of fuses itself alongside cuts when the cut is maintained in a closed position for a few seconds) will also reduce the need for periodic maintenance. However, a challenge of these soft robots is that they are a poor fit for classical rigid-body models, and modeling almost every molecule of their soft rubbers is impossible. We envision great opportunities for model-free approaches on these robots.

In addition to the robot, the environment itself is also challenging to model. Contrary to industrial robots that today still mainly act in fixed security cages, consumer robots act in highly-complex environments, that might change often, and that may involve people and pets for which we have no model. Planning in these environments requires highly creative approaches, as there are challenges in both locating the agent (by observing walls and objects that may have moved overnight) and measuring the effect of actions (people move independently of the robot). The CoBot mobile service robots [Veloso, 2018] plan for navigation tasks in a research lab, using odometry, image and depth sensors to build, maintain and locate themselves in maps of the lab. They use advanced algorithms to detect and act around people [Choi et al., 2013; Gui et al., 2018], or infer their location in the building from sensor readings [Wang et al., 2014].

In this thesis, we consider another path towards successful navigation in human-centric spaces, and rely on the ability of our agents to *learn* creative solutions to the challenges of human spaces. More specifically, we propose to use *model-free* Reinforcement Learning to learn policies that allow a robot to navigate a complex environment, from first-person sensors. We believe that model-based Reinforcement Learning, that requires the agent to learn a model of the environment, may not be suited to this kind of problem. The work on CoBots discussed above demonstrates that models of human spaces are highly difficult to build, and hence to

[2] In this thesis, we refer as *dynamics* how an environment reacts to actions executed in it.

learn, so we believe that mixing Reinforcement Learning with (learned) models of human spaces would introduce the complexities of both approaches, possibly canceling the benefits of both model-based planning (the CoBots approach describe above) and model-free learning. We therefore focus on Reinforcement Learning approaches that learn by directly interacting with the real environment.

On physical robots, fully model-free Reinforcement Learning approaches are rare: current Reinforcement Learning approaches either require a model to be learned [Hester, 2013], a large amount of robots [Gu et al., 2017a], human-provided demonstrations [Hester et al., 2017; Balakuntala et al., 2019], or transfer between a simulated environment and the real world [Bousmalis et al., 2018; Karttunen et al., 2019; Xie et al., 2019]. This last approach has led to impressive results, but requires solutions to the simulation-to-real-world gap [Haarnoja et al., 2017; Tobin et al., 2017], as accurately simulating processes such as the friction of a wheel on the particular carpet in that particular house is impossible. Focusing on agents that learn directly on a real robot, Levine et al. [2016] propose to mix pre-training of convolutional networks, local policies and Policy Gradient, but their approach still requires significant data collection and domain knowledge. Tedrake et al. [2005] propose a Policy Gradient method that learns to walk in 20 minutes, but requires task-specific feature engineering and very low-dimensional observations. Bayesian approaches, such as PILCO [Deisenroth and Rasmussen, 2011], are highly sample-efficient but also limited to low-dimensional state-spaces. Finally, *off-policy batch approaches* execute a good-enough policy in the environment, such as a control program designed by engineers, or actions directly carried out by actual people, and store the resulting experiences in a large buffer. Then, an off-policy Reinforcement Learning algorithm learns a (hopefully) better policy by only observing the collected samples, without interacting with the real environment [Lazic et al., 2018, for a successful application]. On our motorized wheelchair, this would require a person to repeatedly guide the chair to a goal position for some time. This is possible, but we leave the evaluation of that approach to future work. We note, though, that BDPI can trivially be used as a batch Reinforcement Learning algorithm, as it is off-policy and able to cope with large experience buffers.

In the next sections, we present the hardware and software architecture that allows a Reinforcement Learning agent to interact with a commercially-available motorized wheelchair. This work, of making a robot or computer system compatible with Reinforcement Learning, is the primary source of complexity in real-world Reinforcement Learning settings. Fortunately, because our Reinforcement Learning system is model-free and requires no environment-specific design, preparing

Figure 5.3: Closeup of the embedded computer of the wheelchair, a few control buttons, and the joystick.

a robot for Reinforcement Learning has to be done only once per robot type (a wheelchair for instance), not for every environment the robot will be deployed.

5.2 Executing Actions on the Wheelchair

The motorized wheelchair we use in this chapter has not been designed to be controlled by software. It does not expose any standard interface to which a computer can connect to, such as an USB or Ethernet port. Various components of the wheelchair, such as the motors and the joystick, communicate on a proprietary bus inspired from the CAN bus. Unfortunately, while the CAN bus defines the electrical properties of the bus (that are not fully followed by the wheelchair), the contents of the messages exchanged on the bus of the wheelchair is not specified, and can therefore not be replicated by a computer connected on the bus.

As such, in this section, we present how we make the embedded computer on the wheelchair believe that the joystick has moved, by exploiting the electrical properties of the physical joystick. We describe our system starting from our understanding of the (reverse-engineered) physical joystick, then explain how an Arduino connected to a Raspberry Pi controls it, and finally how a Reinforcement Learning agent can send actions to the Raspberry Pi over Wifi.

5.2.1 Making the Chair Believe its Joystick Moved

The wheelchair, as available in the market, is controlled using a joystick shown in Figure 5.3. The exact nature of the joystick is not disclosed by the manufacturer of the wheelchair, but disassembling the embedded computer to reveal the underside of the joystick hints at an inductive joystick [Baker et al., 1997]. An inductive joystick uses a coil on the moving part of the joystick, and (in this case) four stationary coils at the base of the joystick. When the joystick is moved, the distance between the main coil and the four stationary coils varies, and can be used to determine the exact position of the joystick. Baker et al. [1997] present an inductive joystick that relies on alternating current in the coils, and a rectifier and averaging circuit between the coils and the embedded computer, to measure the position of the joystick. While we were able to confirm the presence of the coils in the wheelchair we use, we could only observe that it was connected to the embedded computer using four wires on which we measured a continuous (not alternating) voltage. We therefore suppose that the base of the joystick contains the rectifier and averaging circuit described by Baker et al. [1997], or something equivalent.

The joystick is connected to the embedded computer with 6 wires: ground, +5V, +X, -X, +Y, -Y. The X and Y lines encode the 2D position of the joystick, with (0, 0) the center and (1, 1) right forward. We observed that the X and Y lines each consist of a pair of wires between the joystick and embedded computer, used together in a differential way. If we focus on X (the same applies to Y), various positions of the joystick lead to different voltages being present on the +X and -X wires. When the joystick is fully on the left ($X = -1$), the +X and -X wires are set to 1.25 volts and 3.7 volts respectively. When $X = 1$, the pair is set to 3.7 volts and 1.25 volts. When the joystick is in its center position ($X = 0$), the two wires are set to the same voltage, 2.45 volts.

We observed that appropriately setting the voltages on the four X and Y wires allows, without having to physically move the joystick, to make the embedded computer of the chair believe that the joystick is in a particular position. We therefore soldered four wires on the internal part of the joystick, brought them out of the armrest of the chair through a small hole, and connected them to a digital-to-analog converter (DAC) controlled by an Arduino. We now describe the bus used between the Arduino and the DAC, take the opportunity to also describe the UART and CAN buses used elsewhere on the system, then present the Arduino

platform and how we use it to control the wheelchair, through the DAC, in Section 5.2.4.

5.2.2 Synchronous and Asynchronous Busses

The main purpose of a bus is to allow two electronic devices A and B to exchange *data*, usually sequences of 8-bit bytes, in a standardized way that allows devices compatible with the same bus to be mixed and matched on the same system. Many families of buses exist, and have slightly different properties that are suited to different communication needs. In the interest of brevity, we will only review three buses, used on the wheelchair or as part of our Arduino-based system: the Universal Asynchronous Receiver-Transmitter [Osborne, 1982, UART], the Controller Area Network [De Andrade et al., 2018, CAN], and the Serial Peripheral Interface (SPI), an industry standard for which there is no introductory book, but a large amount of hardware patents, such as Miesterfeld et al. [1988].

UART

The UART bus requires two pins on the two devices to be connected: *TXD* and *RXD*, for *transmit* and *receive*. The TXD pin of A is connected to the RXD pin of B, and vice versa. This allows both devices to send and receive data from each other. We now focus on the connection between the TXD pin on A, and the RXD pin on B, as the B-to-A connection works in exactly the same way.

When no communication needs to happen, the device sets its TXD pin to 1 (that is, its supply voltage). When the device wants to transmit a byte (8 bits), it sets the TXD pin to 0, then sends the 8 bits of the byte sequentially, starting from bit 0. After the byte is transmitted, the TXD pin is set back to 1. The UART bus can also be configured for an extra parity bit to be sent after the 8th bit.

Because the bits are transmitted sequentially, we must define at which frequency they are sent. Both the transmitter device and the receiving one must agree on that frequency, otherwise the receiving device will observe changes of values on its RXD pin without knowing when each individual bit starts. Usually, one of the device is a simple sensor, and the frequency it requires is listed in its documentation. The other device is a computer or an Arduino, something that can be programmed, and its UART end can be configured to use a speed that matches the device connected to it.

SPI

The SPI bus, that we use in Section 5.2.4 to send commands to a digital-to-analog converter to control the joystick of our wheelchair, shares many properties with the UART bus. There are also two wires, that allow two devices to send data to each other. The two devices have pins that are named slightly differently from their UART version: the MISO pins of the two devices (master in, slave out) must be connected together, as do the MOSI pins (master out, slave in). Compared to the UART bus, the SPI bus presents two key differences:

1. While UART treats the two devices equally, SPI considers one of the device to be the master, the other one the slave. The master always initiates a communication, that consists of the master sending a byte to the slave while the slave sends a byte to the master. The slave has no way of telling the master it wants to talk. The SPI bus cannot therefore be used in settings where the slave would look out for specific events, and tell the master when the event occurs.

2. A third wire, named *CLK* for *clock*, allows the master to tell the slave exactly when it can read a bit from its receive pin, and when it can set its output pin to a bit it wants to send.

The presence of a clock, a wire that alternates between 0 and 1 repeatedly, at a fixed frequency chosen by the master, allows the slave to automatically adjust to the transmission speed of the master. This addresses an issue of the UART bus, that requires the two devices to implicitly agree on a transmission speed, with device-specific mechanisms.

CAN

The CAN bus has been designed to connect many devices on a single bus. It is now commonly used in cars, where the windows, ignition, fuel gauge, engine controller, heating system, and so on, need to communication with each other and the dashboard. The need to connect many devices on a single bus, and its use in mission-critical systems, motivate the electrical aspect of the bus.

The CAN bus relies on two wires, called CANH and CANL. Every device on the bus has two pins, CANH and CANL, connected on the corresponding wires. All the CANH/CANL pins of all the devices are therefore connected together. The two lines are constantly monitored by every device for messages, and are also

used to inject a message on the bus. Both lines are also connected to both 0V and the bus' supply voltage, 24V on our motorized wheelchair, through resistors. This makes a voltage divider circuit, such that both CANH and CANL have a voltage of about 12V (half of 24) when resting. When a device wants to send a message on the bus, it first waits for CANH and CANL to be back to their resting state (which indicates that no other device is talking). Then, the message is sent by expressing a 0 with CANH > CANL (usually, CANH is driven to 24V and CANL to 0V), and a 1 with CANH $\leq$ CANL (usually, the bus is left in its resting state while transmitting a 1). Bits of a message are sent sequentially. How many bits are in each message, and what they mean, is device-specific and not specified by the bus. This is the primary reason why we could not interface with the CAN bus on the motorized wheelchair: we could observe bits on it, and group them in bytes, but the meaning of these bytes was unknown.

5.2.3 The Arduino Platform

This section presents the Arduino platform,[3] a family of easily-programmable microcontrollers that we use on our wheelchair to connect a standard computer, running Linux, to a digital-to-analog converter used to programatically move the joystick on the wheelchair.

The Arduino family contains many products, that all share a few common traits: they are built around various kinds of low-power microcontrollers, and can be connected to many electronic devices over several busses, in an easy and experimenter-friendly way. Microcontrollers are tightly-integrated packages that contain a simple low-power microprocessor, some dynamic memory (RAM), a small amount of static memory (ROM, the equivalent of a hard drive on a standard computer), and components that allow the microprocessor to communicate with devices connected to the Arduino over a variety of buses. The Arduino platform is easy to develop for, as a user-friendly development environment, with high-level C++ functions and libraries, is available for free.

The Arduino Nano that we use in Section 5.2.4 contains an 8-bit microprocessor running at 16 Mhz, 32 kilobytes of ROM for storing compiled programs, 2 kilobytes of RAM, and connectivity over a few busses, SPI included.[4] These low specifications, orders of magnitude lower than even the cheapest smartphones,

[3] https://www.arduino.cc/
[4] https://store.arduino.cc/arduino-nano

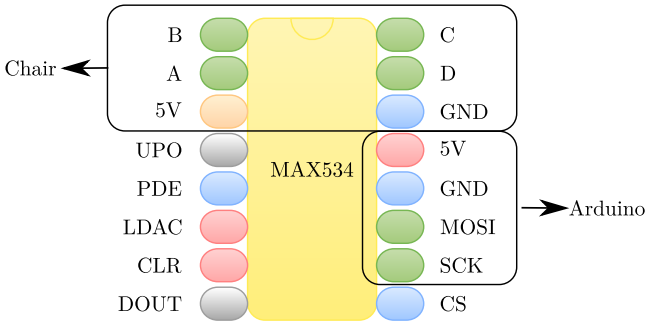

Figure 5.4: The MAX534 digital-to-analog converter. The four outputs (A, B, C, D) are connected to the +X, -X, +Y, -Y wires of the joystick. A 5 volt reference and ground are also connected to the joystick. 4 other pins are connected to an Arduino Nano, and are used to power the MAX534 and communicate with it over the SPI bus. In this figure, the red and blue pins denote two groups of pins that we connect together. The gray pins are unused outputs and are left unconnected.

illustrates the primary use of microcontrollers: extremely low-power operation for simple communication tasks. The Arduino Nano, for instance, consumes 95 mW when in full operation (yes, 0.095 watts, compare to any light bulb that consumes at least 3 watts for the lowest-power most-efficient ones). Arduino boards should not be used for any kind of intensive processing, but excel at acquiring data from a range of electronic devices, and relaying that data to some other device or a computer.

As we will discuss in the next section, the Arduino Nano has a USB port, that allows it to connect to a computer. The USB connection powers the Arduino, and allows the computer to exchange data with the Arduino. Special hardware on the Arduino Nano shows the computer to the program running on the Arduino as an UART device, and presents the Arduino to the computer as an USB-Serial device, easy to use from high-level programming languages such as Python.

5.2.4 Interfacing with the Joystick on the Wheelchair

In this section, we combine the components introduced in the previous sections in a complete system. We start from the physical joystick, controlled by the DAC, controlled by the Arduino, that receives commands from a Raspberry Pi, that

connects to the Reinforcement Learning agent running on a desktop computer over Wifi.

The four wires of the joystick, +X, -X, +Y and -Y, described in Section 5.2.1, are connected to the 4 outputs of a MAX534 digital-to-analog converter (see Figure 5.4). The DAC is able to produce 4 high-precision voltages at the same time, on 4 output pins, which fits our need of precisely controlling four wires at once. The actual joystick is still connected to the embedded computer of the chair, in parallel with the DAC, but we added little switches on the wires to the DAC. When the switches are open (disconnected), the joystick on the wheelchair works as usual. When the switches are closed (which connects the DAC to the control panel), the joystick stops functioning, moving it does not lead to any response from the wheelchair, and the DAC gets full control of the wheelchair. The reason why the DAC takes precedence on the joystick comes from its ability to strongly force a desired voltage on the wires (the MAX534 documentation mentions that the DAC is able to output 34 mA on its 4 outputs, much more than what the actual joystick can produce).

The DAC produces voltages on its four outputs as instructed by an Arduino connected to it with an SPI bus (see Figure 5.4). Because the DAC accepts commands, that it translates to voltages on its four outputs, but does not produce data per se, it has a MOSI pin (master out, slave in) but no MISO pin (master in, slave out). This is not a problem, and we simply connect the MISO pin of the Arduino to the ground, meaning that the Arduino will believe that the DAC always writes 0 on the SPI bus.

Simple C++ software on the Arduino receives commands from the USB port of the Arduino, that simply consist of X,Y coordinates to which the joystick has to be moved. These commands are sent in simple textual form by the Reinforcement Learning agent running on the computer, on the other end of the USB cable. An example of a command is “X0.0 Y0.8", for moving the joystick forwards 80% of the way. When the Arduino receives a command, it converts the X-Y coordinates to voltages to instruct the DAC to inject on +X, -X, +Y, -Y, then sends a sequence of instructions to the DAC on the SPI bus. These instructions tell the DAC to slowly move its output towards the target voltages, over about half a second. We discovered that rapidly changing the voltages makes the embedded computer of the wheelchair believe that the joystick has been hit by an object, leading to it stopping the wheelchair and entering a fail-safe state.

The C++ code that runs on the Arduino is available as supplementary material. Because it contains a few work-arounds for the sensitivity of the wheelchair to abnormal joystick events, it is too complicated and wheelchair-specific to be worth

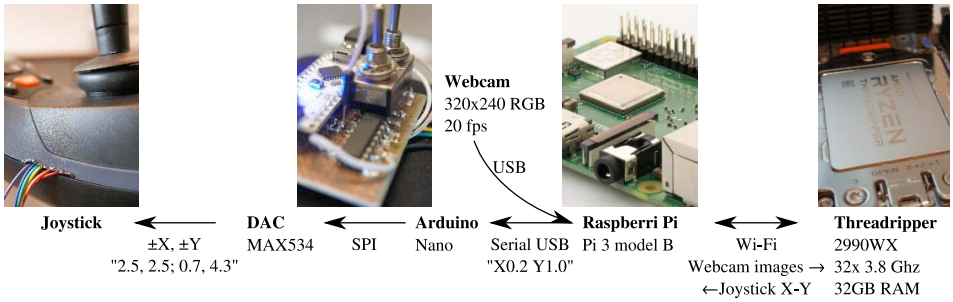

Figure 5.5: Hardware architecture that allows a Reinforcement Learning agent (running on the Threadripper machine) to control a motorized wheelchair through its joystick.

listing in the main text. We now move on the other side of the Arduino, the one connected to a computer over USB. Because almost every kind of computer has USB ports, it is possible to control the wheelchair by putting a laptop on it, and connecting the Arduino to one of its USB ports. Because the wheelchair must be wireless (not tied to a network or power cable), battery life is important. Most laptops in the market have battery lives in the range of a few hours, that dramatically decrease when the laptop is performing computations. Because this was not practical in a real-world settings (even in a lab) to recharge an experiment-critical laptop every few hours, we instead connect the Arduino to a Raspberry Pi, powered by a 40Wh USB power bank, providing about one day of autonomy and easily swappable between experiments.

Because a Raspberry Pi (3, model B, in our case) has limited computing capabilities, we decided to off-load any form of computing to a desktop computer connected to the Raspberry Pi over Wifi. More precisely, the Raspberry Pi is responsible for obtaining images from a webcam placed at the front of the wheelchair (see Section 5.3), making them available on the Wifi network, and accepts X-Y joystick commands from the network and forwards them to the Arduino over USB. The Raspberry Pi hence merely acts as a *Wifi-to-Arduino* bridge. As such, it only consumes a small amount of power (4 watts), and allows continuous use of the wheelchair for up to 10 hours before needing its battery pack to be replaced.

The Raspberry Pi automatically connects to a pre-defined Wifi network at boot time, and exposes itself to the network as a simple server. The Python code that implements that server (that relays webcam images and X-Y coordinates to/from

the network) is available as supplementary material. On the same Wifi network is a powerful desktop computer, built around an AMD Threadripper 2990WX processor (32 cores at 3.6 Ghz). This many-cores machine, far less expensive than actual server hardware but highly powerful, is fully leveraged by the Asynchronous Parallel BDPI algorithm we present in Section 5.6. On the Threadripper machine, a simple Reinforcement Learning environment, implemented in Python and following the OpenAI Gym API [Brockman et al., 2016], connects to the Raspberry Pi, queries it for images (used to produce observations), and sends it X-Y coordinates to allow the agent to execute actions.

The complete hardware architecture of our system is summarized in Figure 5.5. Despite the complexity of this architecture, we point out that it is very easy to use, and is the structure (among many tried over the years) that we prefer. Starting a Reinforcement Learning experiment on the wheelchair simply consists of starting a Python program on the Threadripper machine, and the wheelchair automatically starts to move. We also discovered that, because the Raspberry Pi server allows several clients (but only one at a time can control the joystick), we can connect many laptops to the Wifi network, and open a first-person view of the webcam on all of them. This allows us to perform large-scale demonstrations of the wheelchair, in a big room, in front of up to 50 people, with laptops everywhere that allow everyone to see what the wheelchair sees.

The description of the acting side of our Reinforcement Learning agent is now finished. In the next section, we detail the Computer Vision algorithms that we use to produce *task-agnostic* informative observations from webcam images.

5.3 Computer Vision on the Wheelchair

Reinforcement Learning agents deployed in real-world systems need two interfaces with the world: an *acting* side, that allows the agent to execute actions that have real effects, and a *sensing* side, that collects information from the real world, to be used by the agent.

While Section 5.2 presented the acting side of the agent we deployed on a motorized wheelchair, this section focuses on the sensing side. We detail how we use a single webcam, attached at the front of the wheelchair, to produce task-agnostic observations for a Reinforcement Learning agent. Our observations are higher-level than simple pixels, allowing efficient learning without needing a supercomputer to train deep convolutional neural networks. They are, however, low-level enough to encode nothing about the task to be learned. This allows our

sensing algorithms to be used as-is in many different rooms or offices, to learn many different tasks.

5.3.1 Acquiring Images with OpenCV

As is done in Section 5.2 with electronics, we begin this section with a brief introduction to the fundamentals of Computer Vision, and more precisely the use of the well-known OpenCV library of computer vision functions.[5]

In most computer systems, and specifically in OpenCV, an image is a three-dimensional array of numbers. The first two dimensions represent (respectively) the height and the width of the image in pixels. The third dimension allows to represent color. Each pixel in the image is represented by three numbers, usually for the red, green and blue components of the pixel. This decomposition, often referred to as RGB, closely matches how the human eye and camera sensors see light. In the human eye, three different kinds of *cones* have peak sensitivity at three different locations of the light spectrum: red, green, and blue-violet [Wald, 1964]. By combining the signals obtained from these three kinds of cones, the brain is able to perceive all the colors we can see. Photo cameras mimic the human eye, and also have red, green and blue photo-receptors for every (or almost every) pixel. They simply store the RGB reading of each pixel in the image files they produce.

With OpenCV used in Python, images are therefore three-dimensional Numpy arrays of dimension $(height, width, 3)$. An introduction to Numpy is provided in Section 2.3.1. Acquiring an image consists of producing a Numpy array from some image source: an image file, a webcam, or a video file, most of the time.

Image Files

The simplest method to obtain an image with OpenCV is to open an image file, or a photo, with the imread function. OpenCV is able to read many image formats, such as the well-known JPEG, PNG or TIFF formats, or the rarer but often produced by embedded computers PPM family of formats [Murray and VanRyper, 1996].

```
import cv2

image = cv2.imread("path/to/image.jpg")
```

[5] https://opencv.org/

```
image.shape      # A 3-uple (height, width, 3)
image[100, 100]  # A Numpy array of 3 elements for the pixel at (100, 100)
```

By convention, the X coordinate of a pixel in an image represents its horizontal position, and goes from 0 on the left to the width of the image (excluded) on the right. The Y coordinate of a pixel represents its vertical position, and goes from 0 on the top to the height of the image on the bottom. The (0, 0) pixel is therefore located at the top-left of the image, with the Y axis going downwards. This is opposite to the standard direction in mathematics, where the Y axis goes upwards, and therefore sometimes requires attention when mathematical equations are applied to pixel locations (such as computing angles and slopes).

Webcam and Video Files

OpenCV is a high-level library, that allows many complicated tasks to be performed with one function call. Extracting images from video streams, regardless of the source of format of the stream, is an example of such tasks. Video streams, either produced in real time by a webcam or read from a video file, are read by OpenCV using the VideoCapture class. A VideoCapture instance is created by passing a file-name to its constructor. The file name either points to a movie file, in conventional formats such as MP4 or webm, or a Video4Linux webcam file. When it is certain that the computer on which the code runs only has one webcam, an additional VideoCapture constructor allows to open the first webcam of the system, and is portable across computers and operating systems.

```
movie = cv2.VideoCapture("exciting_movie.mp4")  # A video file
webcam = cv2.VideoCapture("/dev/v4l/by-id/usb-XXX", cv2.CAP_V4L2)
webcam0 = cv2.VideoCapture(0)                   # The first webcam
```

The use of Video4Linux paths, on Linux, allows to unambiguously identify webcams connected to the system, and is therefore particularly relevant to robotic systems that use several webcams, such as one on the left and one on the right or a robot. Using the short and os-agnostic constructor may randomly permute the two webcams, depending on in which order each webcam registered itself on the USB bus.

Once a VideoCapture object is created, images are obtained by repeatedly calling its read method. This method returns the next image in the stream, as fast as it can be decoded for movies, and as fast as they are produced for a webcam (typical webcams produce 20 to 30 images per second at most):

```
ret, image = webcam.read()
```

ret allows to identify errors. Usually, checking whether image is *None* is enough. For movies, a null image is returned at the end of the video file. For webcams, a null image is called if read is called too often. It is therefore recommended to give the webcam a little bit of time to produce a new image, and try again:

```
import time

ret, image = webcam.read()

while image is None:
    time.sleep(0.01)
    ret, image = webcam.read()
```

Saving Images

In the next section, we present how OpenCV can be used to modify images. Modified images can be saved back to a file using the imwrite function. Two other interesting functions are imencode and imdecode. They perform the same operations as imwrite and imread, but consider byte buffers instead of files. For instance, this allows to JPEG-compress an image, obtain the resulting byte sequence, send it over the network, and reload it in a three-dimensional Numpy array on another computer.

```
cv2.imwrite("modified_photo.jpg", image)  # A file
data = cv2.imencode(".jpg", image)        # JPEG bytes
image2 = cv2.imdecode(data)               # Re-load the JPEG-bytes
```

This concludes our introduction of image acquisition with OpenCV. On the motorized wheelchair, we use the webcam method of acquiring images. We then save the acquired images to JPEG data blobs, that we send over the network to the Threadripper machine. The Threadripper machine loads that data and forms a new image from it, on which image processing algorithms are applied. A summary of our processing architecture is given in Figure 5.5 on page 144. The next section presents several common image processing algorithms available in OpenCV.

5.3.2 Image Processing with OpenCV

OpenCV is a vast library, and introducing all its features is outside the scope of this thesis. We refer the interested readers to an excellent book on OpenCV with Python by Howse [2013]. In this section, we briefly present the subset of OpenCV features that we use in Section 5.3.3 to detect obstacles in webcam images.

Blur and Edge Detection

Most image processing functions that we present in this thesis take an image as input, and produce a new image as output. The simplest example of this is the blur function, that returns a blurred version of an image, with the amount of blur specified by a *kernel size*. The kernel size, expressed in pixels, roughly represents how many pixels are averaged together to produce one output pixel. The larger the value is, the more blurry the return image is.

Detecting edges is an important step in most computer vision systems. The Sobel filter detects edges by computing how different the color of a pixel is compared to its horizontal or vertical neighbors [Sobel and Feldman, 1968]. In OpenCV, the Sobel function applies the Sobel filter to an image, and is, like blur, parameterized with a kernel size, that defines how many pixels at once the algorithm considers when computing edges. Because the Sobel filter detects either vertical or horizontal edges, applying it two times on the same image, and combining the result, allows to detect edges in every direction.

```
image = image.astype(np.float32)  # Converts to float

h = cv2.Sobel(image, ddepth=3, dx=1, dy=0, ksize=5)
v = cv2.Sobel(image, ddepth=3, dx=0, dy=1, ksize=5)
edges = h + v
```

In the code above, edges is a Numpy array of the same dimension as image, usually (height, width, 3). Edges have been detected separately for the three color channels of the image. Interestingly, edges contains signed numbers, as the Sobel algorithm differentiates between edges from dark to light and edges from light to dark. Usually, an additional line takes the absolute value of edges with np.abs(edges). The presence of negative values, and the large magnitude they can have, explains why the first line of the code above casts the image to an array of floating-point values, while OpenCV functions like imread produce arrays of

8-bit unsigned integers. With OpenCV, it is common practice to load an image, immediately cast it to floating-point values, and apply all the computer vision algorithms on these floating-point values.

Numpy Operations

Images are Numpy arrays, which means that everything Numpy can do on arrays can be done on images. Examples include type conversion (OpenCV loads images as unsigned 8-bit integers, that Numpy can convert to floating-point), arithmetic operations, and logical operations. Logical operations are particularly useful after edges have been detected in an image, to only keep the strongest edges:

```
# Normalize edges to abstract away the effect of various kernel sizes
edges -= np.mean(edges)
edges /= np.std(edges)

# Threshold to only keep stronger-than-average edges
mask = (edges > 0)
mask = mask.any(2)
```

The code above produces mask, a (height, width) array of boolean values. The first lines use Numpy functions to compute the mean and standard deviation of all the elements (combined) in the detected edges map, produced in the example code snippet shown a few paragraphs higher. The last line, with any, implements a logical or operation on mask. The function considers the third dimension of mask, that corresponds to strong edges in the red, green and blue channel, and performs a logical or between these three boolean values for every pixel in the image. The third dimension of mask therefore vanishes, producing a 2-dimensional mask with a single boolean value per pixel: whether it is a strong edge considering any of its red, green or blue components.

A few OpenCV functions work with arrays of boolean values, and can be used to recognize shapes in images, as we now detail.

Connected Components

2-dimensional arrays of boolean values, as produced above in mask, can also be referred as *binary images*. A class of OpenCV functions allow to consider blobs of binary pixels that have the same value. For instance, the connectedComponents function considers that every pixel having a true value is part of an object, and

that every false pixel is part of the background. Then, objects are identified by looking for collections of adjacent pixels that are true. The connectedComponents function returns an array of the same dimensionality as its input, but with 32-bit integer elements. Each element is set to 0 if the corresponding pixel was part of the background, and to a positive number, uniquely identifying the object it belongs to, if the pixel was part of the foreground. connectedComponents is often used, combined with the HSV color space and thresholding operations on the Hue component, to count how many objects of a given color are in an image, and where they are.

A second function, connectedComponentsWithStats, performs the same operation as connectedComponents, but returns extra information in addition to the 32-bit integer map: how many objects have been detected, what their size in square pixels is, and where their center is located in the image. We use connectedComponentsWithStats in the next section, to remove small-area edges in a map of edges detected in an image, as a noise reduction mechanism.

5.3.3 Obstacle Detection from Monocular Images

While the previous sections introduce Computer Vision algorithms available in the OpenCV library, this section details how we use these algorithms to identify obstacles in images captured by a single webcam, installed at the front of our motorized wheelchair. The wheelchair, as acquired from its manufacturer, has no sensors whatsoever. Because we want to deploy the wheelchair in homes and offices in the simplest way possible, we purposefully limit ourselves to cheap and non-invasive sensors, such as a simple webcam. We do not consider sensing techniques that require the environment of the wheelchair to be annotated, mapped or constrained in any way.

A single webcam produces first-person color image observations. From these observations, we propose to detect obstacles, and ultimately to produce an array of distance readings, as if the chair was fitted with a large number of ultrasonic range finders. This approach, as opposed to feeding images directly to the agent, relieves it from the need of deep convolutional neural networks, that require too much training data to be trained in a sample-efficient way. Arrays of distance readings still allows the agent to recognize the shape of objects in front of it, and leads to policies almost at human expert level in our experiments (see Section 5.7.2). Distance readings can be processed with simple fully-connected feed-forward neural

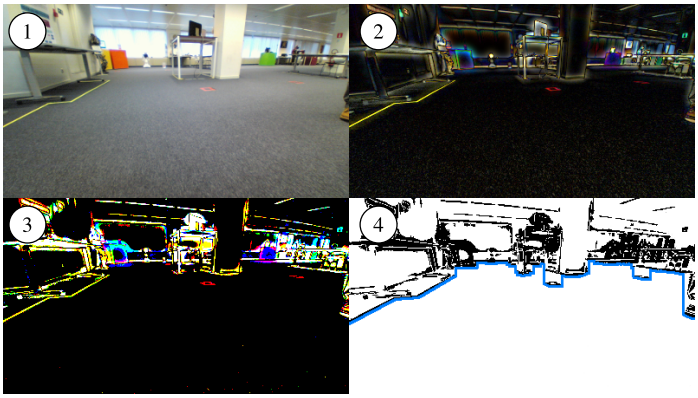

Figure 5.6: Illustration of our obstacle detection algorithm. 1) Original color image. 2) Changes in texture detected by two applications of the difference-of-Gaussians method. 3) Thresholding operation (still in color). 4) Logical or between color fields, removal of small connected components (noise), and final distance readings (blue line).

networks, require less samples before the agent learns a good representation of them, and lead to more general policies.

Detecting obstacles in color images, that is, segmenting *floor* and *non-floor* pixels, is an open problem that is intensively studied. Lenser and Veloso [2003] recognize the floor by its specific color, while Ulrich and Nourbakhsh [2000] propose a simple method that computes the color histogram of parts of the image, and compares it to a moving average color histogram of the floor. Wedel et al. [2006] focus on far obstacles on a road, and detect anomalies in how pixels move between image frames to detect obstacles. Yamaguchi et al. [2006], and others, propose a feature-based approach to movement detection in videos, and consider as obstacles anything that moves at a different speed than the camera. Finally, Lee et al. [2016] use odometry sensors on the robot to measure its physical movements, use that to predict where each floor pixel would be in the next video frame, and any pixel not matching that movement is segmented as part of an obstacle.

Building on the ideas suggested in the monocular obstacle detection literature, we propose a simple yet robust algorithm that only requires access to color images, with no need for odometry sensors or compute-intensive motion detection. Moreover, our algorithm is compatible with smooth floors (such as concrete), textured

Figure 5.7: Examples on floors that have a texture compatible with our obstacle detection algorithms (3 left images, sufficiently small or smooth features). On the right, a tiled hallway on which we could not separate the floor from obstacles.

floors (such as carpet), and some hardwood floors. Our algorithm is however not compatible with floors having edges embedded in them, such as tiles or decorated carpets. Figure 5.7, and a third-party website[6], show examples of floors on which we could successfully apply our algorithm, at various exhibitions of our learning wheelchair. Further research is still required to produce a more robust obstacle detection algorithm.

We first blur the image with a 15×15 Gaussian kernel, then compute the difference between the image and its blurred version (we refer to this as the difference-of-Gaussian method). This produces a *texture* map, a mostly-black image in which edges and rough objects appear brighter. We then apply the difference-of-Gaussian method a second time. While this may seem that we compute the texture of a texture, something of dubious interest, we point out that applying the difference-of-Gaussian method two times also corresponds to identifying regions of an image where the *change in texture* is large. This allows us to detect the boundary between a rough carpet and a smooth wall. Finally, a simple threshold is applied to the second texture map: any value greater than 1, either in the red, green or blue channel, causes the pixel to be treated as an obstacle. We summarize the step of this algorithm, and provide example texture maps, in Figure 5.6.

Once the difference-in-texture map is produced, we clean it up by removing small connected components from the map, that is, groups of less than 50 contiguous pixels that are identified as obstacles. This removes false positives caused by specks of dust and other small debris on the floor. We also evaluated the use of morphological operators, but without much success. Eroding a texture map makes

[6]https://www.humanityhub.net/news/the-hub-hosts-worlds-largest-ai-research-networks-hq-launch/

long but thin (usually very important) edges disappear, while removing small connected components preserves these long but thin edges. Once the texture map is cleaned-up, we assume the floor to be of uniform texture, such as carpet, and measure, for each column of pixels, how many non-obstacle pixels exist between the bottom of the image and the first edge detected by going upwards (see Figure 5.6). This count is then divided by the height of the image in pixels, which produces rough distance readings in the $[0, 1]$ range. In this chapter, we consider 320×240 images, which allows us to produce 320 distance readings per image. To reduce noise and make the learning task slightly easier for the agent, we downsample these 320 distance readings to only 40. Our agent therefore observes vectors of 40 floating-point values between 0 and 1. These distance readings do not have an unit associated with them, and are not corrected for the perspective effect (objects further from the camera appear smaller), but still allow a Reinforcement Learning agent to learn a good policy.

With the sensing side of our agent now described, and its acting side described in Section 5.2, we now introduce our wheelchair navigation task, and detail how we map this task to a Markov Decision Process on which Reinforcement Learning can be applied.

5.4 The Wheelchair Markov Decision Process

In this section, we detail how we connect a motorized wheelchair to a Reinforcement Learning agent. Sections 5.2 and 5.3 explain how commands are sent to the wheelchair for execution, and how to produce observations from a single webcam. This section explains how these observations are used to make Reinforcement Learning safe on the wheelchair, and describes the task to be learned by the wheelchair. We stress the fact that we did not over-simplify the task in any way, nor took any shortcut to obtain more compelling results. For instance, the agent we present in this chapter is general enough to be able to quickly learn relatively random tasks, as proposed by visitors of our lab, without having to tweak how it perceives observations or the backup policy we introduce in Section 5.4.1. An example of a task suggested by a crowd is avoiding being "petted" by people at a local scientific conference.

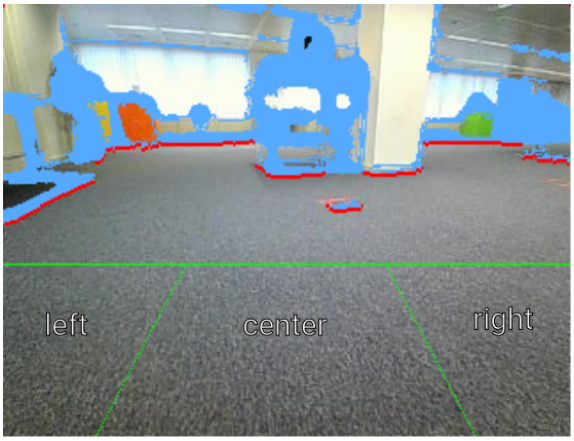

Figure 5.8: View from the front camera on the chair, annotated with detected edges (light blue) and the resulting distance readings (one per pixel column, in red). The backup policy divides that space close to the chair in three portions, and reacts differently according to whether the obstacles are on the right, left or center of that space. The backup policy is inactive when there are no obstacles in any of these three regions.

5.4.1 A Backup Policy to Avoid Obstacles

We begin our description of the Markov Decision Process in which the motorized wheelchair learns with safety considerations. A motorized wheelchair is a highly human-involved and safety-critical robot, as the user is sitting *in* the robot as it is interacting with it. As available from its manufacturer, our wheelchair does not feature any safety mechanisms: it will gladly drive down stairs, or into a wall at 40km/h. Whether the chair is controlled by a person or an agent, preventing it from executing undesirable actions is paramount.

In this section, we detail the design of a safe backup policy that watches over our learning agent. Our motorized wheelchair setting is challenging for the design of a backup policy, as we have no model nor simulator for the task to be learned, which prevents us from using state-of-the-art model-based approaches such as [Berkenkamp et al., 2017]. The use of distance sensors, that produce floating-point readings instead of discrete states, also restricts our choice of algorithm, preventing for instance the use of bayesian modeling [Turchetta et al., 2016]. Based on these two observations, we therefore use *policy shielding*, an approach that consists of manually designing a backup policy that prevents the agent from executing unsafe actions in unsafe states [Alshiekh et al., 2018]. Based on the distance readings obtained from the front-facing camera (see Section 5.3.3), our backup policy prevents the wheelchair from bumping into walls, tables, chairs, and people. Over time, the agent learns to navigate its environment in a safer way, that triggers the backup policy less often. However, as suggested by Alshiekh et al. [2018], we do not intend to ever disable the backup policy.

Our backup policy, that we detail a bit later in this paragraph, is easy to understand and to trust, and has been designed in a task-agnostic way. It takes strong, sudden decisions to prevent the agent from making mistakes. Its behavior does not help learning at all, but our results in Section 5.7 demonstrate that our agent is able to learn to navigate in a room even in the presence of a highly-disruptive backup policy. From an application point of view, these results demonstrate that Reinforcement Learning can be applied even in settings where the backup policy is rigid and kept simple to ensure safety. Our backup policy divides the space in front of the agent in three regions: left, center and right. As shown in Figure 5.8, the regions are defined in a way that approximates perspective, so that the center region roughly corresponds to a projection of where the wheelchair would fit in the real world. Simple thresholding operations are then applied to the 320 distance readings obtained from the front camera. The result of these operations defines the backup policy as follows:

- If every distance reading is above 0.4 (horizontal green line in Figure 5.8), the backup policy does not intervene at all.
- If there is an obstacle on the left, that is, if there exists an $x \in [0, 1]$ such that $d_x < 0.4$ and $d_x > 2.5x - 0.4$ (d_x is the distance reading for the pixel column $x \times 320$, and $2.5x - 0.4$ is the equation of the left sloped line in Figure 5.8), the agent is prevented from turning left.
- The same applies to obstacles on the right, with the condition being $d_x > -2.5x + 2$. If the agent tries to put the joystick in the direction of an obstacle, its decision is overridden by the backup policy and the joystick is put in the neutral (stop) position. The constants appearing in the previous equations have been tuned so that the backup policy triggers when an obstacle is less than about 40 cm from the wheelchair, both in the left, front and right regions of Figure 5.8. This tuning is webcam-specific (it depends on how they are mounted, and what their field of view is), but not task-specific. As such, it has to be done only once.
- If there is an obstacle in the center region, that is, a distance reading is below 0.4 but does not fall in either the left or right regions, the backup policy puts the joystick completely to the right, leading to the wheelchair turning right on its own axis until no obstacle is present in the center region anymore.

The last point revealed to be particularly important in our experiments: as soon as an obstacle enters the center region, the backup policy makes the wheelchair turn left. This is an all-or-nothing decision, that proved to be challenging for the agent to learn to deal with. If the agent nicely follows a wall on its left, and suddenly makes a mistake and puts the wall in the center obstacle region, the chair turns about 180° on its axis and the agent now faces completely away from its destination. With our backup policy, some mistakes are therefore very difficult to recover from. However, we show in the next section and in Section 5.7 that the solutions we propose in the next sections still allow our agent to learn to navigate in an office space, even when our backup policy is used.

5.4.2 Backup Policies with Reinforcement Learning

A backup policy as presented in the previous section, that follows the *policy shielding* formalism, influences the actions executed by the agent, which may have

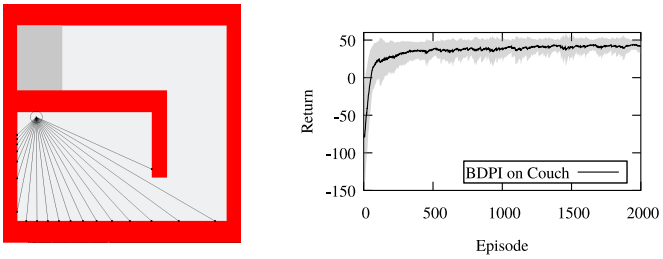

Figure 5.9: *Left:* An agent can rotate and move forward. A backup policy, as described in Section 5.4.1, turns it away from obstacles (red) when any of the sensor readings (bars) is below twice the diameter of the agent, and gives a reward of -2 when doing so. The agent starts at a random position every episode, and has to reach the grey region. *Right:* BDPI (as configured in Section 5.7.5) learns to reach the goal (return of 50) in the presence of a backup policy. The noise of the returns comes from random initial positions that may be too close to obstacles, thus triggering the backup policy.

an effect on learning. In this section, we discuss how the backup policy may change the policy learned by the agent, and what effect on sample-efficiency it may have.

The first question with backup policies is how they influence the policy that will be learned by the agent. We point out that the backup policy is considered to be part of the environment. The agent is not aware of how the backup policy computes its actions. This also means that for the agent, what matters is to learn a policy that is optimal (as good as possible) for the environment, backup policy included. If we assume that 1) the environment is episodic, and resets the agent to some initial position (in a new episode) after some time threshold; and 2) it is possible for the agent to solve the task, for instance reach a goal position, by only visiting a sequence of safe states (states in which the backup policy is not triggered); then the agent is able to learn a policy that solves the task, possibly by taking detours to avoid unsafe states. If no assumption is made on the backup policy, then we cannot prove that learning anything with a backup policy is possible. For instance, with the degenerate backup policy that just forces the agent to remain still in every state, the agent will never learn nor achieve anything. It is therefore critical for the designer of the backup policy to ensure that learning *around it* is possible. The backup policy we present in Section 5.4.1

is an example of such policy, that lets the agent do whatever it wants as long as there is no obstacle too close to it.

To illustrate that BDPI is able to learn a task in an environment with a backup policy, we introduce in Figure 5.9 a small continous-states environment, inspired from our motorized wheelchair setting, and akin to the Virtual Office presented in Section 5.7.3. The agent observes 20 distance readings, measuring the distance between it and the closest obstacle with a 120° field of view. Three actions allow the agent to turn left/right by a few degrees, and move forward by about its diameter. A reward of 0 is given every time-step, except if the agent enters the grey region (+50) or triggers a backup policy (-2). The backup policy is triggered as soon as one distance reading is below 0.05 (about twice the diameter of the agent). When triggered, the backup policy turns the agent away from the obstacle using the "obstacle on the left-middle-right" rules described in Section 5.4.1. In Figure 5.9, right, we show that a BDPI agent is able to learn a policy in this environment, even with a backup policy. The main observation is that the agent *learns to avoid dangerous states and actions*, in this case learns to keep its distances from walls, instead of trying to cut corners. In this environment, the design of the backup policy still leaves a possibility to reach the goal. The agent learns to do it, without being impaired by the actions taken by the backup policy.

The second question with backup policies is their effect on sample-efficiency. Backup policies, even with strong control signals but that leave a possibility to solve the task, allow learning. However, the sample-efficiency of the agent may be affected by the actions taken by the backup policy, and how they are encoded in experiences [Alshiekh et al., 2018; Saunders et al., 2018]. For this discussion, we consider that, at a given time t, the agent wants to execute an action a_t that is deemed unsafe by the backup policy. The backup policy instead executes action a'_t. After some time, 0.33 seconds on our wheelchair, the next time-step starts and the agent observes state s_{t+1}, that is the result of executing a'_t. Because a_t and a'_t are different, the following question arises:

- Should the experience being produced be $(s_t, \mathbf{a_t}, r_t, s_{t+1})$ or $(s_t, \mathbf{a'_t}, r_t, s_{t+1})$. In other words, should the agent be aware of the existence of the backup policy, and the action it executed instead of the chosen one?

The second option, storing a'_t in the experience tuple, is the easiest one as it basically makes the backup policy behave similarly to an off-policy exploration strategy, such as ε-Greedy. We know that Q-Learning is able to learn with this kind of action perturbation [Watkins and Dayan, 1992], and we show in Steckelmacher et al. [2019] that BDPI is robust to such exploration as well. The first

option, storing a_t in the experience tuple instead of a'_t, allows a'_t to be *outside the set of actions* available to the agent. It also makes the backup policy a part of the environment dynamics (not observed by the agent), which may help Policy Gradient approaches.

In our experiments, we opt for storing a_t in the experience tuple, even if the backup policy made the agent execute a'_t instead. The main advantage of this approach is that the backup policy is able to move the joystick in any way it wants, without being restricted to the 4 pre-defined joystick positions made available to the agent as discrete actions. However, storing a_t in the experience tuple leads to a problem that needs to be solved: sometimes, the backup policy has to execute for several time-steps, as the wheelchair may need time to move away from an obstacle in front of it. In this situation, several experiences have states and actions that depend on the backup policy, without the agent being aware that this backup policy exists. We empirically discovered that the more time-steps the backup policy overrides the actions selected by the agent, the lower the sample-efficiency of the agent becomes. Intuitively, every time-step during which the backup policy overrides actions, the agent tries its own actions, but the rewards it receives and states it observes are completely unrelated to them.

Our solution is simple: we *elongate* the time-step when the backup policy overrides the agent's actions. When the backup policy is triggered, time freezes for the agent (the time-step becomes longer, sometimes several seconds) until the backup policy has performed all necessary actions to bring the agent back to a "safe state". The way the agent perceives elongated time-steps is comparable to how top-level policies see options execute, in the Options framework [Sutton et al., 1999], with the key difference that the duration of a backup-policy action, in our case, does not influence the discount factor used by the agent. The agent is therefore fully unaware of how long the backup policy took control of it. Our empirical results, later in this chapter, show that our elongated time-steps without discount factor lead to sample-efficient learning.

After presenting the backup policy we use, we now describe the Reinforcement Learning task to be solved by the wheelchair.

5.4.3 A Wheelchair in an Office Space

This chapter discusses all the real-world issues relevant to our application of Reinforcement Learning to a motorized wheelchair that navigates in an office. After having presented our setting in 5.1, background information on electronics and our robotic platform in Section 5.2, and Computer Vision for backup policies in

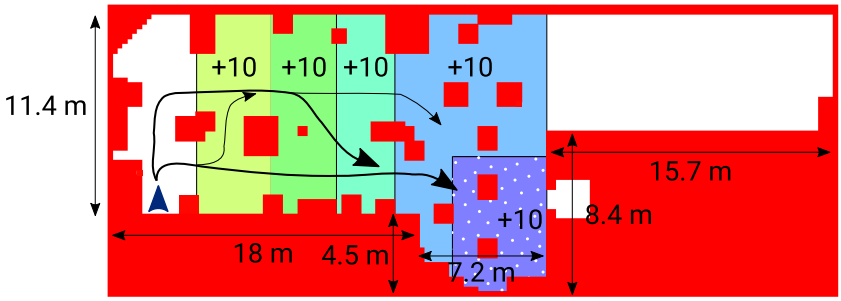

Figure 5.10: Map of the large open-space office in which we perform our experiments. Red regions represent either walls, tables or chairs (any kind of obstacle). We stress that this map does not completely match reality (the space changes as required by its users), and is not provided to the agent. The chair starts on the left end of the space and must reach the dotted zone, while avoiding obstacles on its way. Colored zones indicate when the agent is rewarded (or punished when going in the wrong direction). The agent receives a non-zero reward on average every 20 time-steps. Example paths taken by the wheelchair are drawn, the thicker the line the more frequently visited the path.

Section 5.3, we now formalize our navigation task on a motorized wheelchair as a Reinforcement Learning problem. We designed our problem to be as general as possible, so that the results we present in Section 5.7.2, that show that our agent learns a challenging task in only one hour of wall-clock time on a physical robot, allow us to be optimistic regarding the applicability of Reinforcement Learning to many real-world tasks.

Dynamics

The motorized wheelchair we use is the Invacare TDX SP2 Ultra Low Maxx. The physical environment in which this wheelchair navigates is a standard open-space, as can be found in any large office building. No specific preparation of the space was required prior to our experiments; we solely rely on the the highly adaptive nature of our RL agent to make the best of any space. There are a few pillars, many tables of various heights and lengths, and chairs furnishing the open-space (see Figure 5.1 at the beginning of this chapter). The wheelchair starts on one extreme end of the open-space and tries to reach a goal location, situated around the middle of the space (see the polka dotted zone in Figure 5.10), while avoiding obstacles. The distance separating the starting location and the goal location is roughly 25 meters. A fixed initial position impacts the agent in several ways:

- The agent never spawns near the goal, which makes exploration more difficult than random initial positions;
- In our particular setting, where the agent observes rough distance readings, a fixed initial position reduces state aliasing, as less states are visited by the agent. This helps the agent in the early stages of learning. If more informative observations are provided to the agent, having a fixed position is less important for sample-efficiency. We also note that recent work by Hélène Plisnier, ongoing at the time of finishing this thesis, allows our motorized wheelchair to efficiently learn navigation tasks *from any initial position*, even with state aliasing.
- The policy learned by the agent, because it is Markovian in our case and does not try to learn *sequences of actions*, still works well in every state ever visited by the agent during training.

The episode duration is fixed to 150 time-steps (50 seconds). This is just long enough to allow the agent to reach the target area if it takes the shortest path.

Time lost to hesitation or the backup policy therefore reduces the total return obtained by the agent. While this encourages the agent to reach the goal quickly, truncating episode lengths is known to be challenging to Reinforcement Learning agents [Pardo et al., 2018], and we leave the design of a better episode termination function to future work.

Actions and Reward Function

The actions available to the agent are as follows: putting the joystick forwards, forwards but less so, left or right. The agent has no way to stop, which forces it to constantly move and explore, and has no direct control on the speed of the motors. When the joystick is put in a specific position, the embedded controller of the wheelchair computes a dynamic, time-dependent and user-configurable acceleration scheme for the two motors. Time-steps last 0.33 seconds, except when the backup policy forcibly turns the agent away from an obstacle (see Section 5.4.1). At every time-step, the agent observes a vector of 40 distance readings (floating-point values between 0 and 1, see Section 5.3.3). The rewards are manually provided to the agent by pressing buttons connected to the wheelchair. We motivate our choice of a manual reward function in Section 5.4.5. A reward of +10 is given whenever the agent reaches an intermediate zone closer to the goal location. A reward of -10 is given when the wheelchair enters a zone further from the goal than the current zone. The zones are annotated in Figure 5.10, and roughly align with tables and obstacles, so that identifying in which zone the chair is is easy for a human operator.

Another source of reward is the backup policy. Every time it is triggered (it either turns the agent away from an obstacle, or cancels an action, see Section 5.4.1), a reward of -2 is automatically (not manually) given to the agent. When the backup policy elongates the time-step, the reward is scaled accordingly (-2 per third of a second during which the backup policy is active). We purposely designed a backup policy that is simple, naive and disinterested in efficiency; this consists, with the punishments, in an additional incentive for the agent to try to trigger it the least possible. Moreover, the backup policy is sometimes wrongfully triggered by spurious obstacle detections. This adds to the difficulty of learning. In Section 5.7.2, we show that even though our observations, actions, backup policy and general setting are not designed to make learning easy, our agent manages to reliably learn policies almost as good as human level.

We now discuss two aspects of real-world robotic tasks that do not arise in simulated environments: why we selected to manually reward the chair instead of

computing the reward from observations, and how we reset it after each episode. We believe that these problems arise often in real-world deployments of Reinforcement Learning, and that our solution, simply asking a person to reward and reset the agent, is both simple and made possible by the high sample-efficiency of BDPI.

5.4.4 Resetting the Agent

Resetting a simulated environment is often as simple as clearing a few variables, or restarting a simulator. In the real-world, the agent has to be physically put back in an initial state, either by moving (if the robot navigates its environment), or issuing commands to robotic arms to put them in an original pose. When developing our motorized wheelchair system, we tried three approaches to resetting the agent:

1. The easiest approach is not to reset the agent at all. The initial state of episode $k+1$ is therefore the final state of episode k. This allows the agent to learn cyclic tasks, such as moving as fast as possible in a room while avoiding obstacles. However, this does not allow the agent to learn goal-oriented tasks, such as going somewhere from some initial position. This is therefore an approach that we cannot recommend for real-world deployments of Reinforcement Learning.

2. We evaluated the use of a non-learned controller that is able to take our wheelchair from any point in a room, and bring it to any other point. This controller has to be reliable, but does not have to be optimal, which simplifies its design. However, this approach requires a map of the room, and a method for locating the wheelchair in the room. We were able to provide these components for our office, by measuring it and putting markers on the ceiling that could be observed by the wheelchair using a top-facing camera. However, annotating ceilings is not practical in the applications we envision: the deployment of many wheelchairs in many homes or residences, which would require all these very nice ceilings to be covered in stickers.

3. This led to the last and only possible approach, that we ended up using in our experiments: at the end of an episode, the agent is halted, and releases control of the chair to a human-controlled joystick. A human operator then moves the chair back to its initial position (approximately), with some randomness in its orientation to ensure that general policies are learned.

This last approach is the most practical. First, BDPI learns fast enough that the presence of a human operator is not problematic, as a training session typically lasts only one hour. Second, in the setting we believe model-free Reinforcement Learning is the most desirable, agents that have to be trained in many different locations, having a person reset the agent is much faster than having to develop a *reset controller* for every space in which the agent has to be trained.

We now discuss why we also request the human operator to reward our motorized wheelchair, and why human rewards are possible and desirable in our setting.

5.4.5 Sparse Rewards are Better than Noisy Ones

In real-world settings, designing a reward function is challenging. In this section, we discuss how two reward functions, one dense but noisy and the other sparse but exact, influenced how our motorized wheelchair learned to navigate in an office. While we only have results with our BDPI algorithm, we believe that they are useful for anyone trying to reproduce our experiments.

The goal of our motorized wheelchair navigation task is to reach a particular area in a large office room, full of obstacles. Our first approach to rewarding the agent was to provide a dense potential-based reward, that rewards the agent when it gets closer to the goal, and punishes it when it goes farther. Because time-steps last only 0.33 seconds, a person would be unable to provide this reward signal. To automatically provide it, we placed a Wifi access point at the target location, and used its signal strength (RSSI) as a source of reward. We also experimented with a Bluetooth beacon, again located at the target location, and used its signal strength as the basis of the reward function. However, even though the goal location is at a significant distance from the starting location (around 25 meters), both the Bluetooth and Wifi signal strengths varied very little on the way from the starting location to the goal. Both these signals, reported as integers by the operating system, only had a difference of about 4 between the weakest and strongest signals. Moreover, even though that signal was received by the agent at every time-step of the episode, it was too noisy and poorly informative to allow the agent to learn, with the reported signal strength randomly changing by as much as 2 units every time-step.

We therefore provide the rewards manually using push buttons, which has the significant advantage of not requiring any modification of preparation in the environment, compared to RSSI-based methods that require specific devices to be brought on site, secured in the environment, powered and configured. Because

human operators are unable to provide rewards more than once every few seconds, the wheelchair is rewarded every time it reaches an intermediary area closer to the goal, and is punished when it backtracks (see Figure 5.10). This results in a much sparser reward signal, about one reward or punishment every 15 time-steps, but this reward signal corresponds exactly to the task to be learned and is devoid of noise. This sparse but informative reward signal allowed BDPI to learn the task fairly quickly, as our results in Section 5.7.2 show. The observation that a sparse but exact reward leads to better results than a dense but noisy one is interesting, because much of the current Reinforcement Learning literature considers sparse rewards as being highly challenging. As first described in Chapter 3, and again verified in this chapter, the actor-critic architecture of BDPI, with several critics, seems to allow BDPI to explore well in complicated environments with sparse rewards. That BDPI is robust to sparse rewards is critical for real-world tasks, as having a person provide rewards every now and then is both easy and highly general, but requires a learning algorithm able to cope with sparse rewards.

The description of the Markov Decision Process to be solved by our Reinforcement Learning agent is now complete. In this section and the previous ones, we describe our motorized wheelchair platform, how we produce observations and execute actions, how we ensure safety, and the reward function for the task. We now move to the second part of this chapter, that presents two Reinforcement Learning algorithms that, combined, allow our highly-challenging real-world motorized task to be learned in only one hour.

5.5 Advice at no Cost of Final Quality: the Actor-Advisor[7]

In real-world settings, achieving maximum sample-efficiency is important for the viability of a Reinforcement Learning agent. Sample-efficiency mainly comes from high-quality algorithms, such as Bootstrapped Dual Policy Iteration, for which we show in Section 3.5.1 that it outperforms many state-of-the-art Reinforcement Learning algorithms. Sometimes, additional sources of sample-efficiency are available, such as *domain knowledge*. Domain knowledge is a general term, that basically means that the designer of the agent has some information about the task

[7]The primary investigator of this work is Hélène Plisnier. Every algorithm mentioned in this section has been developed by her. This work is described in this thesis as background information, as our motorized wheelchair application depends on it.

that will be learned, and can infuse some of this knowledge in the agent. Examples of use of domain knowledge are domain-specific feature engineering [Tedrake et al., 2005], the possibility to collect information about the domain to pre-train the agent [Levine et al., 2016], or the design of a domain-specific, highly-informative reward function [Ng et al., 1999].

The main problem with using domain knowledge to increase sample-efficiency is that it requires effort from the designer of the agent. For instance, domain-specific high-level features require careful thinking about what kinds of observations will be useful to the agent. Pre-training requires going on the field and collect data for possibly hours. Reward shaping requires the design of a high-quality reward function, and coding skills to actually implement it in the agent. In Chapter 4, we present a method for addressing partial observability that requires domain knowledge, but so little of it that designing our proposed OOIs takes about 20 minutes per environment. In this section, we introduce the Actor-Advisor, a method that allows to increase the sample-efficiency of the agent, using so little domain knowledge that spending 2 minutes writing a few lines of Python code increases the sample-efficiency of the agent by about 20% in a wide variety of tasks, as illustrated in Figure 5.11.

We begin this section by reviewing Policy Shaping, then present the Actor-Advisor and its implementation in Bootstrapped Dual Policy Iteration (BDPI). We show in Figure 5.11, then in Section 5.7.5, that the Actor-Advisor easily increases sample-efficiency, with minimal engineering effort. More detailed characterizations of the Actor-Advisor, and empirical evidence of its usefulness, are available in two conference papers by Hélène Plisnier [Plisnier et al., 2019a,b].

5.5.1 Policy Shaping

Policy Shaping [Kartoun et al., 2010; Griffith et al., 2013; MacGlashan et al., 2017; Harrison et al., 2018] consists of letting an external advisory policy π_A alter or determine the agent's behavior at acting time. The specific Policy Shaping formula we are considering in this chapter is the one suggested by Griffith et al. [2013]:

$$\pi_{L\times A} = \frac{\pi_L(s_t)\,\pi_A(s_t)}{\pi_L(s_t)\cdot\pi_A(s_t)} \tag{5.1}$$
$$\text{with } \pi_L(s_t)\cdot\pi_A(s_t) = \sum_{a\in A}\pi_L(a|s_t)\pi_A(a|s_t)$$

where $\pi_L(s_t)$ is the state-dependent policy learned by the agent, $\pi_A(s_t)$ is the state-dependent advice, and $\pi_L(s_t) \cdot \pi_A(s_t)$ is the dot product of the two. At acting time, the agent samples an action a_t from the mixture $\pi_{L\times A}$ of the agent's current learned policy $\pi_L(s_t)$ and the external advisory policy $\pi_A(s_t)$, instead of sampling only from $\pi_L(s_t)$. Executing actions from this mixture allows the advisor to guide and improve the agent's exploration. When the advice $\pi_A(s_t)$ is deterministic, i.e., a vector of zeroes and one 1 for the action advised, the agent's policy $\pi_L(s_t)$ is fully overridden and the agent is forced to execute the advised action. However, if the advice $\pi_A(s_t)$ is stochastic, i.e., a vector of floats summing to 1, then the agent's policy $\pi_L(s_t)$ has an impact on the action selection. These two possible kinds of advice are both highly relevant to Reinforcement Learning in real-world settings: deterministic advice can be used to forcibly prevent the agent from executing damaging actions, contrary to reward shaping that intervenes a-posteriori; and stochastic advice can be overcome by the agent, such that even sub-obtimal advice helps in the early stages of learning, but ultimately does not reduce the quality of the learned policy. Because stochastic advice can be overcome by the agent, simple and easy-to-design advice can be used. In this paper, we use simple stochastic advice in our wheelchair experiments (see Section 5.4.3), as it allows us to entice our agent to explore better, without limiting the quality of the policy it learns.

5.5.2 Advice Influences Acting and Learning

The original implementation of the Actor-Advisor [Plisnier et al., 2019a] consists of extending Policy Gradient [Sutton et al., 2000] to allow the Policy Shaping equation (5.1) to be used without impairing learning. The general idea behind the original Actor-Advisor is to modify the parametric policy (usually a neural network) so that it takes as input an advice vector $\pi_A(s)$ in addition to the state s, and directly produce as output the result of Equation 5.1. By computing the Policy Shaping equation in the neural network, it becomes possible to sample actions directly from the probability distribution produced by the network, without having to modify these probabilities. As we explain in Section 2.7.3, any modification of the probabilities output by the network, before sampling an action, leads to unstable learning. The Actor-Advisor is therefore a method that allows Policy Shaping to be used with a Policy Gradient actor in a stable way [Plisnier et al., 2019a].

Because the Policy Shaping equation is directly implemented in the policy, for which the Actor-Advisor computes a gradient, we can intuitively notice that Policy

Shaping influences both how the agent *acts*, as the equation is able to increase or decrease the probability of certain actions according to advice, and how the agent *learns*, as the advice is processed by the policy network, and therefore influences its gradient, thus the way it learns.

We now propose a second implementation of the Actor-Advisor, that replaces Policy Gradient with Bootstrapped Dual Policy Iteration. The main reason for this replacement is sample-efficiency. BDPI is much more sample-efficient than Policy Gradient (see Section 3.5.4). Because the purpose of the Actor-Advisor is to increase sample-efficiency, it would be difficult to justify applying it to anything but the most sample-efficient Reinforcement Learning available. We now discuss how advice influences both *acting* and *learning* of the *actor* and *critics* of BDPI.

5.5.3 BDPI with Advice at Acting Time

We first consider that the BDPI actor remains purely state-dependent, and does not observe any form of advice. This ensures that the addition of advice does not change the shape of the BDPI actor network, does not make its training more difficult or unstable, and does not alter its performance in any way. This is also motivated by the fact that Plisnier et al. [2019a] mix advice with the policy as late as possible in the neural network. The BDPI actor therefore learns $\pi_L(s_t)$, and has no extra input for advice. At acting time, the actions executed by the agent are therefore sampled from the mixture of the learned actor $\pi_L(s_t)$ with the advice $\pi_A(s_t)$, following the Policy Shaping formula shown in Equation 5.1. Mixing π_L and π_H at acting time is possible because BDPI is off-policy, contrary to Policy Gradient (see Section 3.5.4). The advice vector $\pi_A(s_t)$ is a probability distribution over actions summing to 1.

Because BDPI is an off-policy algorithm (see 3.5.4), simply combining the output of the actor with the advice vector, and sampling actions from the result, maintains the convergence properties of BDPI. We show in Plisnier et al. [2019b] that this is indeed the case. However, we also show that slightly modifying the actor update rule to make it include advice in the learning mechanism increases the quality of the executed policy in the early stages of learning. At acting time, we therefore store the advice vector received by the agent at each time-step, which leads to the agent storing $(s_t, a_t, r_t, s_{t+1}, \pi_A(s_t))$ tuples in its experience buffer. We now discuss how the actor learning rule can leverage this advice to increase the quality of the actual behavior of the agent in early learning stages.

5.5.4 BDPI with Advice at Learning Time

While the BDPI critics play no role at acting time, they are updated, along with the actor, at learning time. Fortunately, the critics being off-policy, no special consideration is needed when learning Q^* from experiences generated by the mixture of the actor and advice. The actor, however, can benefit from an involvement of the advice in its learning mechanism. We call *learning correction* the change we make to the BDPI actor learning rule to leverage advice.

Intuitively, in settings where advice is provided continuously (that is, advice is produced by a function that is queried every time-step, as opposed to a human that may leave at some point), such as in our motorized wheelchair setting, the objective is to maximize the expected cumulative reward obtained by *the agent*, that executes a mixture of the BDPI actor *and* the advisor. We therefore want the actor to learn a policy that, *when combined with advice*, is optimal for the task. We formalize this objective with Equations 5.2 and 5.3, shown below, that were first introduced in Plisnier et al. [2019b]:

$$\pi(s, \pi_A(s)) \leftarrow \Gamma(Q(s)) \qquad \text{converges to optimal policy} \tag{5.2}$$

Starting from Equation 5.2, we isolate π_L, the actor of BDPI, and derive an updated learning rule that makes the actor compensate for potentially-suboptimal advice:

$$\begin{aligned}
\pi(s, \pi_A(s)) &\leftarrow \Gamma(Q(s)) \\
\frac{\pi_L(s)\pi_A(s)}{|\pi^L(s) \cdot \pi_A(s)|} &\leftarrow \Gamma(Q(s)) \\
\pi_L(s)\pi_A(s) &\leftarrow \underbrace{\Gamma(Q(s)) \times \overbrace{|\pi_L(s) \cdot \pi_A(s)|}^{\text{a scalar}}}_{\text{a vector}} \\
\pi_L(s) &\leftarrow \frac{\Gamma(Q(s)) \times |\pi_L(s) \cdot \pi_A(s)|}{\pi_A(s) + \varepsilon}
\end{aligned} \tag{5.3}$$

with the fraction an element-wise division between two vectors, and ε a small positive value that prevents a division by zero if the advice contains a zero probability for any of the actions.

Intuitively, Equation 5.3, the actor learning rule that we use instead of the standard BDPI one, moves the actor in the direction of the greedy function of a critic, as described by Steckelmacher et al. [2019], with a force or *pull* that influences how much the greedy function is followed, depending on how much it agrees with the advisor. The more the advice differs from the actor, the more the actor will follow the greedy function (and thus pull away from the advice). Such a pull allows the agent to compensate for bad advice, by learning an actor π_L that, when combined with the faulty advice, still leads to a good policy.

Related to our last remark, that the actor π_L with the learning correction learns a policy *that compensates for sub-optimal advice*, we emphasize that this does not invalidate our claim in Plisnier et al. [2019a] and Plisnier et al. [2019b] that the Actor-Advisor allows optimal policies to be learned even with bad advisors (and no learning correction). In the late learning stages, the Policy Gradient or BDPI actor becomes highly deterministic, and therefore defines (in the limit) the entirety of the behavior of the agent. In late learning stages, bad advice is therefore automatically overcome by the actor, which allows the optimal policy to be executed by the agent even with incorrect advice [Plisnier et al., 2019a]. Our learning correction improves the behavior of the agent in the *early stages* of learning, when the policy is not yet deterministic. By making it compensate for bad advice, we ensure a rapid improvement of the behavior of the agent, even when sub-optimal advice is used. We show in Plisnier et al. [2019b] and Figure 5.11 that the learning correction indeed increases the performance of the agent in the early stages of learning (after about 60 episodes), and provides a significant advantage over not using advice.

5.5.5 Summary of BDPI with Advice

We summarize our implementation of the Actor-Advisor with BDPI, that we call BDPI with Advice, in the pseudocode below:

BDPI with Advice is an extension of BDPI that requires only few changes to some formulas, and the advice we use in Section 5.7 simply consists of always encouraging the agent to go forwards. That this simplicity allows measurable increases in sample-efficiency, and, as shown by Plisnier et al. [2019a,b], many transfer learning and learning from human advice tasks to be learned, demonstrates that elegant algorithms such as the Actor-Advisor can sometimes have a wide range of applications.

We now propose additional improvements to Bootstrapped Dual Policy Iteration, part of our personal research line, that further increase sample-efficiency, and allow tasks in the real world and in real time to be learned effectively.

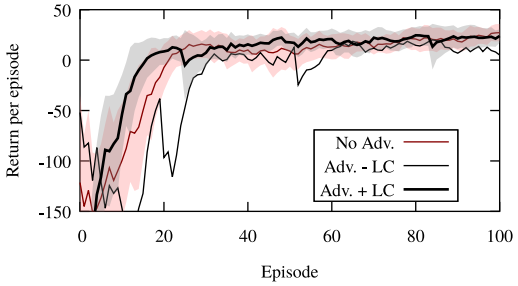

Figure 5.11: Cumulative reward per episode for agents with and without the Actor-Advisor, on the simulated version of our wheelchair-in-an-office task, described in Section 5.7.5. Applying a learning correction, that encourages the BDPI actor to diverge from the advice it receives, increases the quality of the policy in the early stages of learning. This figure shows the BDPI agent described in Section 5.7.5 without advice (No Adv.), with simple advice that encourages it to go forwards (Adv. + LC), and with that advice but no learning correction (Adv. - LC). Every curve show the average of 4 runs.

5.6 Asynchronous Parallel BDPI

In this section, we present solutions to many challenges that derive from the real-world real-time setting, in which time flows continuously, is not simulated, and can therefore not be stopped whenever the agent may want to do something else than interact with the environment. While this section focuses on time-related issues in the real world, Sections 5.4 and 5.7 discuss practical details about producing observations and rewards, safety, and resetting real-world environments.

While a discrete-time Markov Decision Process assumes clear transitions from a state to the next (see Section 2.2.1), robots move in a continuous way, as long as torque is produced by some motors or inertia makes the robot move. This has two implications, that need to be addressed in BDPI, or any other Reinforcement Learning algorithm applied to robots:

1. There is no way to *pause* the environment to give time to the agent to learn, either after every time-step or between episodes.

Algorithm 2 BDPI with Advice. Training the critics is not influenced by advice, and therefore omitted from this pseudocode.

procedure ACT(s_t)
  Obtain advice $\pi_A(s_t)$ as a vector of $|A|$ real values
  Execute $a_t = \frac{\pi(s_t)\,\pi_A(s_t)}{\pi(s_t)\cdot\pi_A(s_t)}$ (the denominator is a dot product)
  Store $(s_t, a_t, \pi_A(s_t), r_t, s_{t+1})$ in the experience buffer
end procedure
procedure UPDATEACTOR
  Sample states s, and advices $s\pi_A(s)$ from the experience buffer
  Query a randomly-chosen critic i for $\Gamma(Q_i(s))$
  $\pi_L \leftarrow \frac{\Gamma(Q_i(s))\times|\pi(s)\cdot\pi_A(s)|}{\pi_A(s)+\varepsilon}$
  Update the actor π towards π_L with Equation 3.2
end procedure

2. The environment does not wait for an action to be selected before changing state, which means that the agent will always execute actions based on outdated information.

The last point is the well-known delay effect in dynamics systems, well-studied by control engineers and roboticists [Youcef-Toumi and Ito, 1990, for instance]. Research on delay effects in Reinforcement Learning is much sparser, though. We now describe how we address the two issues listed above in BDPI, and compare our solutions to related work.

5.6.1 Parallel BDPI

The original BDPI algorithm, presented in Chapter 3, is sequential in nature. Every time-step, the critics are enumerated, updated, and each updated critic is then sequentially used to update the actor. We propose a multiprocessing infrastructure that consists of three components.

1. A shared experience buffer, a shared actor and N shared critics (with $N = 16$ in our experiments). Our implementation being process-based, these three entities lie in shared memory segments.

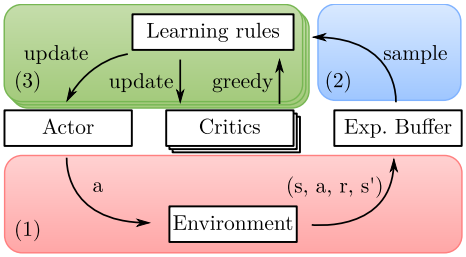

Figure 5.12: Overview of the Asynchronous Parallel BDPI architecture. Rectangles represent objects (the actor, critics, environment and experience buffer). Rounded regions identify processes or threads: *(1)*, the thread that executes actions in the environment and stores experiences; *(2)*, the thread that submits training jobs to the workers; *(3)*, the workers processes, that train the actor and critics on samples of experiences.

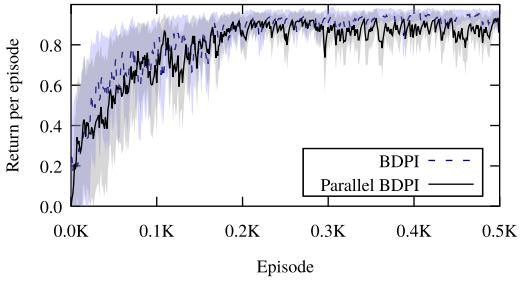

Figure 5.13: Cumulative reward obtained by a sequential implementation of BDPI, and our Parallel BDPI, on the pixel-based *Hallway* environment introduced in Section 3.5.3. The two learning curves are indistinguishable, which shows that parallelism does not reduce sample-efficiency.

2. A main process, that queries the actor for actions and interacts with the environment.

3. M worker processes, that execute training *jobs* from a shared job queue. These processes are the ones responsible for training the actor and critics on batches of experiences.

The main and worker processes all have access to the shared experience buffer, actor and critics. Every time-step, the main process observes a state, asks the actor for an action, executes the action, then submits L jobs to the job queue. The main worker is then free to move on to the next time-step, while the workers process the jobs. Figure 5.12 graphically depicts the communication between the main process and worker processes.

A job consists of the randomly-selected index of a critic, and a randomly selected set of indexes of experiences in the experience buffer (so, not full experiences, to avoid expensive memory copies in the main process, only pointers to them). When a worker process picks up a job, it fetches the experiences from the experience buffer (1024 in our experiments), performs several (2 in our experiments) training iterations on the critic with these experiences, then updates the actor towards the greedy policy of the critic on these experiences. We do not use locks when updating the neural networks, as suggested by Recht et al. [2011, the Hogwild distributed neural network training algorithm]. In Section 5.6.3, we discuss the possible race conditions in our implementation of Parallel BDPI. In our experiments, we set M, the number of worker processes, to the amount of cores available in our machine (32). We set N, the number of critics, to 16, as in Section 3.5. We did not fine-tune this parameter. We set L to 32, such that each of the 32 workers receive a job every time-step, ensuring full CPU utilization even in our slow wheelchair environment (one time-step every 0.33 second). Setting L to 32 leads to every critic being updated by at least two workers at any moment in time, and the actor being constantly updated by 32 workers. We show in Figure 5.13 that the addition of parallelism does not impair learning, which demonstrates that updating neural networks without locks is feasible. We also report that all our 32 cores are always utilized at 100%, and that the learning speed, measured in experiences processed per second, scales almost linearly with the amount of worker processes. The implementation of Parallel BDPI in the supplementary material has command-line parameters for M, N and L, allowing anyone to tune these parameters to the number of cores available on their machine.

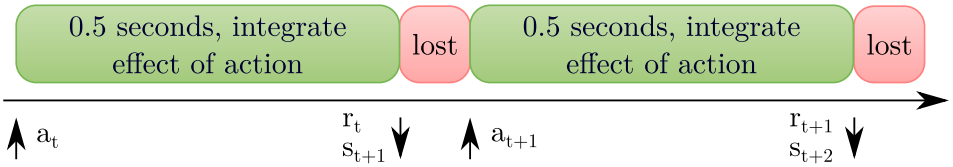

Figure 5.14: Any delay in the action selection (white boxes), due to synchronous learning or gradient updates, decreases the portion of real-time (black boxes) that influences observations and rewards, and as such the task being learned. This problem only occurs on asynchronous environments, such as robots, and the OpenAI Universe,[8] that has indeed now been replaced with a synchronous alternative.

5.6.2 Asynchronous Parallel BDPI

To prevent the job queue described in the previous section becoming too large, its size is limited to $2 \times M$, two times the number of worker processes used by Parallel BDPI. When the main process tries to submit a job while the queue is full, it blocks until the worker processes catch up and remove an element from the queue. This keeps the main process in lock-step with the worker processes, and prevents it, in fast simulated environments, from executing millions of time-steps before the actor and critics have any chance of being updated.

This lock-step is however problematic in real-world robotic tasks, where the time-step has a fixed duration, and the agent needs to meet soft real-time constraints. We therefore introduce yet another parallel component: the main process consists of two threads, a main one that executes actions in the environment, and a second one, that continuously generates and pushes jobs in the job queue (see Figure 5.12). When the job queue is full, the main thread continues executing actions in the environment (preserving the behavior of the agent), while the second thread waits until space is available in the queue before submitting new jobs. This architecture is comparable to the Asynchronous actors of current PPO and ACKTR implementations [Mnih et al., 2016; Schulman et al., 2017; Wu et al., 2017], but somewhat reverses it: instead of many asynchronous actors and a single parameter server, which would require several robots in the real world [Gu et al., 2017a], we use a single actor and many worker processes, that constantly update the actor and critics. Our objective is also completely different: A3C and its successors use

[8] https://github.com/openai/universe

several actors to generate as many experiences as fast as possible, to maintain reasonable run-times in simulated environments at the expense of sample-efficiency, while Asynchronous Parallel BDPI sips through samples, at a rate of 3 samples per second on our wheelchair, but intensively and repeatedly learns from every sample as often as possible, leading to extremely high sample-efficiency.

Making acting and learning fully asynchronous also has the benefit of ensuring that the actor will always quickly select an action in any state given to it. As shown in Figure 5.14, in real-world environments where time cannot be stopped, any delay between the production of an observation and the beginning of the effect of the action is lost to the agent. Not only does this delay lead to oscillation and instability [Youcef-Toumi and Ito, 1990], it also decreases the *integration period* during which effects of the agent's decision influence its reward. Taking our motorized wheelchair as an example, time-steps occur every 0.33 second. If the agent were to take 0.3 second to select an action, this action would execute only for 0.03 second before a new state and a reward would be produced. This tiny duration does not allow the action selected by the agent to have any meaningful effect on the environment. Asynchronous Parallel BDPI, by having a thread dedicated to selecting actions, and a fast and lean action selection mechanism (simply querying an actor, as opposed to planning in a simulator, or processing Q-Values), ensures that the reaction time of the agent to new states is minimal. This drastically reduces the *lost* portions of Figure 5.14, and allows our agent to learn on a physical robot.

5.6.3 Race Conditions in Asynchronous Parallel BDPI

Any component represented in Figure 5.12, that can be accessed by several threads or processes, may lead to a race condition leading to invalid data being written in memory. A detailed introduction to race conditions and how to avoid them is outside the scope of this thesis, but we provide, for every component shared between processes or threads, a discussion of its possible race conditions and their effect on the design of our algorithm.

Experience Buffer

The experience buffer stores states, actions, rewards and next states. It is implemented as a circular buffer, one big PyTorch tensor (an array of floating-point numbers) for the states, one for the actions, and so on. The agent also maintains a pointer to the current insert position in the circular

buffer, and a count of how many experiences are in it. The tensors are shared between the main and worker processes. The insert point and count variables are shared, in the main process, between the main and job-queuing threads. These two threads use locks to ensure that only one of them accesses the experience buffer at once. However, we do not use locks to protect the tensors: for increased compute efficiency, we accept that a race condition exists in which a worker process samples an experience *exactly at the insertion point*, that may have moved since its job has been submitted. In this case, that one sample among many in a batch of samples contains invalid data. We argue that this invalid sample has a negligible effect on the training of the actor and critic, compared to all the other samples in the batch. In Figure 5.13, we provide an empirical argument for our claim.

Critics

The critics (neural networks) have weights shared among all the worker processes. Other data, such as the state of the Adam optimizer or actual mini-batches of training data, are not shared. Following the recommendation of Recht et al. [2011], we do not try to avoid race conditions on the weights of the neural networks. Every training epoch on a neural network only marginally changes its weights, so mis-ordering or missing the update of a particular weight only has minimal impact on the training procedure.

Actor

The actor, more precisely its weights, is shared between the worker processes and the main process, that uses it to produce actions to execute in the environment. As with the critics, lock-free concurrent updates of the weights of the actor lead to unproblematic race conditions, that we do not try to address. Another race condition exists: the weights of the actor may, at the same time, be used by the main process to produce an action, and updated by a worker process. However, following the same argument as for the critic, any change in the weights of a neural network is very small, and has minimal impact on its output. We therefore accept the race condition and do not introduce a lock to avoid it.

We now demonstrate that Asynchronous Parallel BDPI allows the wheelchair described in Sections 5.2 to 5.4 to learn to reach specific locations in a large cluttered room, based only on simple first-person sensors. This challenging task can be learned reliably, every run, in about one hour, 11.000 experiences, a tiny amount

considering the complex dynamics of the environment, and the high-dimensional observations given to the agent.

5.7 Experiments

In this section, we detail how our contributions, presented in Section 5.2 to 5.6, allow a Reinforcement Learning agent to learn to navigate in an office space in one hour. This result, combined with the generality of our model-free, robust, and easy-to-configure method, demonstrate that Reinforcement Learning can now be applied to challenging tasks, defined by people, that involve human spaces. Our agent learns fast enough to allow it to be tailored to every individual user, which allows robots that are *designed once, taught everywhere* to be deployed in the real-world.

5.7.1 Algorithm Configuration

Before presenting our results, we detail the BDPI hyper-parameters that we used in our experiments. We stress that these hyper-parameters have not been systemically optimized for our real-world motorized wheelchair setting. We do not leverage any super-computer to compute absolute-best hyper-parameters over several months. With physical robots, hyper-parameter optimization is almost impossible, and stable and robust algorithms (such as BDPI) are paramount, as any failed or disappointing run is time and money lost. In Section 5.7.3, we introduce a simulated environment, sharing some properties of our wheelchair platform, in which we compare BDPI to state-of-the-art Reinforcement Learning algorithms, and demonstrate the robustness of BDPI to its hyper-parameters.

The agent runs on an AMD Threadripper 2990WX machine, with 32 physical cores running at 3.6 Ghz. At the time of writing, this machine costs slightly under 2800€, and the cost of such compute power will only decrease over time. The agent learns with our Asynchronous Parallel BDPI algorithm (see Section 5.6), using one actor and 16 critics, as in all our experiments of Sections 3.5 and 5.7.5. The actor and the critics are all feed-forward neural networks, with 40 inputs (one per state variable), a single hidden layer of 512 neurons, with the ReLU activation function, and 4 outputs (one per action). The agent executes 3 time-steps per second. Asynchronously from execution, the agent continuously learns using 32 workers. Every worker repeatedly samples 2048 experiences from the shared experience buffer, performs 2 training iterations on a randomly-selected

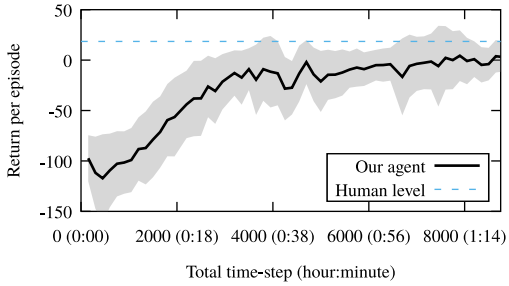

Figure 5.15: Return per episode over time-steps (time) obtained by our Asynchronous Parallel BDPI agent with the Actor-Advisor, averaged over 3 runs on 3 different days. In about 1 hour 15 minutes, our agent successfully reaches the target area, while avoiding obstacles. The returns obtained by the agent are almost as high as what a human expert obtains. We stress that the expert's human eyes provide high-quality binocular color vision and good spatial awareness, while our agent only observes first-person distance readings as described in Section 5.3.3.

critic, then updates the actor. The actor and critic updates each consist of 20 gradient steps on their respective neural network, optimizing the Mean Squared Error loss, with the Adam optimizer [Kingma and Ba, 2014] and a learning rate of 0.0001. Both the actor and critic learning rates that appear in Sections 3.3.1 and 3.3.2 are set to 0.1. Our agent also uses the Actor-Advisor (see Section 5.5) to improve its performance in the early stages of learning. Simple advice, that consists of encouraging the agent to go forward, is provided in the form of a probability distribution: $(0.2, 0.2, 0.4, 0.2)$ (the third action is going forward) at every time-step, regardless of the state. Such advice is easy to produce, and already provides a measurable sample-efficiency boost, as we now demonstrate.

5.7.2 Results on the Motorized Wheelchair

Figure 5.15 shows that *our Reinforcement Learning agent is able to learn our navigation task in a bit more than one hour.* The thin shaded region in our plot demonstrates that all our three runs, using different random seeds, have learned a policy of comparable quality. No run has failed to learn a good policy. To more precisely assess the quality of the learned policies, we also produced expert trajectories (*Human level* line in Figure 5.15). The expert performed 10 episodes,

of which we report the average cumulative reward, and had access to the same action space (four buttons), reward function and initial position as the agent. The expert is familiar with the wheelchair, its dynamics and the backup policy, and is therefore able to precisely and accurately solve the task. The expert level emphasizes the complexity of our navigation task, as even a trained person is unable to reliably collect all 50 reward points (see Figure 5.10), while avoiding triggering the backup policy.

These results show that our Reinforcement Learning system is able to quickly and reliably learn high-quality policies on a physical robot, without any model, pre-training, domain-specific knowledge, or high-level task-specific feature engineering. We now introduce a simulated environment, comparable to our wheelchair setting but easier to learn in, in which we show that state-of-the-art Reinforcement Learning algorithms are much less sample-efficient than BDPI, and could therefore not be used on physical robots in the same practical way.

5.7.3 The Virtual Office Environment

Because comparing algorithms and hyper-parameters on a physical robot, requiring hundreds of runs of at least 15K time-steps, is not practical, we introduce a simulated task that presents many of the challenges of our real-world motorized wheelchair setting. Inspired from the large office space in which we conduct our wheelchair experiments, shown in Figure 5.10, we design our simulated task around a similar map, shown in Figure 5.16. The environment is described as follows:

Dynamics The agent is a zero-surface point that can be in any continuous location in the room. Once built from its tabular representation, the map fits in a 1-by-1 unit square. The agent is able to rotate about its axis by 0.1 radians increments, and move forward 0.01 units at a time.

Partial Observability The actual wheelchair and Virtual Office environments we consider present numerous partially-observable aspects. The most important one is that the agent does not observe its absolute location in the room, only distance sensors. Any corner, anywhere in the room, looks the same to the agent. The second source of partial observability is the hidden dynamics of the wheelchair (see Section 5.1.1), that exhibits acceleration curves, inertia, and past-dependent reactions to actions. In the Virtual Office, we only

consider the first source of partial observability, the use of distance sensors. Our simulated dynamics are completely Markovian, as they implement no inertia or acceleration curves.

Observations 20 or 80 distance sensors, covering a field of view of 120 degrees (the same as our webcam) in front of the agent, measure the distance between the agent and the closest obstacle (in the $[0, 1]$ range). To make these distance readings closer to what a webcam produces, we provide as observation to the agent $\log(1 + d_i)$ for every distance exact distance reading d_i.

Actions 3 actions are available, turn left/right by 0.1 radians, or go forward 0.01 units. In the map of Figure 5.16, this leads to an agent moving forward and turning at about the same rate as our motorized wheelchair in the real office environment we consider.

Initial State The agent always starts at the position shown in Figure 5.16, to match the initial position we consider in our wheelchair task, with a random initial orientation, to ensure that the agent does not simply learn a sequence of moves.

Termination Function The episode ends after 100 time-steps.

Reward function If the action executed by the agent would lead to it entering an obstacle, the action is cancelled and a reward of -1 is given. Otherwise, the reward of the agent depends on its distance to the gray region in Figure 5.16, and is computed as 100 times the change in distance that the action caused. Because the speed of the agent is 0.01 (0.01 units traversed per time-step), the agent therefore receives a reward of +1 every time-step it moves directly towards the gray region, -1 if it goes in the opposite direction, and values closer to zero when it takes a more tangential path. An additional reward of +50 is given when the agent enters the gray region in Figure 5.16. Because the episodes only last 100 time-steps, and the agent moves slowly, reaching this reward is not possible from every initial orientation. It also requires the agent to make no mistake at all during the episode. In Section 5.7.4, we show that only BDPI manages to learn a policy good enough to reach the +50 reward. Neither PPO nor ACKTR managed to achieve this.

We designed this virtual environment to be an approximation of our robotic task, but not as a simulator of our wheelchair task. The virtual environment has no inertia, no complex dynamics, and provides robust distance readings. Its

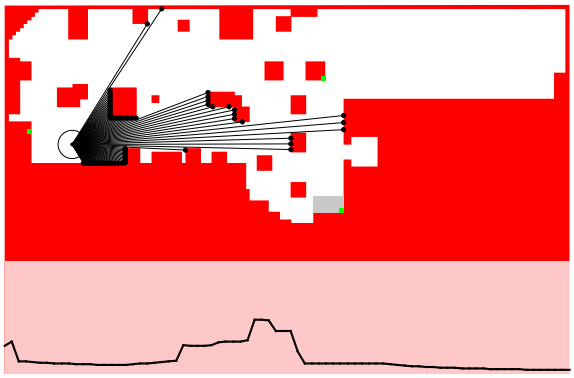

Figure 5.16: *Top:* Our Virtual Office environment representing an open-space cluttered with furniture. The black circle is the agent, and the lines coming from it represent the readings of its distance sensors. The gray area is the target region. This environment cannot be used as a map to help the motorized wheelchair to solve our real-world navigation task, as the wheelchair does not observe its current position, and the map quickly becomes outdated when people move tables and objects. *Bottom:* Distance readings as observed by the agent (sensor index on X, sensor value on Y, between 0 and 1). Making decisions purely on such observations is extremely challenging.

reward is dense, informative, and devoid of noise. This makes it impossible to use for simulation-based pre-training or planning.

We now compare BDPI to two state-of-the-art model-free Reinforcement Learning algorithms for discrete actions, PPO and ACKTR, in the Virtual Experience Center. In Section 3.5, we already demonstrate the superiority of BDPI to these algorithms (and Bootstrapped DQN) in four environments. But to further illustrate the need for a new model-free Reinforcement Learning algorithm, BDPI, on our wheelchair, we also compare it to alternative approaches on an environment that resembles our wheelchair setting more.

5.7.4 BDPI outperforms PPO and ACKTR in the Virtual Office

We compare BDPI to PPO [Schulman et al., 2017] and ACKTR [Wu et al., 2017] in our Virtual Office environment. These two algorithms are considered state-of-the-art in discrete-action model-free Reinforcement Learning. More recent algorithms, such as the Soft Actor-Critic [Haarnoja et al., 2018], are challenging to combine with discrete action. In this section, we configure the Virtual Office to allow for episodes of 150 time-steps, as opposed to 100 time-steps in Section 5.7.5. This leads to higher cumulative rewards, and gives more time to the agent to reach the +50 reward described in Section 5.7.3. Shorter episodes, as in Section 5.7.5, lead to a setting that is more comparable to our physical wheelchair setting.

In our experiment, BDPI is configured as in Section 5.7.2, with the following exceptions: the Virtual Office produces 20 sensors readings instead of 40 on the wheelchair, and we use a batch size of 1024 instead of 2048. This configuration has been selected after a few informal runs of BDPI in the Virtual Office. We did not perform any extensive hyper-parameter search. In Section 5.7.5, we show that BDPI is robust to its hyper-parameters, and that even changing them significantly only minimally impacts BDPI. In this section, we also consider Parallel BDPI, not Asynchronous Parallel BDPI. Practically, after every time-step, the agent updates 8 randomly-selected critics out of 16. Not using the asynchronous version of BDPI is important in fast simulated environments, that would execute much faster than BDPI can learn. This also ensures that our results do not depend on the speed of the machine their are obtained on.

The implementations of PPO and ACKTR come from the Stable Baselines [Hill et al., 2018]. We used neural networks of the exact same shape as BDPI (one hidden layer of 512 neurons). Most of their hyper-parameters have been kept to the default values recommended by the Stable Baselines authors, except: the

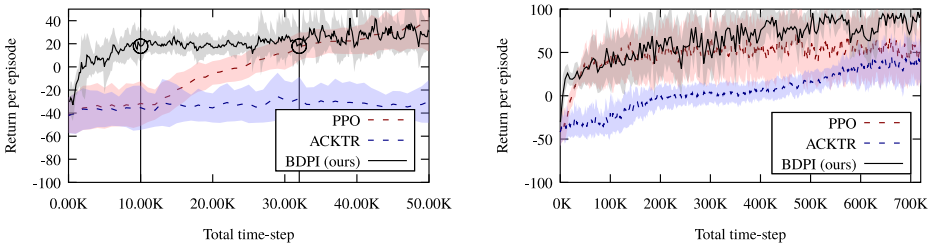

Figure 5.17: Comparison of BDPI to PPO and ACKTR in our Virtual Office environment, with episodes lasting 150 time-steps. Every curve is the average of 8 runs. *Left:* In the early stages of learning, BDPI exhibits significantly higher sample-efficiency than the other algorithms, and learns to move towards the goal while avoiding obstacles in about 10000 time-steps (70 episodes), while PPO needs 3 times more time-steps to do the same. *Right:* When considering long training sessions, the policy learned by BDPI further improves, remains above the learned policies of PPO and ACKTR, and achieves significantly higher end performance. BDPI's sample-efficiency therefore *comes at no cost of top-performance.*

discount factor γ is set to 0.999, to match BDPI; the number of steps per batch-size is set to 150, the length of the episode, which maximizes sample-efficiency; and both PPO and ACKTR are allowed to use *16 replicas of the environment.* We found that using replicas of the environment, something impossible to do on robots, largely increases the sample-efficiency and exploration quality of PPO and ACKTR. By allowing these two algorithms to use replicas, something that BDPI does not need, we favor PPO and ACKTR. Even when favoring PPO and ACKTR in such an unrealistic way, that could never be applied to real robots, BDPI still outperforms them.

In Figure 5.17, we show that BDPI significantly outperforms both PPO and ACKTR in the Virtual Office environment. In the early stages of learning, BDPI achieves significantly higher sample-efficiency, and learns a good policy 10 times faster than PPO, and about 100 times faster than ACKTR (that is particularly sample-inefficient on this task). In the later stages of learning, the policy learned by BDPI continues to improve, and gets to the point where it can reliably obtain the +50 reward detailed in Section 5.7.3. The BDPI curve is always on top of the PPO and ACKTR ones. This demonstrates that BDPI dominates the other

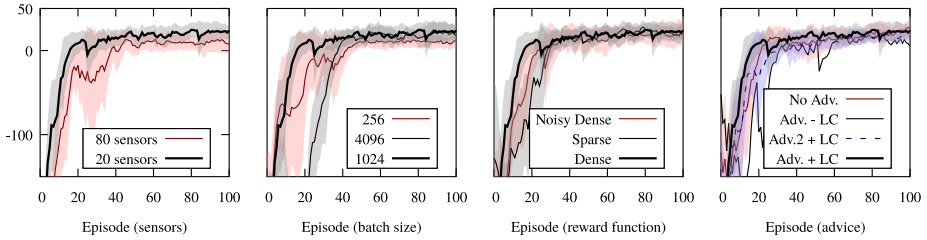

Figure 5.18: Return per episode obtained by our Asynchronous Parallel BDPI agent, with Advice, with various hyper-parameters modified: number of distance sensors (1), number of experiences sampled from the experience buffer for every critic update (2), reward function (3), kind of advice and learning correction (4). See text for details.

two algorithms. Whether sample-efficiency or final quality have to be maximized, BDPI is the algorithm of choice.

We now evaluate the impact of several hyper-parameters of BDPI in the Virtual Office, to illustrate its robustness to its hyper-parameters, and motivate the set of hyper-parameters used in Section 5.7.1.

5.7.5 Exploring Hyper-Parameters in the Virtual Office

In this section, we start with a BDPI agent configured as in Section 5.7.4 (the thick black line in Figure 5.18), and vary several of its hyper-parameters. We also consider episodes that last 100 time-steps, instead of 150 in Section 5.7.4. Such episodes are too short for the agent to ever reach the +50 reward zone described in Section 5.7.3, that does not exist in our motorized wheelchair setting, and better match the actual maximum distance that our wheelchair can travel during one of its episodes.

Varying several hyper-parameters of BDPI leads to the following observations:

Robustness to hyper-parameters Our first observation is that there is no dramatic difference between the performance of our agent with varying hyper-parameters. This illustrates the robustness to hyper-parameters of BDPI. Combined with the very small error margins in Figure 5.18, as no run ever fails to learn the task, and the same robustness we observed on the actual wheelchair, this clearly shows that BDPI allows agents to reliably learn

complicated tasks. This is an important result, as it is crucial in robotic experimentation to be able to quickly assess whether or not a task is learnable.

Distance sensors The intuition is that more distance sensors give the agent a finer view of its surroundings. In practice, we observed that having too many distance sensors makes the agent sensitive to noise in obstacle detection, and decreases its ability to generalize between states that are similar. More distance sensors, leading to more inputs to the actor and critics, also increases the amount of weights in these neural networks. Belkin et al. [2019] provide insights regarding this issue, by discussing the intricate relationship between the size of neural networks and the amount of training data required to train them.

Batch Size When the state-space is complicated, as in this case with high-dimensional sensor readings, larger batch sizes allow the actor and critics to learn better, with less catastrophic forgetting [French, 1999]. However, with BDPI's high sample-efficiency, the time needed to fill the experience buffer up to the batch size, during which no training happens, begins to matter. This is why a batch size of 4096 leads to a learning curve that rises later than 1024, in Figure 5.18.

Reward Function Designing a reward function is always a challenge when deploying Reinforcement Learning in the real world. In Figure 5.18, the dense reward function described in Section 5.7.3 leads to the best results. Adding uniform noise between -1 and 1 to the reward signal decreases sample-efficiency, as does providing the reward in a way comparable to Figure 5.10 (by slices of +10 or -10 about every 10 time-steps).

Advice The *going forward* action has index 2 (with the first action having index 0). The best results are obtained when encouraging the agent to go forward in every state (Adv. + LC), with advice $(0.2, 0.2, 0.4, 0.2)$. If we remove the learning correction described in Section 5.5.4 (Adv. - LC), this advice becomes detrimental to the agent. We also evaluated a smarter advice, that only encourages going forwards when there is no obstacle in front of the agent, instead of always (Adv.2 + LC). This decreases learning stability and sample-efficiency slightly. We have yet to find an explanation to this surprising result, but still present it as a promise relevant to real-world applications of Reinforcement Learning: even suboptimal advice can help an agent learn, and may well be a perfectly valid way of providing advice.

5.8 Conclusion

This chapter presents a sample-efficient model-free Reinforcement Learning system that allows a physical motorized wheelchair to learn a navigation task in about one hour. The agent observes first-person rough distance readings (vectors of 40 floating-point numbers), and has access to 4 discrete actions. Our approach requires no map, model, pre-training, demonstrations, domain-specific high-level features or domain-specific policy shaping. It also requires no absolute positioning system, such as GPS (that does not work indoors) or markers in the room. Learning a new task, in a previously-unseen location, is as simple as bringing the motorized wheelchair there, starting it up, letting it explore and providing it with rewards and punishment according to a human-defined reward function. The high sample-efficiency of our approach, along with its plug-and-play nature, will allow, in our opinion, a whole family of new applications to be tackled by robots. We believe that sample-efficient plug-and-play Reinforcement Learning allows home robots to finally be trained on-site, to produce high-quality user-specific policies. Building on our motorized wheelchair navigation task, we envision a world in which a single day will be enough to train a motorized wheelchair to perform several navigation tasks, such as going to the toilet, answering the door, going to the phone, etc. While this would largely increase the quality of life of people with reduced mobility and hand control skills, our contacts with the industry hint at further applications, such as motor-assisted hospital beds.

As pointed in the introduction of this thesis, our Reinforcement Learning system, compatible with real-world tasks, can serve as the basis for much future research. One such area is human-centric learning. As explained by Carlson and Demiris [2008], most people do not want to be passive around robots, especially when they are wheelchair users in a motorized wheelchair. Future work at the AI Lab Brussels will focus on learning more efficiently thanks to the person in the wheelchair (or any human around the robot), and giving control back to the user of the robot when desired. Other labs at the Vrije Universiteit Brussel are already doing extensive research on human-robot collaboration, that can be combined with Reinforcement Learning [El Makrini et al., 2017].

6 | Discussion

This thesis presents a sequence of contributions that increase the sample-efficiency of model-free Reinforcement Learning, starting from theoretical insights and algorithms in Chapter 2 and 3, addressing partial observability challenges motivated by the real world in Chapter 4, and finally focusing on an application on a commercially-available wheelchair in Chapter 5. With this sequence of contributions, this thesis shows that:

1. Large gains in sample-efficiency for model-free Reinforcement Learning algorithms are both desirable, possible, and lead to new application opportunities;
2. Reinforcement Learning, as presented in this thesis, is able to solve real-world problems with potentially high social impacts;
3. Solving these real-world problems requires a combination of theoretical contributions, software platforms and new algorithms, and real-world engineering.

The algorithms presented in this thesis answer our research questions on sample-efficiency as follows:

Sample-efficiency

In Chapter 3, we present Bootstrapped Dual Policy Iteration (BDPI), a

sample-efficient model-free Reinforcement Learning algorithm. We empirically demonstrate that BDPI is several times more sample-efficient than related work. The main sources of sample-efficiency of BDPI are its use of several off-policy critics, trained several times per time-step on large batches of experiences; and its actor, compatible with off-policy critics and leading to better exploration than using the critics without an actor. In future work, we plan on addressing two limitations of BDPI, them being the cost at which we obtain high sample-efficiency: BDPI is limited to discrete actions, and generalizing it to continuous actions may require the use of generative adversarial networks; and BDPI is extremely compute-intensive, requiring thousands of neural network training epochs *per time-step* to achieve the highest sample-efficiency. We discuss future work opportunities in more detail in Section 6.1.

Sample-Efficiency in POMDPs

In Chapter 4, we introduce a formalism that allows an agent to efficiently remember *discrete* pieces of information for long sequences of time-steps. Our formalism, Option-Observation Initiation Sets (OOIs), is built on Options. As such, understanding it and implementing it requires minimal efforts for someone already familiar with Options, often used for highly-challenging Reinforcement Learning problems. OOIs allow an agent to learn a task that requires memory about 3 times faster than recurrent neural networks, when some domain knowledge is provided to the agent. When no domain knowledge is available, OOIs achieve about 50% better sample-efficiency, and 50% higher final policy quality, than recurrent neural networks. Related to future work opportunities for BDPI, the main open research area regarding OOIs is its limitation to discrete pieces of memory. An agent with OOIs can remember that a light was on or off, or an integer value, but is not able to recover unobserved real-valued quantities. We discuss possible solutions to this limitation in Section 6.1.

Reinforcement Learning in the real world

Our secondary research questions consider a selection of challenges in real-world deployments of Reinforcement Learning, such as asynchronous environments, the need for backup policies, and the production of observations from cheap hardware (such as webcams). In Chapter 5, we provide solutions to these challenges, and show that a Reinforcement Learning agent is able to learn a navigation task in an office, on a commercially-available motorized wheelchair. The results we present most probably generalize to other naviga-

tion tasks, using cameras or distance sensors for sensory input. In particular, our obstacle-detection algorithm can easily be replaced by another one, fit for other vision settings, or replaced with any other kind of sensor. The backup policy we present works well in all the settings we deployed it, but can also be replaced with a higher-quality controller, or any other algorithm that modifies or blocks the actions of the agent in some states. We also believe that BDPI, extended with parallelism and an asynchronous actor, is amenable to a wider range of real-world tasks, be them physical or software-based, that share common properties: only a few time-steps executed per second, requiring high sample-efficiency; the task is expressible with discrete actions, as on the wheelchair; and there is enough compute power available at training time for BDPI. In summary, in Chapter 5, we show that Reinforcement Learning is applicable to a challenging real-world task. While we cannot evaluate our algorithms on the infinite set of real-world tasks, we believe that our result show that research towards applying our algorithms to more real-world tasks is promising.

The discussion above positions our contribution on the research line aiming at deploying Reinforcement Learning in real-world settings, and mentions some future work opportunities. The next section discusses these opportunities in greater detail.

6.1 Future Work

Even though Chapters 3 and 4 are presented in that order in this thesis, I started my PhD with OOIs, and moved to BDPI later (so, Chapter 4 presents older research than Chapter 3). In this section, I will tell the story of why I moved from OOIs to BDPI, and how it relates to future work.

The main limitation of OOIs is that they only allow *discrete pieces* of information to be remembered by the agent. The agent can go somewhere, observe a light that is either on or off (for instance), and start an option depending on that status. Then, OOIs allow the agent to choose future actions and options depending on that past discrete observation. In many real-world settings, it would be beneficial to the agent to be able to remember *continuous pieces* of information, such as how much water was in the tank when the agent observed it, or how far it is from an initial position (without being able to observe anything but its current speed).

The research direction that I think would allow OOIs to be compatible with continuous pieces of information is *parametric options* [Da Silva et al., 2012]. A parametric option is an option (and there is a finite, discrete set of options available to the agent) that, when triggered by the agent, takes a continuous-valued parameter. By allowing the agent to set the continuous parameter depending on a continuous value it observes, and by defining the OOIs as a mapping between option-parameters to option-parameters, I believe that allowing continuous pieces of information to be remembered is possible. In 2017, I therefore started preliminary work in that direction, but quickly stumbled on a big problem: back in the days, no model-free sample-efficient Reinforcement Learning existed (no BDPI, but also no Rainbow, ACKTR or Soft Actor-Critic). The Figures presented in Section 4.5 illustrate this problem very well: most plots show results on large runs of more than 30K episodes.

With parametric options, the problem was exacerbated, and it was impossible for me to know whether an agent that hadn't learned anything after a million episodes would never learn anything, or whether it just needed more time. A brilliant talk by Prof. Manuela Veloso at the Benelux Conference on Artificial Intelligence, in November 2016, also showed me the importance and benefits to be obtained from doing *research focused on the real-world*. Her CoBots are physical robots that perform useful navigation tasks in her lab at Carnegie Mellon University, namely guiding visitors around the lab. The real-world nature of this task leads to many important questions, both interesting from a research point of view, and critical for the application of AI techniques in the real world. *Using* the research, as in having the robots perform their task even if not just for an experiment, is also highly important. In the software engineering world, this is referred as *dogfooding*, or, in a more explicit way, *eating your own dog food*. The idea is that developers that use their own system become perfectly aware of all its bugs, shortcomings, limitations, or potentially-useful features. Dogfooding is now the recommended development approach in many Open Source projects, such as KDE[1] (if you work on a text editor, you have to use that text editor to edit your code).

The two reasons presented above, the need for a sample-efficient Reinforcement Learning algorithm and the appeal of research focused on the real-world, motivated me to work on BDPI. BDPI started as an experiment done in-between the submissions of the OOIs paper, and its off-policy critic started from my mis-

[1] https://kde.org, see https://blog.martin-graesslin.com/blog/2013/09/next-step-dogfooding/ for an example of dogfooding.

understanding of *off-policy* in actor-critic algorithms, as discussed in Section 2.8.1. I therefore implemented an off-policy critic, that did not work with any Policy-Gradient-based actor, and spent two years finding and fixing the problem, leading to an actor based on Conservative Policy Iteration (see Section 3.3.2).

The BDPI algorithm, presented in this thesis, addresses the sample-efficiency problem, but is limited to discrete actions. The main technical future work that I envision after this thesis is the development of a Reinforcement Learning algorithm, comparable to BDPI, but for continuous actions. In this thesis, even on the wheelchair, the agent only has access to a finite set of discrete actions, a bit as if it could only press buttons. This allows the wheelchair to navigate in an office, but more complex robots, that move arms or even fingers, need to perform actions represented by (potentially large) lists of real numbers. The main challenge with continuous actions is that the "maximum over actions" operation, a the core of both BDPI's critic (Equation 3.1) and actor update rules (Equation 3.2), becomes intractable. CACLA, the Continuous Actor-Critic Learning Automaton [Van Hasselt and Wiering, 2007], (intuitively) replaces the maximum over every possible action with the maximum over two actions: the executed one and the one the agent believed to be the best. While CACLA allows tasks with continuous actions to be learned, even in challenging industrial settings [Rodriguez Abed, 2013], its sample-efficiency is low, mainly because the agent can only update its critic when it happens to perform an action that is better than what it usually does. Not every time-step leads to critic updates. Recent advances in Generative Adversarial Networks [Goodfellow et al., 2014], that allow a neural network to learn to produce outputs that look like inputs it has been presented with, may provide a way to generate a finite amount of highly-promising continuous actions, for which Q-Values and a maximum can be computed. This is my current research direction, but training Generative Adversarial Networks is complicated to do in a stable way, and defining *what* they should learn to produce in a Reinforcement-Learning setting (actions? Q-Values? other floating-point values?) is also a research question.

Once a continuous-action version of BDPI is available, if it is possible, I strongly believe that the OOIs problem of discrete-only pieces of information would become much easier to address. BDPI with continuous actions could be used to implement parametric options, and those parametric options could potentially allow for OOIs with continuous pieces of information. The future work I envision, based on this thesis, is therefore:

1. Design a continuous-actions version of BDPI
2. Use that algorithm for a parametric-options version of OOIs

It is probably possible to design a parametric-options version of OOIs based on the Soft Actor-Critic [Haarnoja et al., 2018], but I prefer to focus my attention first on a continuous-actions version of BDPI, as, to my knowledge, no continuous-actions Reinforcement Learning algorithm has yet presented a leap in sample-efficiency as significant as the one of BDPI (see Section 3.5.4, BDPI is not 30% more sample-efficient than the other algorithms, it is 3 times faster at least).

Bibliography

Agache, M. and Oommen, B. J. (2002). Generalized pursuit learning schemes: New families of continuous and discretized learning automata. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 32(6):738–749.

Aggarwal, C. C. (2018). *Neural Networks and Deep Learning: a Textbook*. Springer.

Agrawal, S. and Goyal, N. (2012). Analysis of thompson sampling for the multi-armed bandit problem. In *Conference on Learning Theory (COLT)*.

Akkaya, I., Andrychowicz, M., Chociej, M., Litwin, M., McGrew, B., Petron, A., Paino, A., Plappert, M., Powell, G., Ribas, R., et al. (2019). Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*.

Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., and Topcu, U. (2018). Safe reinforcement learning via shielding. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., and Zaremba, W. (2017). Hindsight experience replay. *Arxiv*, abs/1707.01495.

Angeline, P. J., Saunders, G. M., and Pollack, J. B. (1994). An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1):54–65.

Anschel, O., Baram, N., and Shimkin, N. (2017). Averaged-dqn: Variance reduction and stabilization for deep reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 176–185.

Anthony, T., Tian, Z., and Barber, D. (2017). Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems (NIPS)*, pages 5366–5376.

Arjona-Medina, J. A., Gillhofer, M., Widrich, M., Unterthiner, T., and Hochreiter, S. (2018). RUDDER: return decomposition for delayed rewards. *Arxiv*, abs/1806.07857.

Atrash, A. and Pineau, J. (2009). A bayesian reinforcement learning approach for customizing human-robot interfaces. In *International conference on Intelligent user interfaces*, pages 355–360. ACM.

Auer, P. (2000). Using upper confidence bounds for online learning. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 270–279. IEEE.

Bacon, P., Harb, J., and Precup, D. (2017). The option-critic architecture. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*, pages 1726–1734.

Baker, T. M., Codina, G., and Franzen, L. H. (1997). Inductive joystick apparatus. US Patent 5,598,090.

Bakker, B. (2001). Reinforcement learning with long short-term memory. In *Advances in Neural Information Processing Systems (NIPS)*, volume 14.

Bakker, B. (2007). Reinforcement learning by backpropagation through an LSTM model/critic. *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning, ADPRL 2007*, pages 127–134.

Balakuntala, M. V., Venkatesh, V. L. N., Bindu, J. P., Voyles, R. M., and Wachs, J. (2019). Extending policy from one-shot learning through coaching. *arXiv*, abs/1905.04841.

Belkin, M., Hsu, D., Ma, S., and Mandal, S. (2019). Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854.

Bellemare, M. G., Dabney, W., and Munos, R. (2017). A distributional perspective on reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 449–458.

Bellman, R. (1957). A Markovian decision process. *Journal Of Mathematics And Mechanics*, 6:679–684.

Berkenkamp, F., Turchetta, M., Schoellig, A., and Krause, A. (2017). Safe model-based reinforcement learning with stability guarantees. In *Advances in neural information processing systems*, pages 908–918.

Berry, D. and Fristedt, B. (1985). *Bandit Problems: Sequential Allocation of Experiments*. Springer.

Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight uncertainty in neural networks. *Arxiv*, abs/1505.05424.

Böhmer, W., Guo, R., and Obermayer, K. (2016). Non-deterministic policy improvement stabilizes approximated reinforcement learning. *Arxiv*, abs/1612.07548.

Boots, B., Siddiqi, S. M., and Gordon, G. J. (2011). Closing the learning-planning loop with predictive state representations. *The International Journal of Robotics Research*, 30(7):954–966.

Bousmalis, K., Irpan, A., Wohlhart, P., Bai, Y., Kelcey, M., Kalakrishnan, M., Downs, L., Ibarz, J., Pastor, P., Konolige, K., et al. (2018). Using simulation and domain adaptation to improve efficiency of deep robotic grasping. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4243–4250. IEEE.

Bradtke, S. J. and Duff, M. O. (1995). Reinforcement learning methods for continuous-time markov decision problems. In *Advances in neural information processing systems (NIPS)*, pages 393–400.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI Gym.

Brys, T., Harutyunyan, A., Taylor, M. E., and Nowé, A. (2015). Policy transfer using reward shaping. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 181–188.

Bu, L., Babu, R., De Schutter, B., et al. (2008). A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(2):156–172.

Burda, Y., Edwards, H., Storkey, A., and Klimov, O. (2018). Exploration by random network distillation. *arXiv*, abs/1810.12894.

Carlson, T. and Demiris, Y. (2008). Human-wheelchair collaboration through prediction of intention and adaptive assistance. In *2008 IEEE International Conference on Robotics and Automation*, pages 3926–3931.

Cassandra, A., Kaelbling, L., and Littman, M. (1994). Acting optimaly in partially observable stochastic domains. In *Proceedings of the 12th AAAI National Conference on Artificial Intelligence*, volume 2, pages 1023–1028.

Cesa-Bianchi, N., Gentile, C., Lugosi, G., and Neu, G. (2017). Boltzmann exploration done right. In *Advances in Neural Information Processing Systems (NIPS)*, pages 6284–6293.

Chapelle, O. and Li, L. (2011). An empirical evaluation of thompson sampling. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2249–2257.

Chen, R. Y., Sidor, S., Abbeel, P., and Schulman, J. (2017). UCB exploration via q-ensembles. *arXiv*, abs/1706.01502.

Choi, B., Mericli, C., Biswas, J., and Veloso, M. (2013). Fast human detection for indoor mobile robots using depth images. In *Proceedings of ICRA'13, the IEEE International Conference on Robotics and Automation*.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv*, abs/1412.3555.

Da Silva, B., Konidaris, G., and Barto, A. (2012). Learning parameterized skills. *arXiv preprint arXiv:1206.6398*.

Dallaire, P., Besse, C., Ross, S., and Chaib-draa, B. (2009). Bayesian reinforcement learning in continuous pomdps with gaussian processes. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2604–2609. IEEE.

De Andrade, R., Hodel, K. N., Justo, J. F., Laganá, A. M., Santos, M. M., and Gu, Z. (2018). Analytical and experimental performance evaluations of can-fd bus. *IEEE Access*, 6:21287–21295.

De Bock, Y., Auquilla, A., Kellens, K., Vandevenne, D., Nowé, A., and Duflou, J. (2017). User-adapting system design for improved energy efficiency during the use phase of products: Case study of an occupancy-driven, self-learning thermostat. In *Sustainability Through Innovation in Product Life Cycle Design*, pages 883–898. Springer.

de Wit, C. C., Siciliano, B., and Bastin, G. (2012). *Theory of robot control*. Springer Science & Business Media.

Degris, T., White, M., and Sutton, R. S. (2012). Linear off-policy actor-critic. In *International Conference on Machine Learning, (ICML)*.

Deisenroth, M. P. and Rasmussen, C. E. (2011). PILCO: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on Machine Learning (ICML)*, pages 465–472.

Dietterich, T. G. (1999). State Abstraction in MAXQ Hierarchical Reinforcement Learning. *Advances in Neural Information Processing Systems*, 12:7.

Dietterich, T. G. (2000). Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *Journal of Artificial Intelligence Research*, 13:227–303.

Eckhardt, R., Ulam, S., and Von Neumann, J. (1987). the monte carlo method. *Los Alamos Science*, (15):131.

Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O., and Clune, J. (2019). Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995*.

Efron, B. and Tibshirani, R. J. (1994). *An introduction to the bootstrap*. CRC press.

El Makrini, I., Merckaert, K., Lefeber, D., and Vanderborght, B. (2017). Design of a collaborative architecture for human-robot assembly tasks. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1624–1629. IEEE.

Feng, G., Buşoniu, L., Guerra, T. M., and Mohammad, S. (2018). Reinforcement learning for energy optimization under human fatigue constraints of power-assisted wheelchairs. In *Annual American Control Conference (ACC)*, pages 4117–4122. IEEE.

French, R. M. (1999). Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, 3(4):128–135.

Fu, J., Kumar, A., Soh, M., and Levine, S. (2019). Diagnosing bottlenecks in Deep Q-learning algorithms. In *International Conference on Machine Learning (ICML)*.

Fujimoto, S., Hoof, H. V., and Meger, D. (2018). Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning (ICML)*, pages 1582–1591.

Geist, M. and Scherrer, B. (2014). Off-policy learning with eligibility traces: A survey. *The Journal of Machine Learning Research*, 15(1):289–333.

Glorennec, P. Y. (1994). Fuzzy q-learning and dynamical fuzzy q-learning. In *IEEE International Fuzzy Systems Conference*, pages 474–479. IEEE.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems (NIPS)*, pages 2672–2680.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Gómez Colmenarejo, S., Grefenstette, E., Ramalho, T., Agapiou, J., Badia, A. P., Moritz Hermann, K., Zwols, Y., Ostrovski, G., Cain, A., King, H., Summerfield, C., Blunsom, P., Kavukcuoglu, K., and Hassabis, D. (2016). Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476.

Griffith, S., Subramanian, K., Scholz, J., Isbell, C. L., and Thomaz, A. L. (2013). Policy shaping: Integrating human feedback with reinforcement learning. In *Advances in neural information processing systems (NIPS)*, pages 2625–2633.

Gruslys, A., Azar, M. G., Bellemare, M. G., and Munos, R. (2017). The reactor: A sample-efficient actor-critic architecture. *Arxiv*, abs/1704.04651.

Gu, S., Holly, E., Lillicrap, T., and Levine, S. (2017a). Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3389–3396. IEEE.

Gu, S., Lillicrap, T., Ghahramani, Z., Turner, R. E., and Levine, S. (2017b). Q-prop: Sample-efficient policy gradient with an off-policy critic. In *International Conference on Learning Representations, (ICLR)*.

Gu, S., Lillicrap, T., Turner, R. E., Ghahramani, Z., Schölkopf, B., and Levine, S. (2017c). Interpolated policy gradient: Merging on-policy and off-policy gradient estimation for deep reinforcement learning. In *Advances in Neural Information Processing Systems (NIPS)*, pages 3849–3858.

Gui, L., Zhang, K., Wang, Y., Liang, X., Moura, J. M., and Veloso, M. (2018). Teaching robots to predict human motion. In *Proceedings of IROS'18, the IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Haarnoja, T., Tang, H., Abbeel, P., and Levine, S. (2017). Reinforcement learning with deep energy-based policies. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pages 1352–1361.

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

Harb, J., Bacon, P.-L., Klissarov, M., and Precup, D. (2018). When waiting is not an option: Learning options with a deliberation cost. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Harrison, B., Ehsan, U., and Riedl, M. O. (2018). Guiding reinforcement learning exploration using natural language. In *International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1956–1958.

Harutyunyan, A., Devlin, S., Vrancx, P., and Nowé, A. (2015). Expressing arbitrary reward functions as potential-based advice. In *AAAI Conference on Artificial Intelligence*.

Harutyunyan, A., Vrancx, P., Bacon, P.-L., Precup, D., and Nowe, A. (2018). Learning with options that terminate off-policy. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

He, R., Brunskill, E., and Roy, N. (2011). Efficient planning under uncertainty with macro-actions. *Journal of Artificial Intelligence Research (JAIR)*, 40:523–570.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G., and Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning. *Arxiv*, abs/1710.02298.

Hester, T. (2013). *TEXPLORE: Temporal Difference Reinforcement Learning for Robots and Time-Constrained Domains*. PhD thesis.

Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Sendonaris, A., Dulac-Arnold, G., Osband, I., Agapiou, J., Leibo, J. Z., and Gruslys, A. (2017). Learning from Demonstrations for Real World Reinforcement Learning. Technical report.

Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. (2018). Stable baselines. https://github.com/hill-a/stable-baselines.

Hochreiter, S. and Schmidhuber, J. J. (1997). Long short-term memory. *Neural Computation*, 9(8):1–32.

Howse, J. (2013). *OpenCV computer vision with python*. Packt Publishing Ltd.

Jonsson, A. and Barto, A. G. (2000). Automated State Abstraction for Options using the U-Tree Algorithm. *Advances in Neural Information Processing Systems*, pages 1054–1060.

Józefowicz, R., Zaremba, W., and Sutskever, I. (2015). An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pages 2342–2350.

Kaelbling, L. P., Littman, M. L., and Cassandra ', A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134.

Kakade, S. and Langford, J. (2002). Approximately optimal approximate reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 267–274.

Kartoun, U., Stern, H., and Edan, Y. (2010). A human-robot collaborative reinforcement learning algorithm. *Journal of Intelligent & Robotic Systems*, 60(2):217–239.

Karttunen, J., Kanervisto, A., Hautamäki, V., and Kyrki, V. (2019). From video game to real robot: The transfer between action spaces. *arXiv*, abs/1905.00741.

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Knox, W. B. and Stone, P. (2010). Combining manual feedback with subsequent MDP reward signals for reinforcement learning. In *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*.

Kochenderfer, M. J. and Wheeler, T. A. (2019). *Algorithms for Optimization*. MIT Press.

Konda, V. R. and Borkar, V. S. (1999). Actor-Critic-Type Learning Algorithms for Markov Decision Processes. *SIAM Journal on Control and Optimization*, 38(1):94–123.

Konidaris, G. and Barto, A. G. (2007). Building portable options: Skill transfer in reinforcement learning. In *International Joint Conference on Artificial Intelligence (IJCAI)*, volume 7, pages 895–900.

Konidaris, G. and Barto, A. G. (2009). Skill discovery in continuous reinforcement learning domains using skill chaining. In *Advances in neural information processing systems (NIPS)*, pages 1015–1023.

Konidaris, G., Kuindersma, S., Grupen, R., and Barto, a. (2012). Robot learning from demonstration by constructing skill trees. *The International Journal of Robotics Research*, 31(3):360–375.

Kulkarni, T. D., Narasimhan, K., Saeedi, A., and Tenenbaum, J. (2016). Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in neural information processing systems (NIPS)*, pages 3675–3683.

Lagoudakis, M. G. and Parr, R. (2001). Model-Free Least Squares Policy Iteration. In *Advances in Neural Information Processing Systems (NIPS)*, volume 2, pages 1547–1554.

Lazic, N., Boutilier, C., Lu, T., Wong, E., Roy, B., Ryu, M., and Imwalle, G. (2018). Data center cooling using model-predictive control. In *Advances in Neural Information Processing Systems (NIPS)*, pages 3814–3823.

Lee, T.-J., Yi, D.-H., Cho, D.-I., et al. (2016). A monocular vision sensor-based obstacle detection algorithm for autonomous robots. *Sensors*, 16(3):311.

Lenser, S. and Veloso, M. (2003). Visual sonar: Fast obstacle avoidance using monocular vision. In *International Conference on Intelligent Robots and Systems (IROS)*, volume 1, pages 886–891. IEEE.

Levine, S., Finn, C., Darrell, T., and Abbeel, P. (2016). End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17:39:1–39:40.

Libin, P. (2020). *Guiding the mitigation of epidemics with reinforcement learning*. PhD thesis.

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

Lim, Z. W., Hsu, D., and Lee, W. S. (2011). Monte carlo value iteration with macro-actions. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1287–1295.

Lin, L.-J. and Mitchell, T. M. (1992). *Memory approaches to reinforcement learning in non-Markovian domains*. Carnegie-Mellon University. Department of Computer Science.

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings*, pages 157–163. Elsevier.

Littman, M. L., Sutton, R. S., and Singh, S. (2001). Predictive Representations of State. In *Advances in Neural Information Processing Systems (NIPS)*, volume 14, pages 1555–1561.

Loy, J. (2019). *Neural Network Projects with Python*. Packt Publishing.

MacGlashan, J., Ho, M. K., Loftin, R., Peng, B., Wang, G., Roberts, D. L., Taylor, M. E., and Littman, M. L. (2017). Interactive learning from policy-dependent human feedback. In *International Conference on Machine Learning-Volume (ICML)*, pages 2285–2294.

Mahadevan, S. (1996). Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine learning*, 22(1-3):159–195.

Meuleau, N., Peshkin, L., Kim, K.-e., and Kaelbling, L. P. (1999). Learning Finite-State Controllers for partially observable environments. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 427–436.

Miesterfeld, F. O., McCambridge, J. M., Fassnacht, R. E., and Nasiadka, J. M. (1988). Method for buffered serial peripheral interface (spi) in a serial data bus. US Patent 4,742,349.

Mihaylov, M., Razo-Zapata, I., Radulescu, R., and Nowé, A. (2016). Boosting the renewable energy economy with nrgcoin. In *ICT for Sustainability 2016*. Atlantis Press.

Mikolov, T., Joulin, A., Chopra, S., Mathieu, M., and Ranzato, M. (2014). Learning longer memory in recurrent neural networks. *CoRR*, abs/1412.7753.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, volume 48, pages 1928–1937.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. a., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Montemerlo, M., Pineau, J., Roy, N., Thrun, S., and Verma, V. (2002). Experiences with a mobile robotic guide for the elderly. In *AAAI Conference on Innovative Applications of AI (IAAI)*, volume 2002, pages 587–592.

Moore, A. W. and Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine learning*, 13(1):103–130.

Munos, R., Stepleton, T., Harutyunyan, A., and Bellemare, M. G. (2016). Safe and efficient off-policy reinforcement learning. In *Neural Information Processing Systems (NIPS)*.

Murray, J. D. and VanRyper, W. (1996). *Encyclopedia of graphics file formats, 2nd edition*. O'Reilly.

Narendra, K. S. and Thathachar, M. A. L. (1989). *Learning automata - an introduction*. Prentice Hall.

Ng, A. Y., Harada, D., and Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *International Conference on Machine Learning (ICML)*, volume 99, pages 278–287.

Nikolov, N., Kirschner, J., Berkenkamp, F., and Andreas, K. (2019). Information-directed exploration for deep reinforcement learning. In *International Conference on Learning Representations (ICLR)*. in preparation.

Nilsson, N. J. (2009). *The quest for artificial intelligence*. Cambridge University Press.

O'Donoghue, B., Munos, R., Kavukcuoglu, K., and Mnih, V. (2017). PGQ: Combining policy gradient and Q-learning. In *International Conference on Learning Representations (ICLR)*, page 15.

Omidshafiei, S., Agha-Mohammadi, A., Amato, C., Liu, S., How, J. P., and Vian, J. (2017). Decentralized control of multi-robot partially observable markov decision processes using belief space macro-actions. *International Journal of Robotics Research*, 36(2):231–258.

Osband, I., Aslanides, J., and Cassirer, A. (2018). Randomized prior functions for deep reinforcement learning. *arXiv*, abs/1806.03335.

Osband, I., Blundell, C., Pritzel, A., and Van Roy, B. (2016). Deep exploration via bootstrapped DQN. In *Advances in Neural Information Processing Systems (NIPS)*.

Osborne, A. (1982). *An Introduction to Microcomputers: The beginner's book*. Adam Osborne and Associates.

Pardo, F., Tavakoli, A., Levdik, V., and Kormushev, P. (2018). Time limits in reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 4045–4054.

Parisotto, E., Ba, J., and Salakhutdinov, R. (2016). Actor-mimic: Deep multitask and transfer reinforcement learning. In *International Conference on Learning Representations (ICLR)*.

Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pages 2778–2787.

Perkins, T. J. and Pendrith, M. D. (2002). On the existence of fixed points for q-learning and sarsa in partially observable domains. In *International Conference on Machine Learning (ICML)*, pages 490–497.

Peshkin, L., Meuleau, N., and Kaelbling, L. (1999). Learning policies with external memory. In *Proceedings of the 16th International Conference on Machine Learning (ICML)*, pages 307–314.

Pinsker, M. S. (1960). Information and information stability of random variables and processes.

Pirotta, M., Restelli, M., Pecorino, A., and Calandriello, D. (2013). Safe policy iteration. In *Proceedings of the 30th International Conference on Machine Learning, (ICML)*, pages 307–315.

Plappert, M., Houthooft, R., Dhariwal, P., Sidor, S., Chen, R. Y., Chen, X., Asfour, T., Abbeel, P., and Andrychowicz, M. (2017). Parameter space noise for exploration. *Arxiv*, abs/1706.01905.

Plisnier, H., Steckelmacher, D., Roijers, D. M., and Nowé, A. (2019a). The actor-advisor: Policy gradient with off-policy advice. *arXiv*, abs/1902.02556.

Plisnier, H., Steckelmacher, D., Roijers, D. M., and Nowé, A. (2019b). Transfer reinforcement learning across environment dynamics with multiple advisors. *CEUR-WS Proceedins*, 2491.

Precup, D. (2000a). Eligibility traces for off-policy policy evaluation. Technical report.

Precup, D. (2000b). *Temporal Abstraction in Reinforcement Learning*. PhD thesis, University of Massachusetts.

Puiutta, E. and Veith, E. (2020). Explainable reinforcement learning: A survey. *arXiv preprint arXiv:2005.06247*.

Rashid, T. (2016). *Make Your Own Neural Network*. Amazon Digital Services LLC.

Recht, B., Re, C., Wright, S., and Niu, F. (2011). Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems (NIPS)*, pages 693–701.

Rethink Robotics (2016). Baxter research robot: Technical specification datasheet & hardware architecture overview.

Riedmiller, M. (2005). Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer.

Rodriguez Abed, A. (2013). *Continuous Action Reinforcement Learning Automata, an RL technique for controlling production machines.* PhD thesis.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Rudary, M. R. and Singh, S. P. (2004). A nonlinear predictive state representation. In *Advances in neural information processing systems (NIPS)*, pages 855–862.

Rusu, A. A., Colmenarejo, S. G., Gulcehre, C., Desjardins, G., Kirkpatrick, J., Pascanu, R., Mnih, V., Kavukcuoglu, K., and Hadsell, R. (2015). Policy distillation. *arXiv*, abs/1511.06295.

Sason, I. (2015). On reverse pinsker inequalities. *Arxiv*, abs/1503.07118.

Saunders, W., Sastry, G., Stuhlmueller, A., and Evans, O. (2018). Trial without error: Towards safe reinforcement learning via human intervention. In *17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 2067–2069.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized Experience Replay. In *International Conference on Learning Representations (ICLR)*.

Scherrer, B. (2014). Approximate policy iteration schemes: A comparison. In *Proceedings of the 31th International Conference on Machine Learning (ICML)*, pages 1314–1322.

Schulman, J., Levine, S., Abbeel, P., Jordan, M. I., and Moritz, P. (2015). Trust region policy optimization. In *International Conference on Machine Learning (ICML)*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *Arxiv*, abs/1707.06347.

Sexton, R. S., Dorsey, R. E., and Johnson, J. D. (1998). Toward global optimization of neural networks: a comparison of the genetic algorithm and backpropagation. *Decision Support Systems*, 22(2):171–185.

Shapiro, J. and Narendra, K. S. (1969). Use of stochastic automata for parameter self-optimization with multimodal performance criteria. *IEEE Trans. Systems Science and Cybernetics*, 5(4):352–360.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic Policy Gradient Algorithms. In *International Conference on Machine Learning (ICML)*, pages 387–395.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *Nature*, 550(7676):354.

Sobel, I. and Feldman, G. (1968). A 3x3 isotropic gradient operator for image processing. *Talk at the Stanford Artificial Project*, page 2.

Socha, K. and Blum, C. (2007). An ant colony optimization algorithm for continuous optimization: application to feed-forward neural network training. *Neural Computing and Applications*, 16(3):235–247.

Sridharan, M., Wyatt, J., and Dearden, R. (2010). Planning to see: A hierarchical approach to planning visual actions on a robot using POMDPs. *Artificial Intelligence*, 174:704–725.

Steckelmacher, D., Plisnier, H., Roijers, D. M., and Nowé, A. (2019). Sample-efficient model-free reinforcement learning with off-policy critics. *arXiv*, abs/1903.04193.

Steckelmacher, D., Roijers, D. M., Harutyunyan, A., Vrancx, P., and Nowé, A. (2017). Reinforcement learning in POMDPs with memoryless options and option-observation initiation sets. In *AAAI Conference on Artificial Intelligence*. AAAI Press.

Steckelmacher, D. and Vrancx, P. (2015). An empirical comparison of neural architectures for reinforcement learning in partially observable environments. In *Benelux Conference on Artificial Intelligence (BNAIC)*.

Still, S. and Precup, D. (2012). An information-theoretic approach to curiosity-driven reinforcement learning. *Theory in Biosciences*, 131(3):139–148.

Stolle, M. and Precup, D. (2002). Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pages 212–223. Springer.

Strens, M. (2000). A bayesian framework for reinforcement learning. In *Internation Conference on Machine Learning (ICML)*, pages 943–950.

Sun, W., Gordon, G. J., Boots, B., and Bagnell, J. A. (2018). Dual policy iteration. *Arxiv*, abs/1805.10755.

Sutton, R., McAllester, D., Singh, S., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems (NIPS)*, volume 13.

Sutton, R., Precup, D., and Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112:181–211.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*.

Sutton, R. S., Precup, D., and Singh, S. (1998). Intra-option learning about temporally abstract actions. *Proceedings of the Fifteenth International Conference on Machine Learning (ICML 1998)*, pages 556–564.

Tedrake, R., Zhang, T. W., and Seung, H. S. (2005). Learning to walk in 20 minutes. In *Fourteenth Yale Workshop on Adaptive and Learning Systems*, volume 95585, pages 1939–1412.

Terryn, S., Brancart, J., Lefeber, D., Van Assche, G., and Vanderborght, B. (2017). Self-healing soft pneumatic robots. *Science Robotics*, 2(9).

Tessler, C., Givony, S., Zahavy, T., Mankowitz, D. J., and Mannor, S. (2016). A Deep Hierarchical Approach to Lifelong Learning in Minecraft. In *13th European Workshop on Reinforcement Learning (EWRL)*.

Thathachar, M. A. and Sastry, P. S. (1986). Estimator algorithms for learning automata. In *Platinum Jubilee Conference on Systems and Signal Processing*.

Theocharous, G. (2002). *Hierarchical learning and planning in partially observable Markov decision processes*. PhD thesis, Michigan State University.

Thomas, P. S., Theocharous, G., and Ghavamzadeh, M. (2015). High confidence policy improvement. In *International Conference on Machine Learning (ICML)*, pages 2380–2388.

Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294.

Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., and Abbeel, P. (2017). Domain randomization for transferring deep neural networks from simulation to the real world. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30.

Tsitsiklis, J. N. (1994). Asynchronous stochastic approximation and q-learning. *Machine learning*, 16(3):185–202.

Turchetta, M., Berkenkamp, F., and Krause, A. (2016). Safe exploration in finite markov decision processes with gaussian processes. In *Advances in Neural Information Processing Systems*, pages 4312–4320.

Ulrich, I. and Nourbakhsh, I. (2000). Appearance-based obstacle detection with monocular color vision. In *Innovative Applications of Artificial Intelligence (AAAI)*, pages 866–871.

van Hasselt, H. (2010). Double Q-Learning. In *Neural Information Processing Systems (NIPS)*, page 9.

Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double Q-Learning. In *AAAI Conference on Artificial Intelligence*.

Van Hasselt, H. and Wiering, M. A. (2007). Reinforcement learning in continuous action spaces. *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 272–279.

Veloso, M. M. (2018). The increasingly fascinating opportunity for human-robot interaction: The cobot mobile service robots. *ACM Transactions on Human-Robot Interaction*.

Verstraeten, T., Nowe, A., Keller, J., Guo, Y., Sheng, S., and Helsen, J. (2019). Fleetwide data-enabled reliability improvement of wind turbines. *Renewable and Sustainable Energy Reviews*, 109:428–437.

Wagner, P. (2011). A reinterpretation of the policy oscillation phenomenon in approximate policy iteration. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2573–2581.

Wald, G. (1964). The receptors of human color vision. *Science*, 145(3636):1007–1016.

Wang, R., Veloso, M., and Seshan, S. (2014). O-snap: Optimal snapping of odometry trajectories for route identification. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. (2016a). Sample Efficient Actor-Critic with Experience Replay. Technical report.

Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., and de Freitas, N. (2016b). Dueling Network Architectures for Deep Reinforcement Learning. Technical report.

Watkins, C. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292.

Wedel, A., Franke, U., Klappstein, J., Brox, T., and Cremers, D. (2006). Realtime depth estimation and obstacle detection from monocular video. In *Joint Pattern Recognition Symposium*, pages 475–484. Springer.

Werbos, P. J. et al. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

Wiering, M. and Schmidhuber, J. (1997). HQ-Learning. *Adaptive Behavior*, 6(2):219–246.

Wijmans, E., Kadian, A., Morcos, A., Lee, S., Essa, I., Parikh, D., Savva, M., and Batra, D. (2019). Dd-ppo: Learning near-perfect pointgoal navigators from 2.5 billion frames. *arXiv preprint*, abs/1911.00357.

Williams, R. J. (1992). Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8(3):229–256.

Wu, Y., Mansimov, E., Grosse, R. B., Liao, S., and Ba, J. (2017). Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In *Advances in neural information processing systems (NIPS)*, pages 5279–5288.

Xie, Z., Clary, P., Dao, J., Morais, P., Hurst, J., and van de Panne, M. (2019). Iterative reinforcement learning based design of dynamic locomotion skills for cassie. *arXiv*, abs/1903.09537.

Yamaguchi, K., Kato, T., and Ninomiya, Y. (2006). Moving obstacle detection using monocular vision. In *IEEE Intelligent Vehicles Symposium*, pages 288–293. IEEE.

Yao, H., Szepesvári, C., Sutton, R., Modayil, J., and Bhatnagar, S. (2014). Universal Option Models. *Proceedings of the 28th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 1–9.

Youcef-Toumi, K. and Ito, O. (1990). A time delay controller for systems with unknown dynamics.

Zaremba, W. and Sutskever, I. (2015). Reinforcement Learning Neural Turing Machines. *CoRR*, abs/1505.00521.

Zhang, S. and Sutton, R. S. (2017). A deeper look at experience replay. *Arxiv pre-print*, abs/1712.01275.

Index