



École Doctorale Informatique, Télécommunications et Électronique

Institut des Systèmes Intelligents et de Robotique

Integrating Motion Planning into Reinforcement Learning to solve hard exploration problems

THÈSE DE DOCTORAT

présentée par : Guillaume MATHERON

soutenue le : 18 novembre 2020

discipline : Intelligence Artificielle

Thèse dirigée par M. Olivier SIGAUD, *Professeur*, ISIR

et co-dirigée par M. Nicolas PERRIN, *Chargé de recherche*, ISIR

Jury :

M. Olivier BUFFET, Rapporteur

M. Emmanuel RACHELSON, Rapporteur

M. Jean-Paul LAUMOND, Examineur

Mme Véronique PERDEREAU, Examinatrice

M. Nicolas PERRIN, Encadrant

M. Olivier SIGAUD, Directeur de thèse

Institut des Systèmes Intelligents et de Robotique

Pyramide - T55/65, 4 Place Jussieu, 75005 PARIS - tél. 01 44 27 51 41

<http://www.isir.upmc.fr/>

Contact : guillaume_thesis@matheron.eu

Abstract

The past decade has featured an explosion in the use of reinforcement learning for continuous control tasks. In this context, many issues are solved a few years after they arise, but two main hurdles stand out from the pack. The first hurdle is the difficulty of general robotics problems. Many recent works demonstrate impressive results in simulated environments, but very few feature some important characteristics of real-world problems: continuous control and long sequences of tasks to accomplish. Reinforcement learning algorithms on continuous state problems with continuous actions are a recent addition to the tool-belt that is reinforcement learning, and these algorithms tend to need extensive hyperparameter tuning to reliably converge. The second hurdle is actually reaching the target. As most continuous optimization process, reinforcement learning relies directly or indirectly on gradients to improve policies and incrementally solve the problem at hand. However, this requires a starting strategy that yields a useful gradient, which is seldom the case for complex tasks in mobile robots. Many strategies have been implemented to circumvent this issue, but we believe that the next breakthrough lies in the field of robotics, and more specifically motion planning, which has been dealing with large state-spaces with sparse rewards for decades.

Motion planning is able to solve robotics problems much quicker than any reinforcement learning algorithm by efficiently searching for a viable trajectory. Indeed, while the main object of interest in the field of Reinforcement Learning is the behavior of an agent, Motion Planning is concerned with the geometry and properties of the state-space, and uses a different set of primitives to achieve more efficient exploration. Some of these primitives require a model of the system and are not studied in this work, others such as reset-anywhere are only available in simulated environments.

On the other hand, Motion Planning approaches do not benefit from the same generalization properties as the policies produced by reinforcement learning.

In this thesis, we study the ways in which techniques inspired from motion planning can speed up the solving of hard exploration problems for reinforcement learning without sacrificing the advantages of model-free learning and generalization. We identify a deadlock that can occur when applying reinforcement learning to seemingly-trivial sparse-reward problems, and contribute an exploration algorithm inspired by motion planning but specifically designed for reinforcement learning environments, as well as a framework to use the collected data to train a reinforcement learning algorithm in previously-intractable scenarios.

Acknowledgments

I would first like to thank the École Normale Supérieure de Paris, Sorbonne Université, the École Doctorale Informatique, Télécommunications et Électronique and the Institut des Systèmes Intelligents et de Robotique for the classes and lectures that I have benefited from during these three years of PhD thesis, and their support.

I am grateful to the professors of ENS Ulm, and especially Jean-Paul Laumond who gave me the extra boost in interest towards robotics and motion planning, that led me to a fantastic internship and ultimately to performing a PhD in this domain.

I would like to thank all my colleagues at NASA/JPL during my internship and in particular Olivier Toupet, who was an inspiration and gave me the confidence to dare mighty things.

I want to thank all the great professors and speakers that I was lucky to meet on my way, who communicated their passion for computer science and robotics, as well as Christian Lorenzi for encouraging me with managing the fab lab of ENS.

I want to warmly thank my family and step-family who supported me during these years, and my wife Marguerite Matheron for her support and proofreading of this manuscript.

My deepest gratitude goes to my mentors Olivier Sigaud and Nicolas Perrin, who stimulated my curiosity and my interest through passionate discussions about Reinforcement Learning techniques, definitions and much

wider topics. These discussions were an essential part of my journey, but as important was the advice given about deadlines, strategy, writing articles and conferences. Striking a balance between exploration and exploitation is a hard but integral part of my research subject, and striking a balance between exploring, experimenting and writing has been a hard but integral part of this PhD.

I want to thank all the researchers who gave me their opinion and reviews, especially my reviewers Emmanuel Rachelson and Olivier Buffet, as well as Stéphane Doncieux.

This work was partially supported by the French National Research Agency (ANR), Project ANR-18-CE33-0005 HUSKI

Contents

List of Figures	11
Introduction	14
Motivation	14
Methodology	16
Summary of contributions	17
1 Background	19
1.1 Background on Motion Planning	19
1.1.1 Configurations, constraints and holonomy	20
1.1.2 Implications of non-holonomic constraints	22
1.1.3 Common assumptions in Motion Planning	23
1.2 Background on Reinforcement Learning	23
1.2.1 Markov Decision Processes	23
1.2.2 Matching Reinforcement Learning and Motion Planning concepts	24
1.2.3 Policies and agents	25
1.2.4 Q-Learning	26
1.2.5 Deep Deterministic Policy Gradient	27
1.3 Reset-anywhere	30
2 Related work	31
2.1 Taxonomy of Reinforcement Learning algorithms	32
2.2 Exploration mechanisms for behavioral learning	35
2.3 Attempts to use Motion Planning techniques with Reinforce- ment Learning	36

3	Using motion planning in a reinforcement learning environment	38
3.1	Adapting Rapidly-exploring Random Tree to RL Algorithms .	39
3.1.1	Rapidly-exploring Random Tree	39
3.1.2	RRT in RL environments	42
3.1.3	Limitations of RRT due to sampling	44
3.2	Expansive Spaces Tree	45
3.3	Ex	46
3.3.1	Pitfalls of density as a measure for novelty	46
3.3.2	The Ex algorithm	48
3.3.3	Related work	50
3.4	Experiments using a feature space	52
3.4.1	The curse of dimensionality	52
3.4.2	Feature spaces	53
3.4.3	Benchmark on Ant-Maze	53
3.5	Conclusion	54
4	Exploiting exploration data in Reinforcement Learning through the experience replay buffer	56
4.1	Experimental setup	57
4.2	Behavior of DDPG, RRT-NH and Ex on mazes	57
4.3	Motivation	59
4.4	Methods	60
4.5	Results	60
4.6	Analysis	62
4.7	Thin walls and the limits of “sets of transitions”	63
4.8	Conclusion	64
5	Exploiting exploration data as a training curriculum: backtracking	66
5.1	Introduction	66
5.2	Related work	67
5.3	Backtracking algorithm	68
5.4	Experimental Setup	69
5.5	Choice of discount factor γ	70
5.6	Results on 2D mazes and analysis	71
6	The problem of deterministic policy gradients in deterministic environments with sparse rewards	74
6.1	Related work	75
6.2	A new failure mode	76

6.2.1	The 1D-Toy environment	76
6.2.2	Residual failure to converge using different noise processes.	76
6.3	Correlation between finding the reward early and finding the optimal policy.	78
6.3.1	Spontaneous actor drift	79
6.3.2	Explaining the deadlock situation for DDPG on 1D-Toy	80
6.3.3	Formal proof of the existence of a deadlock in 1D-Toy	82
6.4	Generalization to all deterministic Policy Gradient algorithms in deterministic environments with sparse rewards	83
6.4.1	Proof of convergence of the critic to Q^π	84
6.4.2	Proof that Q^π is piecewise-constant	85
6.4.3	Consequences of the convergence cycle	88
6.5	Impact of function approximation	89
6.6	Potential solutions	90
6.6.1	Avoiding sparse rewards	90
6.6.2	Replacing the policy-based critic update	90
6.6.3	Replacing the deterministic policy gradient update	93
6.7	Experiments on larger benchmarks	93
6.8	Conclusion	95
7	Going one step further: backtracking with skill chaining	96
7.1	Related work	97
7.2	Methods	98
7.2.1	Skill chaining algorithm	98
7.2.2	Adapted backplay algorithm	99
7.2.3	Reward shaping	101
7.2.4	Need for Resetting to Unseen States	103
7.3	Results on 2D mazes	105
7.4	Analysis of results	105
7.5	Influence of hyperparameters	108
7.6	Conclusion	109
	Conclusion	111
	Bibliography	115
	Acronyms	128
	Glossary	130

A	Appendix: tooling and other contributions	133
A.1	Reinforcement Learning toolchain	133
A.2	Gazebo simulator	134
A.3	RRT pitfalls	134
A.4	Analysis of 1D-Toy	134
A.4.1	Probability of success for an unbiased random walk with sink	135
A.4.2	Probability of success for unbiased p-greedy random walk with sink	137
A.4.3	Sanity check	138

List of Figures

1	Relations between the problems we faced and our contributions	18
1.1	Klann linkage [Klann, 2005] in four different positions.	20
2.1	Properties of Reinforcement Learning (RL) algorithms	32
2.2	Equations of RL algorithms	33
3.1	Behavior of Rapidly-exploring Random Tree (RRT). For each iteration, first s_{rand} is chosen randomly, then its nearest neighbor in the exploration tree s_{near} is found, and finally an action a is chosen either using a heuristic (in the case of RRT), or randomly (in the case of Rapidly-exploring Random Tree Non-Holonomic (RRT-NH)). The resulting state s' is added to the tree.	40
3.2	Test of RRT when the reachable space is different from the sampling space	43
3.3	Test of RRT-NH when the reachable space is different from the sampling space	43
3.4	Test of Ex when the reachable space is different from the sampling space	44
3.5	Behavior of EST	45
3.6	Paradox occurring when using a disc kernel to measure the density of an area	47
3.7	Operation of RRT, EST and Ex	48
3.8	AntMaze environment	54
3.9	Performance of Ex and RRT-NH on the AntMaze environment	55

4.1	Actor and critic of DDPG in a maze with thin walls	58
4.2	Exploration pattern of RRT-NH and Ex in a maze	58
4.3	Exploration pattern of RRT-NH and Ex in a maze when continuing beyond the first reward found	59
4.4	Actor and critic of offline DDPG preloaded with exploration data in a maze with thin walls	61
4.5	Actor and critic of DDPG preloaded with exploration data in a 2x2 maze with thin walls	61
4.6	Actor and critic of DDPG in a 3x3 maze with thin walls . . .	62
4.7	Mini-batch created from exploration data on a 2D maze	63
5.1	Variants of the <i>Plan, Backplay</i> algorithm.	69
5.2	Success rate of DDPG and PB, depending on γ	72
5.3	Success rate of PB on 2D mazes, when compared to DDPG. Error bars are computed using Wilson error intervals. From left to right, $N = 183, 169, 185, 168, 101, 101$	72
5.4	Example run of vanilla DDPG on a 2x2 maze for 100k steps. The sparse reward, which was placed in the top-left hand corner, was not found.	73
6.1	Presentation of the 1D-TOY environment	76
6.2	Performance of DDPG on 1D-TOY	77
6.3	Detection of rewards in mini-batches during training	78
6.4	Spontaneous drift of the actor and critic in DDPG	79
6.5	Visualization of the DDPG critic in a deadlock	80
6.6	Deadlock cycle in 1D-TOY	81
6.7	Generalized deadlock cycle	83
6.8	Illustration of the proof that Q^π has flat gradient	86
6.9	Local extrema and function approximation	89
6.10	Performance of DDPG-argmax and SAC on 1D-TOY	91
6.11	Performance of DDPG and DDPG-argmax on a sparse HALFCHEETAH-v2	94
7.1	Performance of DDPG and PB on 2x2 mazes with and without reward shaping	101
7.2	Problematic behavior when combining backtracking and skill chaining	104
7.3	Results of DDPG, TD3, PBCS, PB on maze environments . .	106
7.4	Enlarged view of the results of PBCS on mazes environments .	107
A.1	PDF of state after n steps of the unbiased random walk with sink	137

A.2	Cumulative success rate after n steps of the unbiased random walk with sink	137
A.3	Cumulative success rate after n steps of the p-greedy unbiased random walk with sink	139
A.4	PDF of state after 21 steps of unbiased random walk with sink	140
A.5	Cumulative success rate after n steps of the 0.1-greedy unbiased random walk with sink	140

Introduction

Artificial Intelligence is widely regarded as the next frontier, feeding some form of self-reinforcing human fantasy that machine thinking will reach a point when it can learn and improve itself exponentially until it reaches a singularity beyond which it completely escapes human comprehension. In this regard, Artificial Intelligence has been described as “The last invention that humanity will ever need to make” [Bostrom, 2015].

Advances with neural networks in the past decade have fueled this idea that humans are already unable to understand the reasoning behind decisions made by computers, and that super-intelligence is only one step away. However, as with flying cars and moon bases, the horizon of super-intelligent machines seems as elusive as an actual horizon, always further as we advance.

Motivation

We dream of a strong artificial intelligence that is able to drive the behavior of robots, but by testing leading edge artificial intelligence algorithms on extremely simple problems we show that fundamental problems are still holding back these approaches. In this PhD thesis, we scratch the surface of one of the many roadblocks that lie ahead, specifically in the domain of learning motion tasks for virtual robots.

Suppose you just bought a small mobile robot like Nao, and you want to teach it to go to the coffee machine, make a cup of hot coffee, and bring it back to you. You don’t know where the coffee machine is, you don’t know the torque profile of the actuators of Nao, and you don’t know the field of view of its cameras.

The promise of Reinforcement Learning (RL) is that by rewarding good behavior and punishing bad behavior, a computer program can progressively learn how to maximize the reward it receives. This directly parallels how animals and humans learn to behave and move: toddlers are hurt when falling over, and are congratulated by their parents when making their first steps. Can we transpose this to robots?

The answer is yes, and many works in the past decade [Lillicrap et al., 2015; Mnih et al., 2013; Schulman et al., 2015; Fujimoto et al., 2018b] have proposed algorithms and methods for training smart controllers that are able to drive the actions of a virtual or physical robot, or similar problems such as controlling a character in computer games. However, these methods usually fall short in tasks where progress towards the goal is difficult to evaluate [Achiam et al., 2019]. Collectively, these problems are known as *exploration problems*, contrasting with *optimization problems* in which the policy can be incrementally improved with useful feedback from the environment or an external trainer.

In the context of RL where the goal is expressed as a reward function, these hard exploration problems are said to have *sparse rewards* in contrast with *dense rewards* that guide the learning process towards the final goal. Some RL problems have *deceptive rewards*, which are dense rewards that feature several local maxima, however these can usually be converted to sparse reward problems by applying a threshold to the reward signal. Some other problems have rewards that are hard to define because the user only has an implicit understanding of the task objective. These are not covered in this work, but are studied in the work of Leike et al. [2018].

This inability of RL algorithms to solve exploration problems is especially frustrating because exploration can seem to be added to RL algorithms as an afterthought that theoretically guarantees convergence with no consideration given to the actual performance of training in exploration problems. Other classes of algorithms such as Motion Planning (MP) are often able to solve these problems orders of magnitude faster, and are still predominant in the domain of real-world robotics. However, these approaches often need an expert-designed model of the robot and the environment, and output a single open-loop trajectory instead of a motion controller. Therefore closing the control loop is a major challenge, to which RL algorithms seem suited because of their generalization properties.

Research question. In this PhD thesis, we attempt to bridge the gap between these two worlds, the ultimate goal being achieving the exploration capabilities of MP while training efficient controllers, all this with minimal

expert input and modeling. Therefore our main research question is *how to integrate components of MP in RL algorithms in order to improve the automated exploration of large deterministic environments with sparse rewards.*

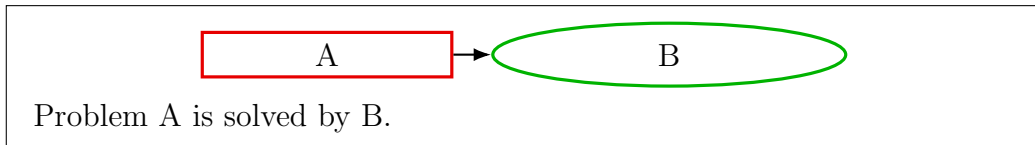
Methodology

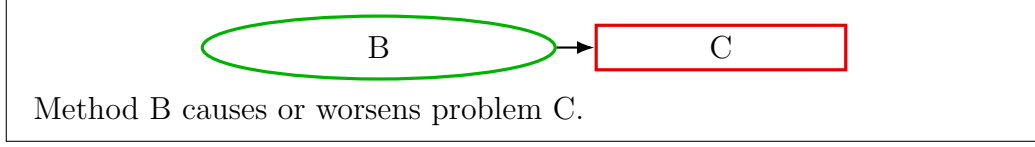
All of the experiments in this work are performed on either virtual robots which are designed and simulated to emulate the behavior of their real-world physical counterparts, or simulated environments that have been crafted to compare a specific property of several algorithms. Lower-dimension environments such as cliff walk [Sutton and Barto, 2018b] are often used to demonstrate fundamental properties of RL algorithms, and testing in these environments occasionally reveals fundamental flaws [Matheron et al., 2019].

Another class of simulated environments we use are 2D mazes, because they are hard exploration problems and reward shaping behaves very poorly in such environments, creating many local optima. Our results in Section 7.3 show that state-of-the-art algorithms such as DDPG [Lillicrap et al., 2015] and Twin Delayed DDPG (TD3) [Fujimoto et al., 2018b] fail to solve even very simple mazes, validating the pertinence of these environments as a benchmark for exploration.

In some parts of this thesis, we use the *reset-anywhere* primitive which is only available in a simulated environment, therefore, some of these methods are not directly applicable to physical robots. However, training RL agents on these virtual robots can be used as a demonstration tool, as visualization, for entertainment or in order to transfer the learned policies to physical robots, for direct use or further training.

In this thesis, we have identified several issues facing the field of RL, some of which are well-known and others are contributed. Some are inherent to the problem statement, and others are introduced or worsened by solutions we brought to solve other problems. In order to clarify the effect of each of the methods presented in this document, the main arguments of this PhD thesis are summarized in Fig. 1. In the body of the text, this formatting indicates that the paragraph directly references Fig. 1:





Therefore, in Fig. 1, problems that have no incoming edge are inherent to the field of RL.

Summary of contributions

All research is based on existing work, and builds on the shoulders of giants. However, we introduce some elements that are, to the best of our knowledge, original or published by ourselves during the course of this thesis.

- In Section 3.3, we contribute a novel exploration algorithm called Ex, which is based on MP concepts but is well-suited to the exploration of RL environments, and outperforms other MP algorithms on these environments [Matheron et al., 2020].
- In Section 4.7, we demonstrate that using mini-batches of transitions from an exploration tree to pre-seed a RL replay buffer is futile in 2D mazes with thin walls, and therefore that no RL algorithm in continuous state-spaces can truly be off-policy [Matheron et al., 2019].
- In Section 5.3, we propose a variant of the Backplay algorithm [Resnick et al., 2018] in which the backtracking process is controlled by the actual performance of the underlying RL algorithm. We then combine this backtracking algorithm with Ex using the framework of Go-Explore [Ecoffet et al., 2019] to produce an algorithm called Plan, Backplay (PB) [Matheron et al., 2020].
- In Chapter 6, we demonstrate that Deep Deterministic Policy Gradient (DDPG) can fail in even trivial environments with unlimited time, and prove that this deadlock is possible for any Policy Gradient (PG) algorithm in a deterministic environment with sparse rewards [Matheron et al., 2019].
- In Chapter 7, we contribute a novel algorithm called Plan, Backplay, Chain Skills (PBCS) which builds on PB and efficiently solves 2D maze environments [Matheron et al., 2020].

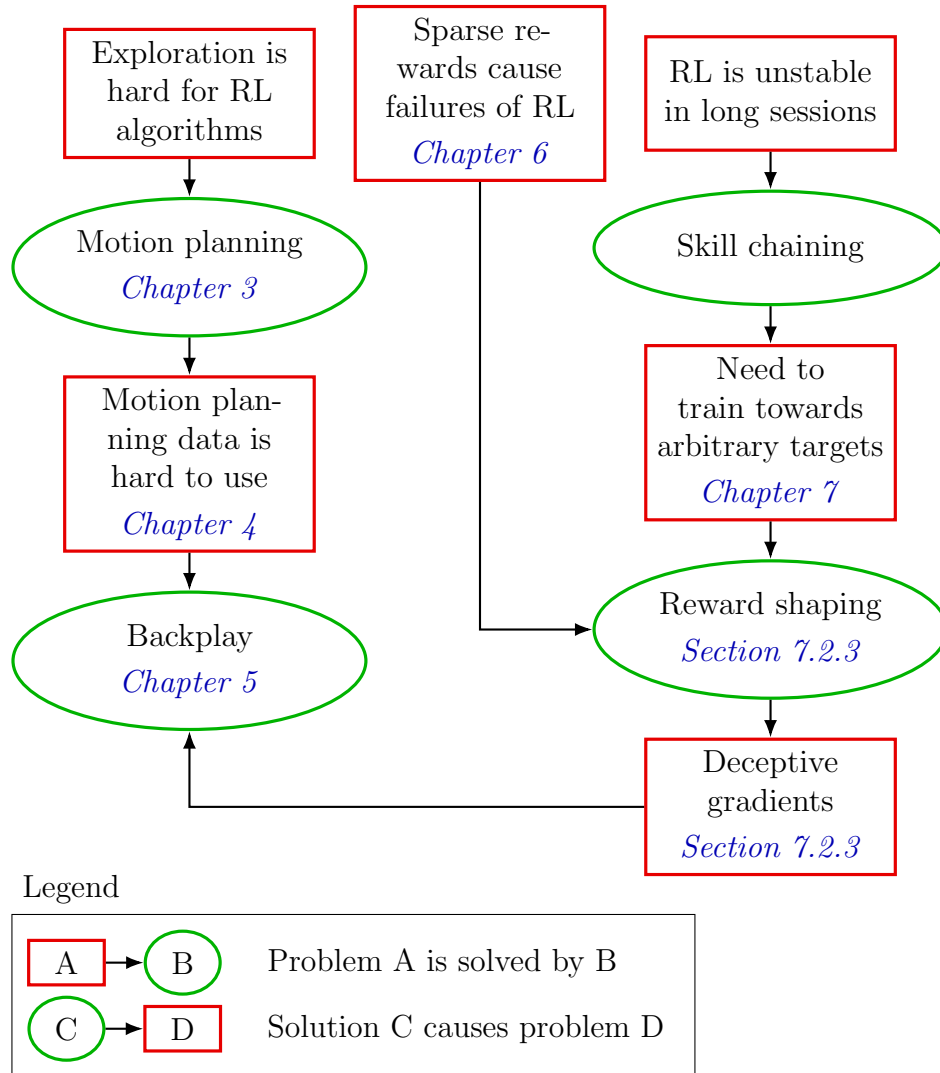


Figure 1: Relations between the problems we faced and our contributions

This PhD thesis builds on sixty years of research in Machine Learning (ML) [Samuel, 1959], and thirty years in Motion Planning (MP) [Latombe, 1991]. This chapter presents prerequisites that should be sufficient for any researcher in Computer Science to understand the following chapters, but are common knowledge to scientists working in Reinforcement Learning (RL).

Section 1.1 defines basic robotics concepts such as configuration spaces, implicit parameterizations, reachability and holonomy. Section 1.2 defines the concepts of Markov Decision Process (MDP) and how it relates to MP, and presents Q-Learning and Deep Deterministic Policy Gradient (DDPG), two algorithms on which we rely in the following chapters. Section 1.3 studies *reset anywhere*, an additional primitive which is usually available in simulated environments such as Go [Silver et al., 2016], but is not as commonly used in RL continuous control frameworks.

1.1 Background on Motion Planning

Motion Planning is usually studied in the context of real robots, in which an embedded planning component sends commands to motion controllers that drive actuators interacting with the physical world. Periodically, the state of the robot is estimated through the output of sensors which can be of many types (cameras, switches, angle sensors, ...).

A discrete-time approximation of this behavior is often used, especially when working in a simulated environment in which the state of the robot is computed as a sequence of values.

The following sections define useful concepts in the study of robotics, and are largely inspired by the work of Lynch and Park [2017].

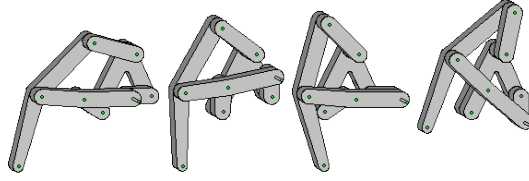


Figure 1.1: Klann linkage [Klann, 2005] in four different positions.

1.1.1 Configurations, constraints and holonomy

Robots are commonly described as a graph of rigid *links*, chained together with actuated or unactuated *joints* that constrain the motion of the robot. Additional *constraints* may be applied by obstacles, ground contacts or forces.

Degrees of Freedom (DoF). The configuration of a rigid link can be defined by a unique coordinate system, which features three degrees of freedom for position and three degrees of freedom for orientation, for a total of 6 DoF. Therefore, the configuration of a robot with N links can be described using $6N$ parameters.

However, each *independent* constraint on the configuration of the robot introduces a redundancy in this full configuration. More precisely, each independent constraint removes one DoF from the robot. Joints can be of different types. For instance the pivot joint introduces five independent constraints: three positional constraints and two orientation constraints. Therefore a robot with N links and J pivot joints has $6N - 5J$ DoF. Note that this only holds true if all constraints are independent, which can be hard to determine.

For instance, a mobile robot with two links joined by one pivot joint has $6 \times 2 - 5 \times 1 = 7$ DoF. If one of these two links is fixed to the ground (representing six constraints), then the resulting manipulator arm has only $6 \times 2 - 5 \times 1 - 6 = 1$ DoF. However, applying the same reasoning to the Klann linkage depicted in Fig. 1.1 would result in a negative number of DoF because some non-independent constraints are accounted for twice. In this section when counting constraints we assume they are all independent.

Explicit vs. implicit parameterizations. The configuration of a robot can be expressed using the union of all 6 configuration parameters of each link. However, constraints on the configuration can also be used to reduce the dimension of the configuration space. For instance, the configuration of the manipulator arm mentioned earlier can be described using only the angle of the pivot joint. More generally, the configuration of an acyclic mobile robot

using only J pivot joints can be parametrized using the 6 parameters of one of its links, and the angles of each joint, resulting in a configuration space of dimension $6 + J$. Such a robot has $J + 1$ links and J pivot joints, therefore it has $6(J + 1) - 5J = J + 1$ DoF.

parameterizations that use a minimum number of coordinates are called *explicit parameterizations*, while parameterizations that use redundant coordinates are called *implicit*.

In the case of an implicit parameterization using k parameters and c independent constraints, the set of admissible configurations can be seen as a $(k - c)$ -dimensional surface embedded in a k -dimension space of all possible parameterizations (called the *configuration space* or *C-space*).

Constraints on velocity. Constraints on the configuration of the robot are called *holonomic*. They are only present when the parameterization is implicit, and can be differentiated such that one constraint on the configuration translates into one constraint on the derivative of the configuration.

Holonomic constraints are also called *integrable* because when they are specified as a constraint on the derivative of the configuration \dot{q} , they can be integrated back to a constraint on the robot configuration q .

On the other hand, constraints that are specified in terms of \dot{q} and that cannot be integrated to a constraint on q are called non-holonomic. These constraints do not decrease the dimension of the configuration space but still restrict the ways in which the robot can move.

Example of the Reeds-Shepp Car. Robots that do not have non-holonomic constraints are called *holonomic*, while systems that have at least one non-holonomic constraint are called *non-holonomic*.

An example of non-holonomic system is the parallel parking example, also known as the *Reeds-Shepp Car* [Reeds and Shepp, 1990]. In this system, a single link is subject to two positional constraints (stay on the ground), one angular constraint (stay flat on the ground), and one non-holonomic constraint (do not slide sideways).

This means that the system has three DoF which can be represented using an explicit parameterization using the planar coordinates of a point in the car, and a steering angle. The holonomic constraint does not further reduce the dimension of the configuration space (all positions and steering angles are still valid configurations) but it reduces the possible velocities, therefore some C-space trajectories are infeasible (namely the ones that require the car to slide sideways). However, these trajectories can be approximated to any precision by small back-and-forth maneuvers (this property is called

small-time local controllability). On the other hand, a similar car that can only go forward is not small-time local controllable [Dubins, 1957].

1.1.2 Implications of non-holonomic constraints

Holonomy is an important property for our research problem because it simplifies many MP approaches that are based on geometry. In a holonomic problems, all C-space trajectories that follow the C-space constraints are admissible. This has several implications:

- The shape of the set of admissible configurations can be studied independently from the velocities. For instance, reachability problems are equivalent to geometrical connectivity problems.
- All trajectories are independent of speed, and can be specified as trajectories in C-space.

Furthermore, many robotics problems can be specified as explicit parameterizations. For instance, humanoid robots are often a tree of links joined by pivot, and can be described using the 6 parameters of a link, plus the angles of all joints. A holonomic problem with an explicit parameterization has the following additional properties (assuming no obstacles):

- The shortest trajectory between two configurations in C-space is a straight line.
- All configurations are reachable from any other configuration.

This means that proximity in C-space is a good indicator of the similarity of two states, and the time it takes to reach one from the other. This property plays a major role in the exploration of state spaces because the C-space distance between two states can be used as a proxy for novelty (Section 3.2).

Holonomic problems (especially when an explicit parameterization is available) often have a straightforward geometric solution based on the topology of their configuration space, while non-holonomic problems have to rely on sampling approaches or more complex geometric primitives [Sussmann and Tang, 1991].

Real-world robotics problem are rarely holonomic, because the dynamics of the robot itself (the inertia of the links) and gravity introduce non-holonomic constraints.

1.1.3 Common assumptions in Motion Planning

MP approaches can be versatile, and depending on the level of available expert knowledge, different algorithms can be used. For instance, in Chapter 3, we show how two variants of the Rapidly-exploring Random Tree (RRT) algorithm can be used depending on the available primitives.

Here are common assumptions that MP can take advantage of:

1. An invertible model of the robot geometry is known, and can be used to command actuators in order to achieve a target velocity for each link.
2. In the presence of non-holonomic constraints, a heuristic is available and gives an estimation of the required actuation in order to achieve a target velocity for each link.
3. We can sample points in a reasonable approximation of the reachable configuration space.
4. Given two points in C-space, it is possible to determine whether the patch the two is either free or blocked by an obstacle (this sometimes takes the form of a *local planner*)

1.2 Background on Reinforcement Learning

This section presents the usual framework for RL, and how robotics problems (that are more commonly studied within the framework of MP) can be integrated in an RL framework.

1.2.1 Markov Decision Processes

The most common mathematical model for RL is a MDP, which can be defined as a tuple (S, A, P_a, R_a) in which:

- S is the set of possible *environment states*,
- A is the set of possible actions,
- with $s, s' \in S$, $P_a(s, s')$ is the probability that the next environment state will be s' , given that its present state is s and the action that is chosen by the controller is currently a .

Deterministic environments are such that for each state s and action a , only one next-state s' is possible, in other words $P_a(s, s') = 1$. In this case, the transition distribution is commonly written as a function $\text{step}(s, a) = s'$.

- Given that the current state of the environment is s , the next state of the environment is s' , and the chosen action was a , $R_a(s, s')$ is the reward used as feedback by a learning algorithm on the next time step. In deterministic environments, the reward function is also deterministic and the parameter s' can be omitted. The reward can then be written as $R(s, a)$.

1.2.2 Matching Reinforcement Learning and Motion Planning concepts

In this section, we cast a new light on the formulation of MDPs by applying this framework to MP and more generally robotics. This is useful when interfacing a simulated robot with a RL algorithm.

- **State space S .** Robots are commonly described as a tree of rigid *links*, chained together with *joints*. The environment state then includes the angles of each robot joint, but also their angular velocity. The state of mobile robots includes the position and orientation of their base, as well as their velocity and angular velocity. If the environment contains mobile objects in addition to the robot itself, then these also need to be included in the environment state. More precisely, the state of a robot described by a configuration q is (q, \dot{q}) . In robotics, S is usually continuous but in a general MDP, it can be either finite, countable or continuous.
- **Action space A .** This can be a low-level hardware parameter such as motor voltages, but simulation models sometimes omit the controller and assume that the joints are torque-controlled, and therefore an action a is a vector of torque values in $\text{N} \cdot \text{m}^{-1}$. In robotics, the most natural definition of A is usually continuous but in a general MDP, it can be either finite, countable or continuous. Some approaches are based on using only a subset of the possible continuous actions, and represent A as a finite set although the underlying environment could use continuous actions.
- **Transition function.** The transition function is usually implemented as a deterministic function, but can include stochastic noise to account for uncertainty, measurement or odometry errors.
- **Reward.** The concept of RL reward has no direct equivalent in MP, although some MP approaches use *artificial potential fields* [Fakoor et al., 2015]. However, when the target state is known, a sparse reward

can be added to this target (and to nearby states, since finding a point in a continuous state space is a zero-probability event).

1.2.3 Policies and agents

A policy is a function π that maps a state s to a distribution over possible actions A . Deterministic policies are simply functions $S \rightarrow A$.

Functions of an infinite set cannot generally be represented using a finite amount of information. Therefore, when the state space S is continuous, policies need to be parametric: the output of the policy π depends on its input s but also a set of continuous parameters collectively represented as a vector θ . This output distribution is then written $\pi_\psi(s)$, and this definition is easily transposed to deterministic policies.

The interaction between policy and environment is the heart of an MDP, and is depicted in Algorithm 1. All RL techniques studied in this thesis are based on this cycle.

A learning *agent* is an algorithm that stores an internal state which forms its *experience*. When a *transition* is observed between state s and s' after performing action a , the agent collects this experience in the form of a *transition* (s, a, r, s') , where $r = R_a(s, s')$. The agent continuously collects more experience through interactions with the environment using a policy π (this process is called *rollouts*).

Algorithm 1: Basic experience collection on a MDP using a fixed policy

Input : $s_0 \in S$ the initial environment state
A MDP (S, A, P, R)
A policy π

```

1  $s \leftarrow s_0$ 
2 while True do
3   | Sample an action  $a$  according to policy  $\pi(s)$ 
4   | Sample a new state  $s'$  in  $S$  according to distribution  $P_a(s, \cdot)$ 
5   | Obtain the reward  $R_a(s, s')$ 
6   | Update the current state  $s \leftarrow s'$ 
7 end
```

1.2.4 Q-Learning

From value iteration to Q-Learning

Q-Learning can be derived from the value iteration algorithm that is common for finding optimal policies in finite discrete MDPs [Sutton and Barto, 2018a].

The following equation is the update rule for value iteration:

$$V_{n+1}(s) = \max_a \sum_{s'} P_a(s, s') (R_a(s, s') + \gamma V_n(s')).$$

Applying this equation to a deterministic environment yields:

$$V_{n+1}(s) = \max_a (R(s, a) + \gamma V_n(\text{step}(s, a))).$$

We then introduce the state-action value function $Q(s, a)$, describing the expected reward from state s , given that the first action is a :

$$V_{n+1}(s) = \max_a \left(\underbrace{R(s, a) + \gamma V_n(\text{step}(s, a))}_{Q_n(s, a)} \right). \quad (1.1)$$

By performing a change of index, and setting $s' = \text{step}(s, a)$, $Q_{n+1}(s, a)$ can be written as:

$$Q_{n+1}(s, a) = R(s, a) + \gamma V_{n+1}(s').$$

By substitution using Eq. (1.1), we get the state-action value update equation:

$$Q_{n+1}(s, a) = R(s, a) + \gamma \max_{a'} Q_n(s', a')$$

The Q-Learning algorithm stores a table of state-action values and updates it according to the previous equation. As it typically targets non-deterministic environments, it relies on a soft update to guarantee convergence, introducing an update rate hyperparameter α :

$$Q_{n+1}(s, a) = (1 - \alpha)Q_n(s, a) + \alpha \left(R(s, a) + \gamma \max_{a'} Q_n(s', a') \right)$$

This equation can be generalized to continuous states by using a function approximator for Q , which is the technique used in the Deep Q-Network (DQN) algorithm. In Section 1.2.5 we introduce a way to generalize the Q-Learning equation to environments with continuous action spaces, resulting in the DDPG algorithm.

1.2.5 Deep Deterministic Policy Gradient

DDPG is one of the seminal RL algorithm for learning continuous behaviors in continuous state spaces, while exploiting data collected from experience. Some other algorithms fitting this description have emerged since, but DDPG remains the most straightforward transcription of Q-Learning equations in the realm of continuous state and action spaces, as described in Section 1.2.5.

The DDPG algorithm [Lillicrap et al., 2015] is a deep RL algorithm based on the Deterministic Policy Gradient theorem [Silver et al., 2014]. It borrows the use of a replay buffer and target networks from DQN [Mnih et al., 2015].

Derivation of DDPG from the base equation of Q-Learning

We remind the state-action value iteration equation, which we derived from state value iteration in Section 1.2.4:

$$Q_{n+1}(s, a) = R(s, a) + \gamma \max_{a'} Q_n(s', a').$$

The maximization operation can be performed easily when the action-state is finite, but in order to generalize to environments with continuous actions, we factor out the maximization problem by introducing an actor function π representing the policy:

$$\begin{cases} Q_{n+1}(s, a) = R(s, a) + \gamma Q_n(s', \pi(s')) \\ \pi(s') = \operatorname{argmax}_{a'} Q_n(s', a') \end{cases}.$$

Since Q and π are functions of continuous inputs, they both need to be stored in a parametric form Q_θ and π_ψ , which are commonly implemented as neural networks:

$$\begin{cases} \text{Regress } Q_\theta(s, a) \text{ towards } R(s, a) + \gamma Q_\theta(s', \pi(s')) \\ \text{Maximize } Q_\theta(s, \pi_\psi(s)) \text{ w.r.t. } \psi \end{cases}.$$

This formulation defines DDPG as a *bootstrapping algorithm*: the value estimate $Q(s, a)$ is computed using $Q(s', \pi(s'))$ where Q and π are themselves approximations.

This optimization problem can be solved using a gradient descent algorithm. Since Q is used to evaluate the performance of π , it is commonly called the *critic*.

The actor and critic are updated using stochastic gradient descent on two losses L_ψ and L_θ . These losses are computed from mini-batches of samples $(s_i, a_i, r_i, t_i, s_{i+1})$, where each sample corresponds to a transition

$s_i \rightarrow s_{i+1}$ resulting from performing action a_i in state s_i , with subsequent reward $r_i = r(s_i, a_i)$. In some environments, some transitions are marked as terminal, therefore we incorporate a termination index $t_i = t(s_i, a_i)$ to the critic update. Equations (1.2) and (1.3) define L_ψ and L_θ :

$$\begin{cases} \forall i, y_i = r_i + \gamma(1 - t_i)Q_{\theta'}(s_{i+1}, \pi_{\psi'}(s_{i+1})) \\ L_\theta = \frac{1}{2} \sum_i \left[Q_\theta(s_i, a_i) - y_i \right]^2. \end{cases} \quad (1.2)$$

$$L_\psi = - \sum_i Q_\theta(s_i, \pi_\psi(s_i)). \quad (1.3)$$

As DDPG uses a replay buffer, the mini-batch samples are acquired using a behavior policy β which may be different from the actor π . Usually, β is defined as π plus a noise distribution, which in the case of DDPG is either a Gaussian function or the more sophisticated Ornstein-Uhlenbeck noise.

Two target networks $\pi_{\psi'}$ and $Q_{\theta'}$ are also used in DDPG. Their parameters ψ' and θ' respectively track ψ and θ using exponential smoothing and help the convergence properties of the algorithm.

Training for the loss given in Eq. (1.2) yields the critic parameter update in Eq. (1.4), with α_c the learning rate. The multiplication by 2 comes from the differentiation of the square.

$$\theta \leftarrow \theta - \alpha_c \sum_i \frac{\partial Q_\theta(s_i, a_i)^T}{\partial \theta} (Q_\theta(s_i, a_i) - y_i). \quad (1.4)$$

Note that in the second equation, the optimized variable is ψ , therefore the gradient has to be computed using the chain rule. In the context of RL, this is called the policy gradient trick. Training for the loss given in Eq. (1.3) yields the actor parameter update in Eq. (1.5), with α_a the learning rate:

$$\psi \leftarrow \psi + \alpha_a \sum_i \frac{\partial \pi_\psi(s_i)^T}{\partial \psi} \nabla_a Q_\theta(s_i, a)|_{a=\pi_\psi(s_i)}. \quad (1.5)$$

Characterization as an intermediate between two extreme regimes

In this section, we characterize the behavior of DDPG as an intermediate between two extremes, that we respectively call the *critic-centric view*, where the actor is updated faster, resulting in an algorithm close to Q-Learning, and the *actor-centric view*, where the critic is updated faster, resulting in a behavior more similar to Policy Gradient.

Actor update: the critic-centric view. The Q-Learning algorithm [Watkins, 1989] and its continuous state counterpart DQN [Mnih et al., 2013] rely on the computation of a policy which is greedy with respect to the current critic at every time step, as they simply take the maximum of the Q-values over a set of discrete actions. In continuous action settings, this amounts to an intractable optimization problem if the action space is large and non-trivial.

We get a simplified vision of DDPG by considering an extreme regime where the actor updates are both fast enough and good enough so that $\forall s, \pi(s) \approx \operatorname{argmax}_a Q(s, a)$. We call this the *critic-centric* vision of DDPG, since the actor updates are assumed to be ideal and the only remaining training is performed on the critic.

In this regime, by replacing $\pi(s)$ with $\operatorname{argmax}_a Q(s, a)$ in Eq. (1.2), we get $y_i = r_i + \gamma(1 - t_i) \max_a Q(s_i, a)$, which corresponds to the update of the critic in Q-Learning and DQN. A key property of this regime is that, since the update is based on a maximum over actions, the resulting algorithm can learn from any transition even if it was collected using a policy that differs from the current actor.

Such algorithms are called *off-policy*, the reverse being *on-policy* algorithms that are only able to learn from data generated using their current policy, and therefore need to discard collected data each time the policy changes. Between these two extremes, a spectrum exists: some algorithms are more or less robust to the difference between the sampling policy and their current policy.

Chapter 6 explores the *off-policyness* of DDPG, but since the critic-centric vision of DDPG yields the equations of Q-Learning which is fully off-policy, we can infer that most of the off-policiness property of DDPG comes from keeping it close to this regime. The notion of *off-policiness* is studied in more detail in Chapter 4.

Critic update: the actor-centric view. Symmetrically to the previous case, if the critic is updated well and faster than the actor, it tends to represent the critic of the current policy, Q^π . Furthermore, if the actor tends to change slowly enough, critic updates can be both fast and good enough so that it reaches the fixed point of the Bellman equation, that is $\forall(s, a, r, t, s'), Q(s, a) = r + \gamma(1 - t)Q(s', \pi(s'))$.

In this case, the optimization performed in Eq. (1.3) mostly consists in updating the actor so that it exploits the corresponding critic by applying the deterministic policy gradient on the actor. This gives rise to a *actor-centric* vision of DDPG.

1.3 Reset-anywhere

Our work makes heavy use of the ability to reset an environment to any state. The use of this primitive is relatively uncommon in robotics RL because it is not always readily available, especially in real-world robotics problems. However, it can be invaluable to speed up exploration of large state spaces. It was used in the context of Atari games by [Hosu and Rebedea, 2016], proposed in [Schulman et al., 2015] as VINE, and gained popularity with [Salimans and Chen, 2018].

We can also note that although rare in the field of robotics-inspired RL, this primitive is used both in the wider field of RL [Silver et al., 2016] and in MP when exploring configurations.

Two different types of reset-anywhere exist: resetting to an already-visited state, or resetting to an arbitrary state, that is known only by its configuration q and generalized velocity \dot{q} .

Resetting to an already-visited state This primitive is available in most simulation software, and when unavailable (or in real-world robots) it can be simulated by resetting the environment to its initial state and re-playing a sequence of actions that led to the state in a previous visit. This requires the environment to be deterministic, and more importantly it greatly decreases the performance advantage of resetting.

Resetting to a new state This primitive cannot be simulated when not available in the simulation software in the general case, because the sequence of actions that leads to the desired state is unknown. This primitive is used in our work to ensure the robustness of policies to changes in the initial state of the robot, therefore this primitive can be approximated to finding an arbitrary state s_2 close to a known state s . This can be done by either adding noise to a recorded sequence of actions that leads to s , or by simulating the environment backwards from s (if available in the simulation software) with random actions.

Related work

In each chapter of this work, we present methods and experiments that further our research question: *how to integrate components of Motion Planning (MP) in Reinforcement Learning (RL) algorithms*. Each chapter contains contributions and novel analyses and, when applicable, a *Related Work* section that details other efforts in tackling the same problem, or other works that use similar methods.

In this chapter, we provide a more general overview of works that are a bit further from our main thread, or otherwise do not fit in a specific chapter.

This thesis spans mostly on the domains of model-free RL and MP. Although we believe it is an under-exploited area, the intersection between the two fields is not empty, and many approaches tackle the problem of exploration in model-free RL.

Most MP algorithms are not easily applicable to our problem statement, and understanding them only gives marginal insight into our problem, therefore we do not attempt to complete a general survey of MP techniques. However, the seminal MP algorithm for exploration, Rapidly-exploring Random Tree (RRT), is studied in Chapter 3.

In this chapter, we begin by presenting a taxonomy of model-free RL algorithms, inspired by the work of OpenAI [Achiam and Morales, 2018] and augmented with considerations that are relevant to this thesis. Then, we take a step back and present a wider survey of behavioral learning techniques that are designed to explore state spaces with sparse rewards. Finally, we explore previous attempts at combining MP and RL which are too dissimilar to ours to be mentioned in the next chapter, but deserve a mention to paint a complete picture of the research landscape.

Algorithm	State-space	Action-space	Off-policy	Actor type
PG	continuous	continuous	no	stochastic
PPO	continuous	both	no	stochastic
TRPO	continuous	both	no	stochastic
SARSA	finite	finite	no	deterministic
Q-Learning	finite	finite	yes	deterministic
DQN	continuous	finite	yes	deterministic
A2C	continuous	continuous	no	stochastic
DDPG	continuous	continuous	yes ¹	deterministic
TD3	continuous	continuous	yes ¹	deterministic
SAC	continuous	both	yes ¹	stochastic

¹ This is disputed, see Chapter 6.

Figure 2.1: Properties of RL algorithms

2.1 Taxonomy of Reinforcement Learning algorithms

RL is concerned with training an agent to maximize the reward it obtains when interacting with the environment. In this thesis, we are only concerned with model-free RL algorithms, which means that they only rely on trial-and-error and do not use any other information about the environment [Sutton and Barto, 2018b].

These algorithms can be classified by their properties, as summarized in Fig. 2.1. Figure 2.2 summarizes the update equations for each algorithm.

Policy optimization approaches: PG, PPO, TRPO. Policy optimization refers, as its name suggests, to incremental improvements of a learned policy with respect to its discounted reward. These processes generally aim to provide monotonic improvements of the reward, and use various techniques to compute the best change to the policy parameters.

A key property of these algorithms is that they can only learn from trajectories generated using their current policy. Therefore these algorithms are said to be *on-policy*, the reverse being *off-policy* algorithms that are able to learn from any trajectory.

On-policy algorithms are usually less sample-efficient than their off-policy counterparts because they have to regenerate new trajectories for each policy update. Policy optimization algorithms include Policy Gradient (PG) algorithms (such as REINFORCE [Williams, 1992]), Proximal Policy Opti-

Algorithm	Update rules
SARSA	$\{Q(s_t, a_t) \stackrel{\alpha}{\leftarrow} r + \gamma Q(s_{t+1}, a_{t+1})\}$
Q-Learning	$\{Q(s_t, a_t) \stackrel{\alpha}{\leftarrow} r + \gamma \max_a Q(s_{t+1}, a)\}$
DQN	$\{Q(s_t, a_t) \stackrel{\alpha}{\rightsquigarrow} r + \gamma \max_a Q(s_{t+1}, a)\}$
A2C	$\begin{cases} V(s_t) \stackrel{\alpha_v}{\leftarrow} r + \gamma V(s_{t+1}) \\ P^\pi(a_t s_t) \text{ is updated according to } r + \gamma V(s_{t+1}) - V(s_t) \end{cases}$
DDPG	$\begin{cases} Q(s_t, a_t) \stackrel{\alpha_Q}{\rightsquigarrow} r + \gamma Q'(s_{t+1}, \pi'(s_{t+1})) \\ \pi(s_t) \text{ is updated using the gradient of } Q(s_t, \pi(s_t)) \\ \theta_{Q'} \stackrel{\alpha_{Q'}}{\rightsquigarrow} \theta_Q \\ \theta_{\pi'} \stackrel{\alpha_{\pi'}}{\rightsquigarrow} \theta_\pi \end{cases}$
TD3	$\begin{cases} Q_{\{1,2\}}(s_t, a_t) \stackrel{\alpha_Q}{\rightsquigarrow} r + \gamma \min_{i=1,2} Q'_i(s_{t+1}, \pi'(s_{t+1})) + \text{noise} \\ \pi(s_t) \text{ is updated using the gradient of } Q_1(s_t, \pi(s_t)) \\ \theta_{Q'_1} \stackrel{\alpha_{Q'}}{\rightsquigarrow} \theta_{Q_1} \\ \theta_{Q'_2} \stackrel{\alpha_{Q'}}{\rightsquigarrow} \theta_{Q_2} \\ \theta_{\pi'} \stackrel{\alpha_{\pi'}}{\rightsquigarrow} \theta_\pi \end{cases}$

These equations describe the update that is applied to the state of the learning agent when a transition between states s_t and s_{t+1} is observed after performing action a . In the case of SARSA, s_{t+1} is the action performed at the following time step.

The notation $f(x) \stackrel{\alpha}{\leftarrow} g(x)$ is used to designate a soft update, that is $f(x) \leftarrow \alpha f(x) + (1 - \alpha)g(x)$. When f is a discrete function, then $f(x) \leftarrow y$ is an update to a table, similar to dynamic programming. However, when f is a continuous function then the update is performed using a regression with loss $\frac{1}{2}(f(x) - y)^2$ and learning rate α , which we write $f(x) \stackrel{\alpha}{\rightsquigarrow} y$. These equations present single updates, but several updates are commonly bundled together using mini-batches.

Figure 2.2: Equations of RL algorithms.

mization (PPO) [Schulman et al., 2017] and Trust Region Policy Optimization (TRPO) [Schulman et al., 2015].

Tabular algorithms: SARSA and Q-Learning. When the state and action spaces are finite, the state-action value function can be stored exactly, and a dynamic programming update can be performed to update its value. Rollouts can be performed to collect new experience without the need to maintain a separate actor, because the action to take from a certain state s can be computed by finding $\operatorname{argmax}_a Q(s, a)$.

The two approaches that fit in this category are SARSA and Q-Learning. They both use slightly different update rules detailed in Fig. 2.2. Q-Learning is fully off-policy in the sense that if all the transitions composing an optimal trajectory are sampled regularly, this algorithms will converge to an optimal policy. SARSA on the other hand is not off-policy because its collected experience $(s_t, a_t, r, s_{t+1}, a_{t+1})$ includes a_{t+1} and the update rule requires that a_{t+1} converges towards an optimal action to take from state s_{t+1} .

Continuous-state and finite-action algorithm: DQN. Deep Q-Network (DQN) can be seen as a generalization of Q-Learning to continuous state spaces. The tabular critic Q is replaced with a function estimator (such as a Multi-Layer Perceptron), but as with Q-Learning, the finite action space allows finding the best action a for a given state s and critic Q by computing $\operatorname{argmax}_a Q(s, a)$.

The hard update rule of Q-Learning is also replaced in DQN with a soft update, computed as a regression which slowly changes the parameters of the function approximator to tend towards the computed target. This allows the function approximator to generalize slowly and reliably.

Continuous actor-critic approaches: A2C, DDPG, TD3 and SAC. Actor-critic approaches have been first introduced for discretized problems [Barto et al., 1983], however in this area they have been supplanted by Q-Learning [Watkins, 1989] and other approaches which are simpler and allow for convergence proofs. Today, the main benefit of actor-critic methods is their ability to function in an environment with continuous states and actions. Contrarily to DQN, this requires the use of a separate policy estimator because the continuous actions do not allow the computation of an explicit $\operatorname{argmax}_a Q(s, a)$.

The critic Q or V is used to estimate the state-action or state value function, while the policy estimator (called the *actor*) is used to generate

rollouts, and is updated in order to estimate the best policy given the past experience.

In the case of Advantage Actor Critic (A2C), the value function is estimated using the same update rule as SARSA. Since only the value function is estimated, there is no need for a maximization operation. The change applied to the value function $r + \gamma V(s_{t+1}) - V(s)$ is called the Time-Difference Error or TD-Error δ . It measures the difference between the expected value of a state and its measured value in one transition. This TD-Error can be used to update the actor: good surprises result in this action being chosen more often, while bad surprises result in this action being chosen less often. Since A2C uses a stochastic actor, this update is performed using the negative log likelihood [Mnih et al., 2016].

Deep Deterministic Policy Gradient (DDPG) is studied in more detail in Section 1.2.5, and generalizes DQN to continuous-control environments by introducing a deterministic actor, which is updated using the gradient of $Q(s_t, \pi_\psi(s_t))$ with respect to the policy parameters ψ . It also uses target networks, which allows for smoother convergence.

Twin Delayed DDPG (TD3) is an extension of DDPG which uses two critics to further stabilize the learning process, as well as mitigate an issue with DDPG known as *over-estimation bias*. It also incorporates delayed policy updates which makes it closer to the actor-centric view discussed in Section 1.2.5, and adds clipped noise to the action used in the critic update.

Finally, Soft Actor-Critic (SAC) is a stochastic-policy Actor-Critic architecture which incorporates an entropy maximization term. This forces the policy to be as stochastic as possible, which improves the exploration capabilities and prevents some of the issues we identified in DDPG in Chapter 6.

2.2 Exploration mechanisms for behavioral learning

Many approaches attempt to improve the exploration capabilities of RL algorithms. The most straightforward way to improve exploration in RL is to initialize the policy with an educated guess, which is then refined until it is close to optimal. This makes a lot of sense when there are local minima in the policy for instance in the MountainCar environment: the policy-space is partitioned in several areas, some of which have *deceptive* gradients that lead gradient-based optimization processes to sub-optimal policies.

This can be extended to sets of initial policies: a separate RL instance can be trained using each policy, and after some time the best trained policy

is kept. Therefore, finding a diverse initial set of policies is a very promising goal, and is the main focus of Quality-Diversity approaches [Pugh et al., 2016; Cideron et al., 2020] such as Novelty Search with Local Competition (NSLC) [Lehman and Stanley, 2011] or MAP-Elites [Mouret and Clune, 2015]. This path leads directly to evolutionary algorithms, which are the seminal method for discovering and selecting parametric agents [Ursem, 2002]. Diversity has even been proposed as a full replacement to the RL reward signal, since emerging locomotion behavior can be discovered simply through diversity [Eysenbach et al., 2018].

Another set of algorithms do not explore the state-space or policy-space as a first step, but instead enforce behavioral diversity through a form of reward shaping. #Exploration [Tang et al., 2016] uses spatial hashing to penalize transitions leading to already-visited areas, while SAC [Haarnoja et al., 2018] maintains a stochastic policy by using an entropy maximization term in its update rule.

The other class of methods to improve exploration in RL is to pre-seed the experience replay buffer of an off-policy algorithm with varied data. This was attempted for instance by GEP-PG [Colas et al., 2018], which uses Goal Exploration Processes (GEP) [Forestier et al., 2017] to find a set of goals, and generate samples that are preloaded in the replay buffer of DDPG. However, in Chapter 6, we show that this technique is inherently limited by the fact that even algorithms such as DDPG which have a replay buffer and are said to be off-policy cannot always learn from data that is too different from their current policy.

Some methods are closer to MP in the sense that they do not necessarily aim at producing a robust controller, however they tackle the problem of exploration by enforcing diversity in the set of paths they explore. This idea of *path diversity* [Voss et al., 2015; Vonásek and Saska, 2018; Knepper and Mason, 2009; Erickson and Lavalley, 2009] mirrors the ones found in Quality-Diversity.

2.3 Attempts to use Motion Planning techniques with Reinforcement Learning

In this category, we only list the approaches that use MP directly in the configuration or state space. Algorithms that operate in the policy parameter space are listed in the previous section. Approaches that exploit backtracking or skill chaining are listed in Sections 5.2 and 7.1 respectively (pages 67, 97).

MP has been used as a primitive of a more general RL algorithm. In [Yamada, 2020], an RL algorithm is used but its action space is not the joint torque or speed, but instead the desired change in angle. In most RL settings, the actor directly controls the torque or speed of the actuators, and the action is therefore constrained to the maximum speed of the actuators over a single time step. However, in [Yamada, 2020] the action can be much greater, and when the magnitude of the selected action is above a certain threshold a motion planner is used to achieve the desired displacement instead of a torque controller.

On the other hand, RL has also been used as a primitive in MP. RL agents are sometimes used as trainable controllers that perform simple tasks such as torque control, however they can take a larger role. In [Faust et al., 2018; Chiang et al., 2019], an RL agent is used as a local planner, and another as a reachability estimator for the overarching MP architecture. The authors of [Bharadhwaj et al., 2020] tackle vision-based MP, and also use a RL agent as a reachability estimator.

In RL, goal-conditioned policies take as input not only the current state but also the target (goal), and output an action. This allows for training with *hindsight*: experience can be gathered from unsuccessful agents by moving the goal, and recording them as a successful attempt to reach a different goal [Andrychowicz et al., 2017]. This concept can be expanded to construct trees of goal-oriented policies [Lai et al., 2020].

Using motion planning in a reinforcement learning environment

Reinforcement Learning (RL) algorithms are designed to optimize a parametric policy by incrementally increasing its return. However, they usually have poor exploration mechanisms. On the other hand, efficient exploration is a common and well-studied topic in the field of Motion Planning (MP), and algorithms such as Rapidly-exploring Random Tree (RRT) are known to explore high-dimensional state spaces efficiently. However, the framework commonly used by RL algorithms is slightly different from the one in MP and is detailed in Section 1.1.3. On the other hand, the minimum requirement for RL algorithms is usually modeling the problem as a Markov Decision Process (MDP), in which the only information about the environment is obtained through interactions and rewards.

Therefore, transposing efficient exploration algorithms from MP to RL is not always straightforward.

When testing an exploration algorithm, it can be hard to define a metric for performance. However, the goal of exploration in the context of sparse-reward environments is to find sources of reward as soon as possible. Therefore, when comparing two exploration trees, we are interested mainly in its extent (the set of points that are closer than ϵ to the tree, for some ϵ). Exploration trees that maximize the extent also tend to have uniform densities, because clumps of close nodes do not contribute as much to increasing the extent as well-spread nodes.

In this chapter, we start by presenting the most classical version of RRT in Section 3.1.1, then in Section 3.1.2 we show that a small change in the algorithm allows it to operate under assumptions that are closer to those of RL environments. However, in Section 3.1.3 we show that the performance

of RRT quickly diminishes when the reachable space is smaller than the sampling space. In Section 3.3, we contribute a new algorithm called Ex, which overcomes this limitation and has computational complexity $O(1)$ with respect to the size of the exploration tree. Finally, in Section 3.4 we demonstrate the use of *feature spaces* as a dimension-reduction technique, by showing that both RRT and Ex are able to find trajectories that reach sparse rewards on the Ant-Maze environment.

3.1 Adapting Rapidly-exploring Random Tree to RL Algorithms

3.1.1 Rapidly-exploring Random Tree

RRT, presented in Algorithm 2 and Fig. 3.1, takes a uniformly random sample in the space (Line 4), and attempts to reach it (Line 6) from the tree’s nearest node (computed on Line 5). This attempt results in a new state s' (Line 7) which is added to the tree (Line 8). The uniformity of the chosen sample biases the algorithm towards exploring the biggest unexplored areas first.

However, RRT relies on three non-trivial primitives:

1. Line 6 of the algorithm relies on a heuristic function `ADVANCE_TOWARDS` which must be given as a primitive. This function takes as input a pair of states $(s_{\text{near}}, s_{\text{target}})$, and returns an action a such that, ideally, $s' = \text{STEP}(s_{\text{near}}, a)$ should bring the environment closer to s_{target} . This can sometimes be computed when a forward model is known and simple enough, but in the general case, and especially in non-holonomic environments, this heuristic is unavailable (see Section 1.1.2). In Section 3.1.2, we show that this primitive can be replaced by selecting a random action, at the cost of efficiency.
2. Line 4 relies on a primitive `RANDOM_STATE`, which selects a state uniformly in a *sampling space*. Although defining a bounded state-space is usually straightforward (the robot is fully described by a vector of real numbers and angles, and a vector of velocities and angular velocities which can be bounded by velocities that are too large to simulate in discrete-time), we show in Section 3.1.3 that the performance of RRT decreases when the sampling space is larger than the reachable space. This is a major caveat especially in highly-constrained environments ¹.

¹This is a well-known problem and some mitigations have been proposed [Shkolnik et al., 2009; Wu et al., 2020] although they are beyond the scope of this work. When working with

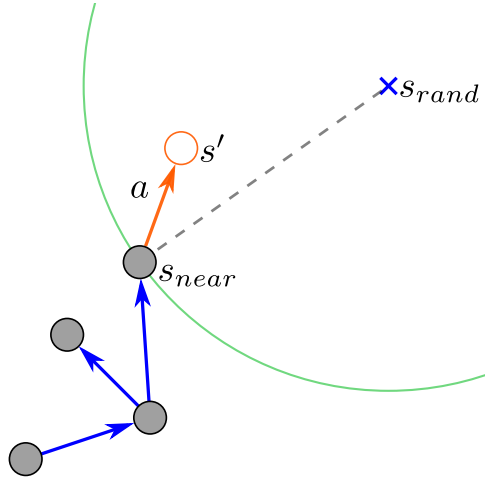


Figure 3.1: Behavior of RRT. For each iteration, first s_{rand} is chosen randomly, then its nearest neighbor in the exploration tree s_{near} is found, and finally an action a is chosen either using a heuristic (in the case of RRT), or randomly (in the case of Rapidly-exploring Random Tree Non-Holonomic (RRT-NH)). The resulting state s' is added to the tree.

3. Line 5 relies on a distance function over states. This metric is used to choose the already-explored state s_{near} that is closest to the sampled state s_{target} . The reasoning is that to reach the area around s_{target} , the best starting point is the closest. Contrary to ADVANCE_TOWARDS, this is part of the core of RRT, and is responsible for biasing the exploration toward unexplored areas. However, in environments with non-holonomic constraints or thin obstacles, the distance between two states may not be a good indicator of whether one can be reached from the other easily.

We test RRT on an environment with state space $[0, 5]^2$, action space $[-0.1, 0.1]^2$ and step function $s' = s + a$. If $s + a$ is out of bounds, then $s' = s$.

Figure 3.2(a) shows the result of running 2000 iterations of RRT in this environment. In the following sections, we visually compare the extent of the RRT tree with results from other algorithms.

humanoid robots, a more common algorithm is Probabilistic Roadmaps [Kavraki et al., 1996], which uses a local planner to connect nearby states. Using this planner was not needed for the scale of experiments we performed, although it may be a necessary trade-off between exploration performance and expert input when working with larger benchmarks such as humanoids.

Algorithm 2: Holonomic RRT

Input : $s_0 \in S$ the initial environment state
iterations $\in \mathbb{N}$ the number of samples to accumulate
 $d : S \times S \rightarrow \mathbb{R}^+$ a distance function over states
STEP : $S \times A \rightarrow S \times \mathbb{R} \times \mathbb{B}$ the environment step function
RANDOM_STATE, a function that returns a random state,
uniformly selected from the set of all possible states S
ADVANCE_TOWARDS : $S \times S \rightarrow A$ a function that suggests
an action to advance towards a new state
Output : transitions $\subseteq S \times A \times \mathbb{R} \times \mathbb{B} \times S$ the set of explored
transitions

```
1 transitions  $\leftarrow \emptyset$ 
2 visited  $\leftarrow \{s_0\}$ 
3 while |transitions| < iterations do
4    $s_{\text{target}} \leftarrow \text{RANDOM\_STATE}()$ 
5    $s_{\text{near}} \leftarrow \text{argmin}_{s \in \text{visited}} d(s, s_{\text{target}})$ 
6    $a \leftarrow \text{ADVANCE\_TOWARDS}(s_{\text{near}}, s_{\text{target}})$ 
7    $s', \text{reward}, \text{terminal} \leftarrow \text{STEP}(s_{\text{near}}, a)$ 
8   transitions  $\leftarrow \text{transitions} \cup \{(s, \text{action}, \text{reward}, \text{terminal}, s')\}$ 
9   if Not terminal then
10    | visited  $\leftarrow \text{visited} \cup \{s'\}$ 
11  end
12 end
```

3.1.2 RRT in RL environments

A variant of RRT relaxes the assumption that the ADVANCE_TOWARDS heuristic is available, by instead choosing a random action at each iteration. The exploration is still biased towards unexplored areas because of the non-random choice of s_{near} . According to [Lavalle and Kuffner, 2000], this is mainly useful for non-holonomic environments, therefore we refer to this variant as RRT-NH, and present it in Algorithm 3.

Algorithm 3: Non-holonomic RRT

Input : $s_0 \in S$ the initial environment state
iterations $\in \mathbb{N}$ the number of samples to accumulate
 $d : S \times S \rightarrow \mathbb{R}^+$ a distance function over states
STEP : $S \times A \rightarrow S \times \mathbb{R} \times \mathbb{B}$ the environment step function
RANDOM_STATE, a function that returns a random state,
uniformly selected from the set of all possible states S
RANDOM_ACTION, a function that returns a random action

Output : transitions $\subseteq S \times A \times \mathbb{R} \times \mathbb{B} \times S$ the set of explored transitions

```

1 transitions  $\leftarrow \emptyset$ 
2 visited  $\leftarrow \{s_0\}$ 
3 while |transitions| < iterations do
4    $s_{target} \leftarrow \text{RANDOM\_STATE}()$ 
5    $s_{near} \leftarrow \text{argmin}_{s \in \text{visited}} d(s, s_{target})$ 
6    $a \leftarrow \text{RANDOM\_ACTION}()$ 
7    $s', \text{reward}, \text{terminal} \leftarrow \text{STEP}(s_{near}, a)$ 
8   transitions  $\leftarrow \text{transitions} \cup \{(s, \text{action}, \text{reward}, \text{terminal}, s')\}$ 
9   if Not terminal then
10    | visited  $\leftarrow \text{visited} \cup \{s'\}$ 
11  end
12 end

```

Figure 3.3(a) shows the result of running 2000 iterations of RRT-NH in the same box environment we used in Fig. 3.2(a). We can observe that the state space coverage is not as uniform as with RRT, which is expected. We can also note that many transitions are clumped around the main branches of the tree, which resembles the pattern of random walk. This is understandable since RRT-NH is essentially a directed random walk.

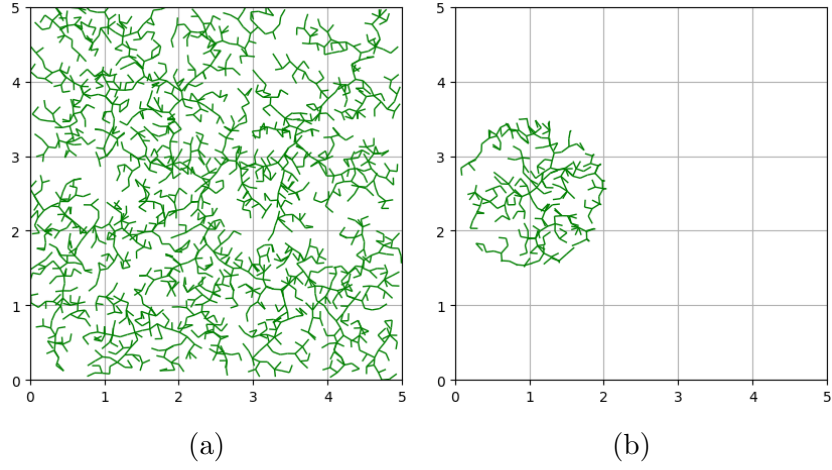


Figure 3.2: Running 2000 iterations of RRT in (a) a box environment and (b) a box in which only an off-center disc is reachable.

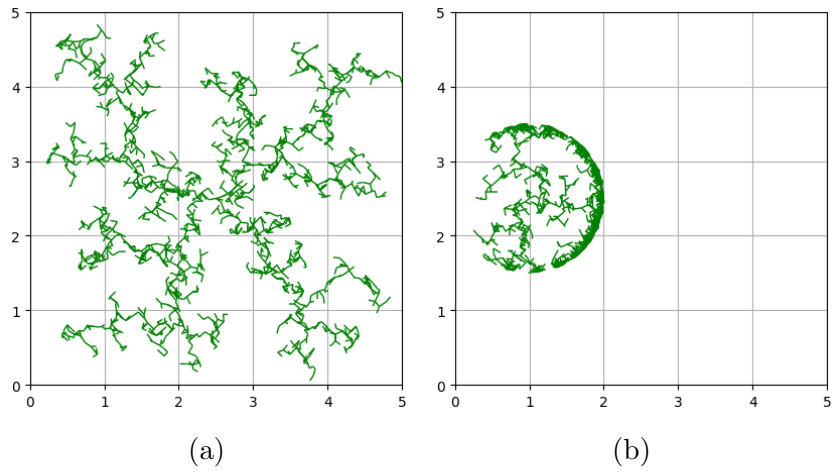


Figure 3.3: Running 2000 iterations of RRT-NH in (a) a box environment and (b) a box in which only an off-center disc is reachable.

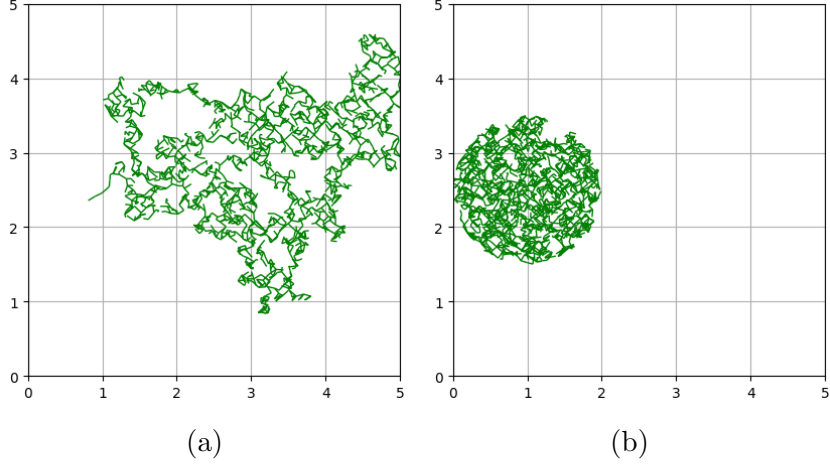


Figure 3.4: Running 2000 iterations of Ex in (a) a box environment and (b) a box in which only an off-center disc is reachable.

3.1.3 Limitations of RRT due to sampling

In the previous section, we showed that RRT could be implemented in RL environments by removing the need for the `ADVANCE_TOWARDS` primitive. In this section, we show that the use of the `RANDOM_STATE` primitive is also a weakness that reduces the exploration efficiency of RRT-NH. RRT explores an empty environment very quickly, however it shows major limitations when the sampling space is larger than the reachable state space.

To demonstrate this, we modified our test environment so that the reachable space R is limited to only a disc-shaped subset of $S = [0, 5]^2$, but S is still used for sampling in RRT. This restriction was implemented by changing the step function so that moving outside of the disc results in no change of the state².

The results are presented in Figs. 3.2(b) and 3.3(b), and show that the density of the random tree inside the reachable space R is similar to the density observed when running the algorithm without reachability restrictions. In other words, only the samples that were selected inside of R contributed to exploration.

Indeed, RRT biases exploration towards samples taken in S , which results in an uneven distribution inside of reachable space R . This problem can seem minor in this example, but we expect it to be worse with increasing dimensionality, for the reasons stated in the previous paragraphs.

²In order to emphasize the effect, when $s + a$ is outside the disc, s' is set to $s + \epsilon$, where ϵ is a small perturbation. Therefore, areas where many such attempts were made appear denser in the figures.

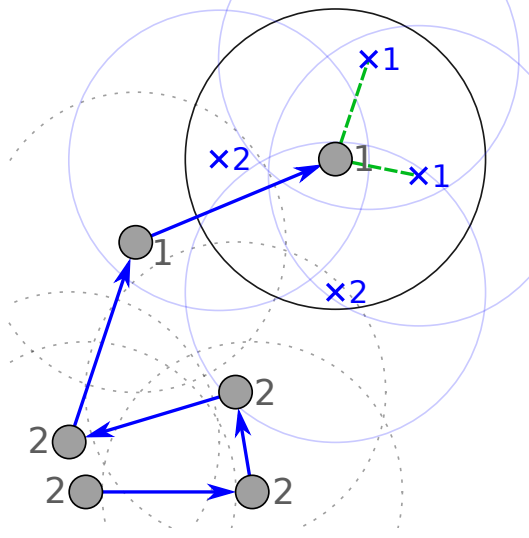


Figure 3.5: Behavior of Expansive Spaces Tree (EST). For a fixed distance d , each node in the tree is assigned a score $\omega(s)$, which is the number of nodes (including itself) closer than d . A node is sampled from the tree according to $\frac{1}{\omega(s)}$, and k new states are sampled randomly in its neighborhood. ω is also computed for these new states, and samples with high ω are discarded, while the others are added to the tree.

These cases show the limitations of sampling-based exploration techniques: they bias the search towards large unexplored areas, but are only efficient when a good approximation of the reachable space is known, otherwise "unexplored areas" has a loose definition.

On the opposite end of the spectrum, an unbiased random walk fills the reachable space, but with a non-uniform density: states closer to the starting point are more likely to be seen. A middle-ground is found by performing a random walk while favoring less-dense areas of the search tree, which is the idea we explore in Section 3.3.

3.2 Expansive Spaces Tree

Expansive Spaces Tree (EST) [Hsu et al., 1997] is a MP algorithm that builds an exploration tree similar to the one of RRT, but uses a different heuristic when choosing from which node of the tree to expand. Figure 3.5 describes the operation of the algorithm.

Where RRT and RRT-NH select the closest node to a randomly sampled target, EST chooses a node x that minimizes $\omega(x)$, where $\omega(x)$ is defined as

the number of nodes that are too close to x , where *too close* is defined using a threshold distance d that is a hyperparameter of the algorithm. Once the node x from which to expand has been chosen, EST samples several candidate targets close to x , and tries to add them to the tree.

Similar to RRT, EST requires either a reachability primitive or an ADVANCE_TOWARDS primitive, however EST can be adapted to RL environments: once a state s is selected using the ω distribution, random actions are applied from state s to discover new states instead of sampling new states in the neighborhood of s .

EST can also be used with a heuristic to bias the exploration direction, although this heuristic may be hard to formulate in complex problems and be deceptive [Phillips et al., 2004].

3.3 Ex

In this section, we propose a new exploration algorithm called Ex inspired from EST [Hsu et al., 1997], which biases a random walk towards parts of the tree that lie in less dense areas of the search space. Contrary to RRT, it does not rely on sampling and is therefore immune to the issues presented in Section 3.1.3.

3.3.1 Pitfalls of density as a measure for novelty

As described in Section 3.1.3, there are two main ways of building an exploration tree: biasing the exploration towards large unexplored areas by sampling them and choosing nodes that are closest (this is the approach taken by RRT and RRT-NH), and biasing the exploration towards areas in which it has the lowest density (approach taken by EST, #Exploration and Ex).

However, we found a pitfall when using density as a metric, which is summarized in Fig. 3.6. When using a disc kernel in order to count the number of neighbor nodes, if the radius r of this kernel is smaller than the maximum length of a graph edge, then exploring from a node n may not increase the density around n . This can lead to a deadlock, especially if the newly-created nodes land in a dense area, as depicted in Fig. 3.6.

Several approaches can be taken in order to eliminate this deadlock:

1. Ensure the size of the kernel is greater than the maximum edge length of the exploration graph. This guarantees that the density around the origin node n increases by at least one, and since any other density can only be increased by one when adding a new node to the graph, this guarantees the absence of deadlocks (providing ties in density values

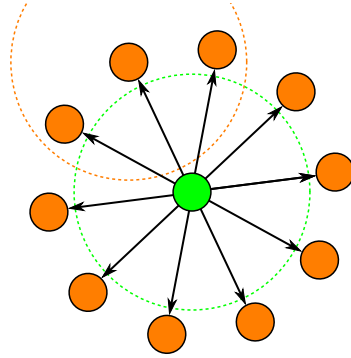
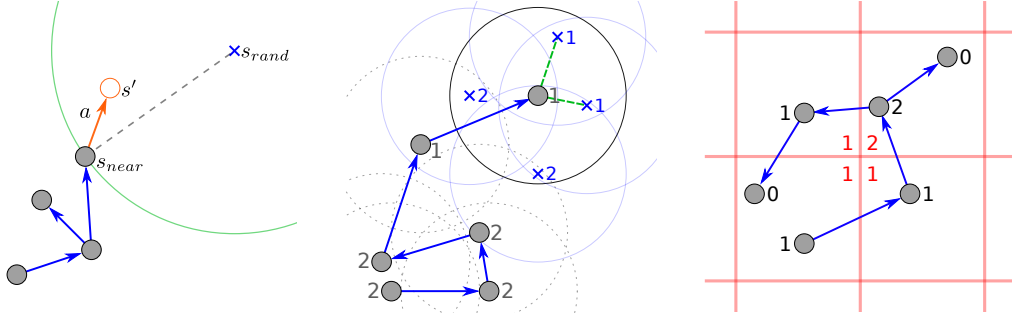


Figure 3.6: Density paradox: in this exploration tree, the radius r used to compute the density around a node is lower than the distance between two consecutive states. Therefore, if the metric for density is the number of nodes within a disc of radius r , then the density around the center node is 1, whereas the density around the surrounding nodes is 3. As a result, the center node appears to be a better choice for expansion of the search tree, causing more nodes to be added around it. This increases the density in the ring around the center, but not the density computed *at* the center, ultimately causing a deadlock.

are handled randomly). Unfortunately, the maximum length of edges in state-space is often hard to bound, or may even be unbounded in some cases, which makes this approach unwieldy for all except simple exploration problems.

2. Measure the density around node n as the number of nodes in the disc kernel, but add any child nodes of n even if they lie outside the kernel. This approach prevents the deadlock but is not as easy to implement as the following.
3. Measure the density around node n as the sum of the number of children of all nodes within the disc kernel, including n itself. In other words, this amounts to counting the number of outgoing arrows within the kernel (including arrows that leave the disc and arrows that stay in the disc).

This third approach is the basis for our proposed algorithm Ex, which is designed to bias exploration in less-dense parts of the search tree and is presented in the following section.



(a) Behavior of RRT. For each iteration, first s_{rand} is chosen randomly, then its nearest neighbor in the exploration tree s_{near} is found, and finally an action a is chosen either using a heuristic (in the case of RRT), or randomly (in the case of RRT-NH). The resulting state s' is added to the tree.

(b) Behavior of EST. For a fixed distance d , each node in the tree is assigned a score $\omega(s)$, which is the number of nodes (including itself) closer than d . A node is sampled from the tree according to $\frac{1}{\omega(s)}$, and k new states are sampled randomly in its neighborhood. ω is also computed for these new states, and samples with high ω are discarded, while the others are added to the tree.

(c) Behavior of Ex. For each graph node, the number of outgoing edges is counted. For each bin, another counter (here in red) keeps track of the number of all node counters in the bin. At each step, the non-empty bin with the lowest counter is selected, and in this bin the node with the lowest counter is selected.

Figure 3.7

3.3.2 The Ex algorithm

In this section, we contribute an algorithm called Ex which builds an exploration tree in a MDP in $O(1)$ time³ by expanding the tree from well-selected explored states with random actions. Ex differs from RRT-NH in the choice of the visited states from which to explore. While RRT-NH samples distant states and selects the nearest visited state, Ex selects a state that is in a less-explored region of the search tree. The definition of *less-explored* is a combination of the number of times this state was selected, and the number of times nearby states were selected, using binning to define nearby states efficiently.

The Ex algorithm is presented in Algorithm 4, and illustrated in Fig. 3.7(c).

³Each expansion operation increases the size of the tree by 1 and is performed in $O(1)$ time on average

The core idea of the algorithm is to divide the state-space in bins, and maintain the following counters:

1. For each node s in the search tree, its number of children c_s
2. For each state-space bin b , the total number of children of all nodes in the bin $c_b = \sum_{s \in b} c_s$.

At each iteration of Ex, a state is chosen in the search tree in two steps: first, the bin b with the lowest c_b is chosen. Then, the node $s \in b$ with the lowest c_s is selected. The environment is reset to this state s , and a random action is performed. The resulting state s' is added to the search tree and the counters are updated accordingly.

When several bins or states are tied for the lowest c_b or c_s , then one of the possible tied bins or states is chosen uniformly randomly.

Algorithm 4 presents an overview of Ex, while Algorithm 5 describes how Ex can be implemented so that each iteration is performed in $O(1)$ time.

O(1) implementation of Ex. The $O(1)$ implementation of Ex presented in Algorithm 5 relies on these principles:

1. Hash tables are used in order to quickly lookup data about a bin, using a spatial hashing technique to find the correct bin for any given state in constant time. For instance, in a n -dimensional state-space, a grid with resolution $1 \times \dots \times 1$ (meaning bins have size $1 \times \dots \times 1$) can be implemented by the following spatial hash: state (s_1, \dots, s_n) can be mapped to the integer coordinates $(\lfloor s_1 \rfloor, \dots, \lfloor s_n \rfloor)$, which can then be hashed using any tuple hashing function.
2. In order to keep track of the number of outgoing edges from bins, all bins with the same number of outgoing edges c are stored in a list $W_b[c]$. A variable m keeps track of the lowest c .
3. States within a bin are stored in a similar manner, grouped by their number of outgoing edges in a hash table B_s that maps a bin hash b and state counter c to the list of states in bin b that have c outgoing edges. The minimum number of outgoing edges in a bin b is kept in $B_m[b]$.

Selecting a state from which to expand in the search tree is done in constant time by selecting a bin hash b randomly in $W_b[m]$, and moving this bin hash to the next group $W_b[m+1]$ since a single outgoing edge is going to

be added to it. The counter m is incremented if this causes $W_m[m]$ to become empty.

In order to select a state in this bin b , first the minimum number of outgoing edges for a state in this bin is looked up in the hash table B_m . This value is $m_b = B_m[b]$. A state is randomly selected from $B_s[b][m_b]$ and moved to the next group $B_s[b][m_b + 1]$. $B_m[b]$ is incremented if the originating group becomes empty.

In this analysis we considered that the state-space dimension d was not a variable and focused on the computational complexity with respect to the number of traversed states n . The state-space dimension affects both RRT and Ex similarly. Note that although the spatial hashing used by Ex is asymptotically faster than the nearest-neighbor search of RRT, both EST and RRT have approximate variants that reduce their complexity to $O(1)$ through various techniques including spatial hashing [Leskovec et al., 2020].

Algorithm 4: Ex algorithm

Input : $s_0 \in S$ the initial environment state
step : $S \times A \rightarrow S \times \mathbb{R} \times \mathbb{B}$ the environment step function
iterations $\in \mathbb{N}$ the number of samples to accumulate
Bin : $S \rightarrow \mathbb{N}$ a binning function

Output : The search tree

- 1 Initialize the exploration set T to a single node s_0
- 2 $c_{s_0} \leftarrow 0$
- 3 **while** |search tree| < iterations **do**
- 4 $b \leftarrow \operatorname{argmin}_{b \in \{\operatorname{Bin}(s), s \in T\}} \sum_{s \in T, \operatorname{Bin}(s)=b} c_s$
- 5 $s \leftarrow \operatorname{argmin}_{s \in T, \operatorname{Bin}(s)=b} c_s$
- 6 Increment c_s
- 7 $a \leftarrow \operatorname{random_action}()$
- 8 $s', \text{reward}, \text{terminal} \leftarrow \text{step}(s, a)$
- 9 $T \leftarrow T \cup \{s'\}$
- 10 If $c_{s'}$ is undefined, then $c_{s'} \leftarrow 0$
- 11 **end**

3.3.3 Related work

EST. In Section 3.2, we described the EST algorithm. Ex is different from EST in three major aspects:

Algorithm 5: Ex algorithm

Input : $s_0 \in S$ the initial environment state
step : $S \times A \rightarrow S$ the environment step function
iterations $\in \mathbb{N}$ the number of samples to accumulate
BinHasher : **State** \rightarrow BinHash a binning function that maps a point in state space to a bin hash

Output : The search tree

- 1 W_b is a hash table $\text{Int} \rightarrow \text{List}(\text{BinHash})$
- 2 B_s is a hash table $\text{BinHash} \times \text{Int} \rightarrow \text{List}(\text{State})$
- 3 B_m is a hash table $\text{BinHash} \rightarrow \text{Int}$
- 4 $m \leftarrow 0$
- 5
- 6 **exInsert**(s_0)
- 7 **for** $i \leftarrow 0$ **to** iterations **do**
- 8 $s \leftarrow \text{exSelect}()$
- 9 $a \leftarrow \text{random_action}()$
- 10 $s' \leftarrow \text{step}(s, a)$
- 11 **exInsert**(s')
- 12 **end**
- 13
- 14 **Function** **exInsert**(s)
- 15 $b \leftarrow \text{BinHasher}(s)$
- 16 **if** $b \notin B_m.\text{keys}()$ **then**
- 17 $m \leftarrow 0$
- 18 $W_b[0].\text{append}(b)$
- 19 **end**
- 20 **binInsert**(b, s)
- 21 **Function** **binInsert**(b, s)
- 22 $B_s[b][0].\text{append}(s)$
- 23 $B_m[b] \leftarrow 0$
- 24 **Function** **exSelect**()
- 25 $b \leftarrow$ a random element from list $W_b[m]$
- 26 Remove b from the list $W_b[m]$
- 27 $W_b[m+1].\text{append}(b)$
- 28 If $W_b[m]$ is empty, then $m \leftarrow m+1$
- 29 **return** **binSelect**(b)
- 30 **Function** **binSelect**(b)
- 31 $m_b \leftarrow B_m[b]$
- 32 $s \leftarrow$ a random element from list $B_s[b][m_b]$
- 33 Remove s from the list $B_s[b][m_b]$
- 34 $B_s[b][m_b+1].\text{append}(s)$
- 35 If $B_s[b][m_b]$ is empty, then $B_m[b] \leftarrow m_b+1$
- 36 **return** s

1. Ex does not use the number of close nodes as a metric for density, but instead uses a counting method based on a fixed grid.
2. Once a node from which to expand has been chosen, Ex performs a random action from this node similarly to RRT-NH.
3. Ex can be implemented so that it chooses the node from which to expand in constant time, whereas EST requires several nearest-neighbor operations for each expansion.

#Exploration. On the RL side, Ex is closest to #Exploration [Tang et al., 2016], an algorithm that shapes the reward used by a RL algorithm to penalize states that have already been explored. The method used to determine whether a state is novel is very similar to Ex, and uses a grid-based counter. However, this novelty measure is not used to build an exploration tree but rather to guide a RL algorithm towards novel regions.

3.4 Experiments using a feature space

3.4.1 The curse of dimensionality

At this point, it seems important to mention the curse of dimensionality [Pratt, 2018]: in state-spaces with high dimensionality, the notion of density gets weaker, and a random walk has exponentially low probabilities of encountering itself [Erdős and Taylor, 1960]. In other words, large spaces are easier to explore because random walks are likely to always yield novel states, in the sense that they are far apart in Euclidean space. However this distance gives equal importance to all state-space parameters: joint angles and floating base coordinates. This may not reflect the actual exploration that is desired, which in the case of a mobile robot would be the exploration of the environment. Exploring the set of robot poses is necessary to some extent, but only as a means to move in the environment.

This curse affects bin-based algorithms such as Ex slightly less because bins are a representation of the uniform norm⁴: a bin with center c and edge size $2r$ is the set of states $\{s \mid \|s - c\|_\infty \leq r\}$.

This means that the number of bins needed to cover a state-space $[0, R]^D$ with bins of size 1 is R^D . As a comparison, using Euclidean norm (or

⁴Also called infinity norm

hypersphere) kernels, covering the same state-space would require $R^D D^{\frac{D}{2}}$ spheres⁵.

RRT and EST could both be implemented using the uniform norm, but the implications of this change are beyond the scope of this work. Indeed, in robotics problems we need to explore high-dimension state spaces, and when D is on the order of 40, both R^D and $2R^D D^{\frac{D}{2}}$ are equally intractable.

3.4.2 Feature spaces

In a robot with many degrees of freedom, exploring each and every possible pose is not desirable. Instead, the objective is exploring high-level motions such as moving the floating base in the 3D environment. The low-level motion of articulations and controllers can be handled by random-walk motion alone.

This can be achieved by introducing a feature function $f : S \rightarrow F$ that maps a state s to a smaller vector in a *feature space*, which is used to drive the exploration.

This concept can be implemented in RRT-NH by computing the distance between $f(s)$ and $f(s_{\text{target}})$ instead of between s and s_{target} . It can also be implemented in Ex by changing the binning function to hash $f(s)$ instead of s . An example of feature function is described in Section 3.4.3.

Many high-dimension mobile robotic environments have a straightforward feature space which is the position and orientation of the root link of the robot, however in the general case the choice of a feature space is an expert input which greatly conditions the exploration process and limits the generality of this work.

In the following section, we use feature spaces to benchmark Ex and RRT on the ANTMAZE environment which has 8 action parameters, 14 degrees of freedom and a state-space of dimension 28.

3.4.3 Benchmark on Ant-Maze

We tested the performance of Ex and RRT-NH on a benchmark called ANTMAZE [OpenAI, 2018] which features a mobile four-legged robot with 14 degrees of freedom and 28 state variables traversing a 2x2 maze, similar to the ones studied in Chapter 4. Figure 3.8 shows a render of the ANTMAZE environment.

We tested RRT-NH and Ex on this environment by using the (x, y) coordinates of the mobile robot as a feature space. In other words, when

⁵This result is found by densely packing the hypercubes inscribed in the hyperspheres of dimension D , which have size $D^{-\frac{1}{2}}$ when the hyperspheres have radius $\frac{1}{2}$.

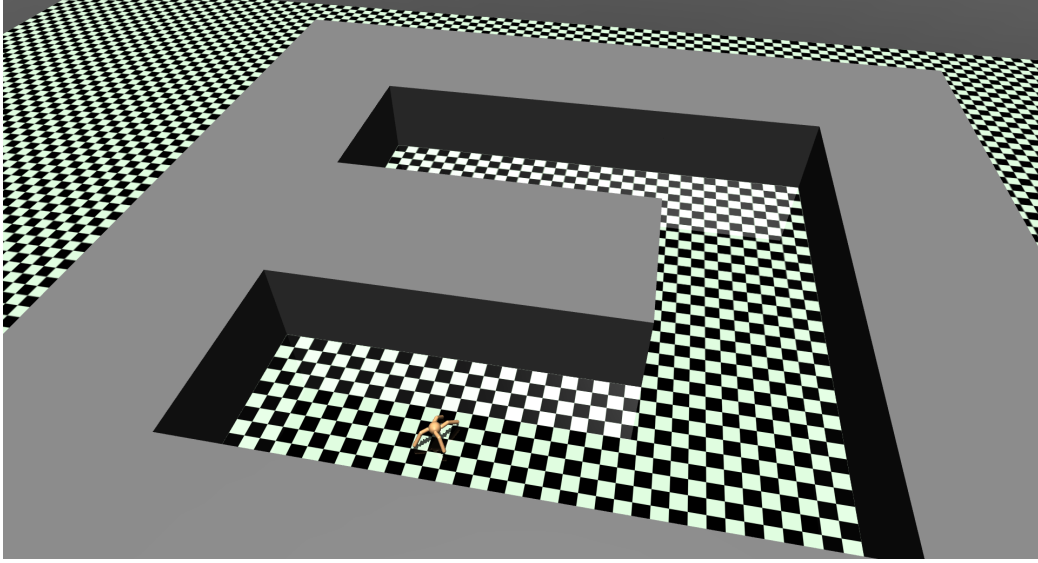


Figure 3.8: Render of the ANTMAZE environment. The initial position of the robot is the bottom-left hand corner of the maze and the goal is in the top-left hand corner.

computing the spatial hashing function of Ex or the nearest point in RRT-NH, only the global position in the maze is taken into account instead of each angle and velocity of the system.

We ran these simulations with several seeds, and present the results in Fig. 3.9. Ex significantly outperformed RRT in this environment, finding the reward quickly, while RRT was often unsuccessful after 600k steps and had to be interrupted. Although it was not the main focus of this study and therefore not visible in Fig. 3.9, the wall clock time was also significantly in favor of Ex with all 100 seeds finishing within a day, while RRT has to be interrupted after several weeks.

3.5 Conclusion

In this chapter, we showed that MP algorithms such as RRT-NH are a viable solution when exploring even high-dimension RL environments, and proposed a novel algorithm called Ex that outperforms RRT on ANTMAZE and offers different properties that make it more suited to some classes of problems where the sampling space required by RRT is hard to define.

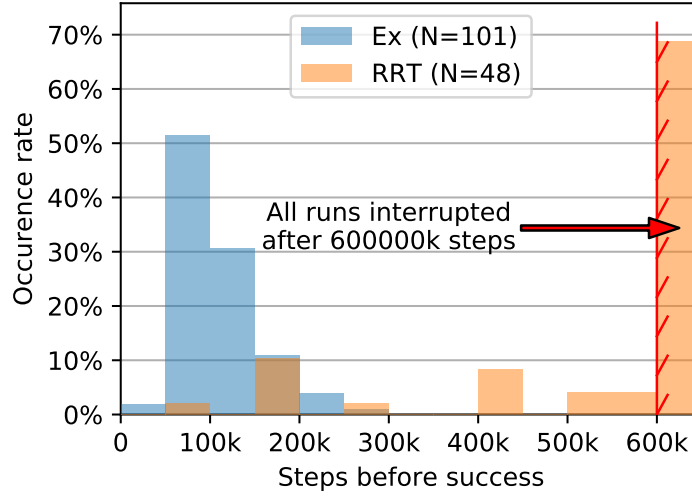
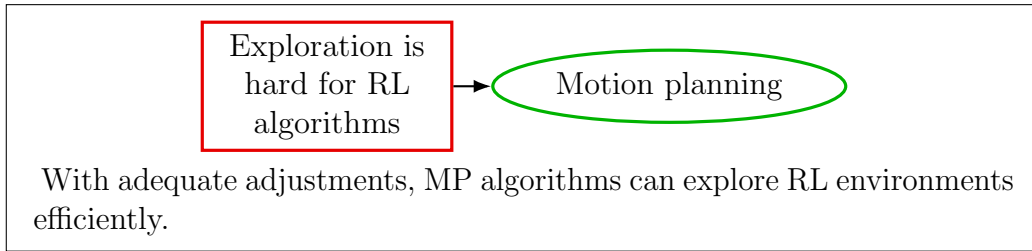


Figure 3.9: Performance of Ex and RRT-NH in the ANTMaze environment. The simulation was run with different seeds, and the x-axis represents the number of environment steps before the reward was found. The simulation is considered a failure when the reward is not found within 600k steps.



In the next chapter, we show that these MP methods outperform RL algorithms in exploration tasks, however converting the exploration data to a robust controller is a difficult task, that cannot be solved by simply bootstrapping so-called off-policy RL algorithm with this data.

Exploiting exploration data in Reinforcement Learning through the experience replay buffer

In the previous chapter, we presented several algorithms adapted from the field of Motion Planning (MP) which are able to find sparse rewards. These algorithms output a tree of states, which may contain one or more paths from the starting state to a rewarded state. When controlling an actual robot, the list of actions required to reach the goal can be implemented in the form of an open-loop controller, or a trajectory-following controller. However, open-loop control has limited applications because it is not resilient to outside perturbations, and trajectory-following controllers usually require a model of the robot. Furthermore, experience shows that optimizing this trajectory can be a hard problem even when a model of the robot is known, and it gets harder for robots with complex dynamics and many non-holonomic constraints.

Closing the loop is notoriously difficult, especially since in the context of Reinforcement Learning (RL) environments very little is known about the control law of the robot. On the other hand, RL algorithms directly output a controller that is resilient to noise and perturbations.

Therefore, a major challenge of robotics is transferring the knowledge acquired through MP to a controller trained using RL.

In this chapter, we dive deeper in the characteristics of the exploration data resulting from MP and the components of Deep Deterministic Policy Gradient (DDPG) to find how the two can be combined. First, we build an intuition by observing the behavior of DDPG, Rapidly-exploring Random Tree Non-Holonomic (RRT-NH) and Ex on simple continuous mazes in which walls are generated in a 2×2 grid (all environments in this thesis are continuous therefore we simply refer to them as 2×2 maze environments). Then, we attempt to transfer exploration data by pre-loading the DDPG experience

replay buffer with transitions from an exploration algorithm. Although some promising results are observed and provide insight in the inner workings of DDPG, these results do not scale. This leads us to present an argument in Section 4.7 for the need to exploit the connectivity of the exploration tree instead of individual transitions.

4.1 Experimental setup

In this chapter, we run experiments in a 2×2 maze environment with thin walls and a sparse reward. The agent starts at $(.5, .5)$, and the reward is -1 if the agent touches the wall, 1 if the Euclidean distance to the goal $(.5, 1.5)$ is less than 0.2 , and 0 otherwise. No state is terminal, and when performing DDPG rollouts we add a 10% chance that a random action is applied instead of the policy.

More formally, the environment is described by the following Markov Decision Process (MDP):

$$\begin{aligned} S &= [0, 2] \times [0, 2] \\ A &= [-0.1, 0.1] \times [-0.1, 0.1] \\ \text{step}(s, a) &= \begin{cases} s & \text{if } [s, s+a] \text{ intersects a wall} \\ s+a & \text{otherwise.} \end{cases} \\ R(s, a, s') &= \mathbb{1}_{\|s'-(.5,1.5)\|<0.2} - \mathbb{1}_{[s,s+a] \text{ intersects a wall}} \end{aligned}$$

In this thesis, unless specified otherwise DDPG is used with its default parameters from the OpenAI Baselines implementation [Github, 2020b].

4.2 Behavior of DDPG, RRT-NH and Ex on mazes

Figure 4.1 shows the behavior of DDPG over 10k steps in our test environment. We observe that DDPG trained a policy that avoids the walls but does not reach the reward. This is expected since DDPG has a basic built-in exploration mechanism, and touching the walls carries a high penalty. This is an example where the exploration-exploitation trade-off is exacerbated by deceptive rewards.

Figures 4.2(a) and 4.2(b) show the exploration pattern of Ex and RRT-NH up to the first reward found. It shows that both algorithms are able to find the reward very quickly, as expected for algorithms designed for exploration and that exploit the reset-anywhere primitive.

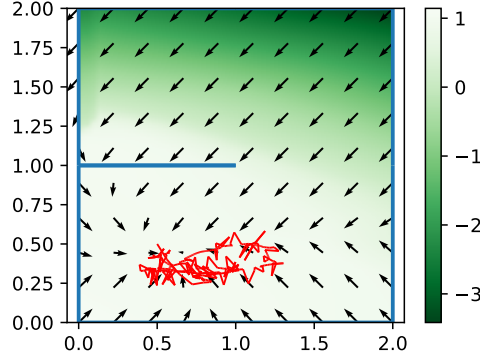


Figure 4.1: State of DDPG after 10k training steps in a 2×2 maze. The green gradient represents the function $Q(s, \pi(s))$ where Q and π are the critic and actor trained by DDPG. The black arrows show the direction of $\pi(s)$, and the red line is a single rollout of DDPG.

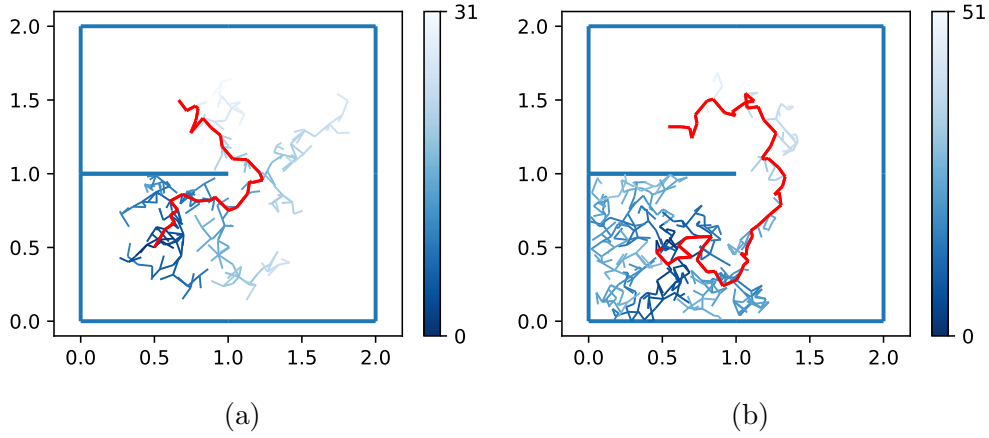


Figure 4.2: In blue, exploration pattern of (a) RRT-NH and (b) Ex. The color value indicates the depth of each transition in the exploration tree. Exploration stops as soon as a single rewarded transition is found, and the corresponding path is shown in red. This path is about (a) 3 or (b) 5 times longer than the optimal path.

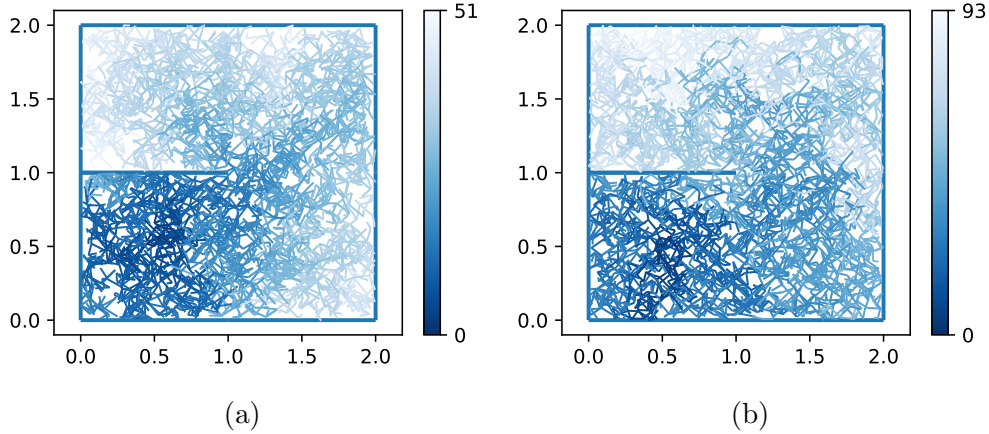


Figure 4.3: 5k exploration steps of (a) RRT-NH and (b) Ex. The color value indicates the depth of each transition in the exploration tree. Contrary to Fig. 4.2, the exploration continues regardless of any reward found.

However, this exploration data does not give a full picture of the environment: most of the walls have not been found, therefore we cannot reasonably expect DDPG to be able to improve on this path. Figures 4.3(a) and 4.3(b) show the coverage reached when we let the exploration continue for a fixed number of steps, even if the reward is found earlier, so that the set of transitions covers the full set of possible state-action pairs near-uniformly. These figures use 5k exploration steps for clarity, but the following experiments are performed with 10k exploration steps.

4.3 Motivation

In this chapter, we make the argument that DDPG cannot make full use of the set of transitions collected by a MP algorithm such as RRT-NH or Ex, and therefore is not fully off-policy. An algorithm that can learn from a fixed set of data and nothing more is called an *Offline* or *Batch* Reinforcement Learning algorithm [Lange et al., 2012]. Therefore, we test to what extent DDPG can be used in an offline setting. Some offline RL algorithms have already been proposed for continuous-state and continuous-action environments, but are beyond the scope of this work, see [Levine et al., 2020] for a recent survey. In Section 4.7 we argue that even such algorithms would have issues learning from a set of transitions in some challenging environments with thin walls.

4.4 Methods

The following section studies two variants of DDPG that exploit exploration data. The first variant, *offline DDPG*, is defined as such:

1. First, a set of transitions is collected using Ex.
2. Then, this set of transitions is inserted into the experience replay buffer of DDPG.
3. Finally, DDPG is run but no new experience is collected: the replay buffer is not modified and no rollouts are performed. Instead, the actor and critic networks of DDPG are simply allowed to converge by training them using the usual losses.

In the second variant called *bootstrapped DDPG*, the DDPG algorithm is able to collect additional experience while training, but is still bootstrapped with the transitions obtained from the exploration phase. Bootstrapped DDPG is defined as:

1. First, a set of transitions is collected using Ex.
2. Then, this set of transitions is inserted into the experience replay buffer of DDPG.
3. Finally, DDPG is run and new experience is collected using rollouts and appended to the experience replay buffer.

4.5 Results

Figure 4.4 shows the actor and critic that are trained by offline DDPG, which does not record any further experience except the exploration data given by Ex. It shows that the reward was indeed acknowledged and results in a high critic value, however the wall seems to be completely transparent, and the resulting actor is not able to solve the environment.

However, as shown in Figs. 4.5(a) and 4.5(b), bootstrapped DDPG quickly finds the reward.

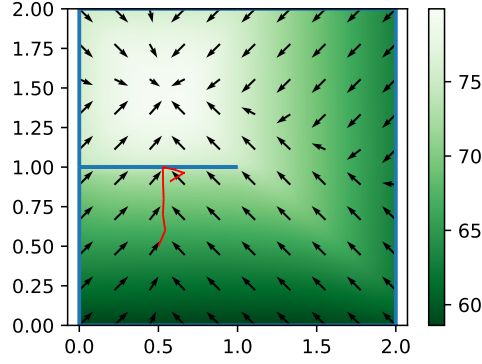


Figure 4.4: Actor and critic trained by offline DDPG: 10k exploration transitions from Ex are preloaded in the replay buffer and no new experience is added during training

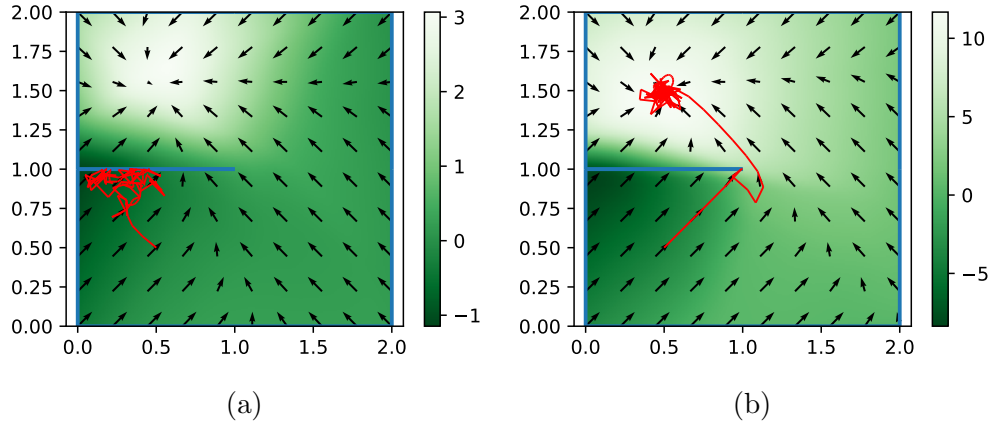


Figure 4.5: State of bootstrapped DDPG when its replay buffer pre-filled with 10k transitions obtained using Ex, after (a) 1000 (b) 2000 rollout steps. Note the different color scales.

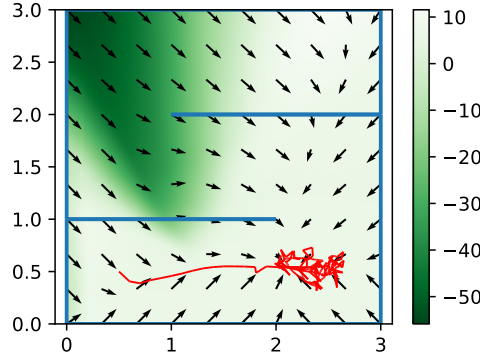


Figure 4.6: State of bootstrapped DDPG when its replay buffer is pre-filled with 20k transitions obtained using Ex, after 20k rollout steps

4.6 Analysis

Looking closely at the critic values along the wall shows the mechanisms of this recovery. Initially, the actor learns from the optimistic critic that the best course of action is to go straight into the wall. Transitions hitting the wall were already present in the experience replay buffer, however now that the rollouts are heading straight into the wall, the number of such transitions increase, which causes them to become more prevalent in sampled mini-batches. In turn, the critic values near the wall decrease until the actor is led towards a new path.

In this example, it is clear that the existence of transitions indicating the position of the wall is not enough: their number and therefore prevalence is a critical factor in the update of the critic. Therefore, DDPG is only able to function properly when the distribution of samples in its experience replay buffer roughly approximates the behavior of its current actor. This is an argument for a more precise definition or a method of measuring how off-policy an algorithm is.

Although the exploration data alone was not enough for DDPG to solve the 2×2 environment, preloading the exploration buffer with exploration data while retaining the standard DDPG rollouts helped it stay optimistic about the existence of reward, and prevented the deadlock we observed when no exploration data was provided.

This seems like a promising approach, and was used by [Hollenstein et al., 2019; Colas et al., 2018] but, as we show in Fig. 4.6, this technique rapidly falls short in more complex environments because the rollout data overwhelms the exploration samples and the critic “forgets” about the existence of the reward.

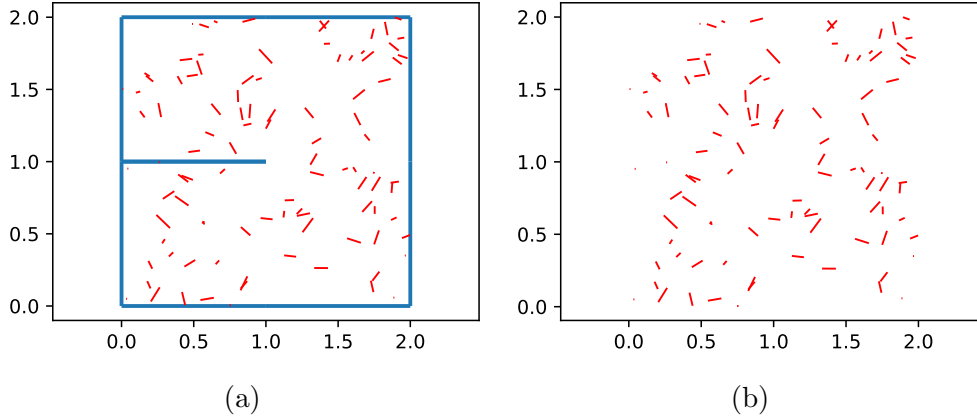


Figure 4.7: Mini-batch of 100 transitions from exploration data generated by Ex on a 2×2 maze. The information is displayed (a) with and (b) without the maze environment and outlines the difficulty of finding the wall when the data is presented as a disjointed set of transitions, one mini-batch at a time.

In the next section, we argue that no RL algorithm can learn from the set of transitions generated by Ex or DDPG in this 2×2 maze because the representation of this exploration data lacks connectivity information.

4.7 Thin walls and the limits of “sets of transitions”

In this section, we perform a thought experiment in which we consider the work to be done by a RL algorithm that receives exploration data in the form of a set of transitions that is pre-loaded in its experience replay buffer, and which are then sampled one mini-batch at a time.

Fig. 4.7(a) shows a mini-batch of 100 transitions extracted from an Ex exploration tree. As the RL algorithm has no knowledge of the environment except through the collected experience, Fig. 4.7(b) is a better representation of the task that is expected of an offline learning algorithm. This shows the loss of information when converting a tree to a set of disjointed transitions; Since there is no reasonable way of finding the wall from this data (even when using multiple mini-batches), any off-policy algorithm that maintains some form of heuristic based on the state (such as an actor and a critic) will propagate the reward signal through the thin wall and output a non-optimal policy.

Several elements of this problem deserve more consideration.

Thin walls. Thin walls are a notoriously difficult element of RL problems [Penedones et al., 2018], since the state and state-action value functions are discontinuous around these features, and discontinuous functions are hard to approximate using neural networks. Furthermore, attempting to approximate discontinuous functions can cause unbounded gradient values at the discontinuity, causing challenges in the optimization process itself, ultimately leading to stability issues and divergence.

Note that thin walls in C-space do not necessarily translate to actual thin walls in the environment. Non-holonomic constraints such as the ones introduced by non-slip contacts introduce discontinuities of the C-space which can cause similar issues.

Reward signal. The behavior of RL algorithms is already greatly influenced by the reward signal, but these differences are also relevant when using them in an offline learning setting. First, note that setting a negative reward to hitting walls has an impact on the behavior of the agent because it changes the exploration-exploitation trade-off. As observed in Fig. 4.1, DDPG converges towards a sub-optimal policy avoiding the negative reward of hitting the walls. Removing this penalty can change the learning dynamics of the environment.

Collision semantics. What happens when the agent hits the wall can seem obvious because we are used to the physical world, but in a discrete-time simulation, several implementations are equally applicable. The first option is to prevent collisions by removing the invalid actions from the set of actions from which to choose from. This is often done in environments with finite state and actions, but is harder when the actor is defined as continuous policy.

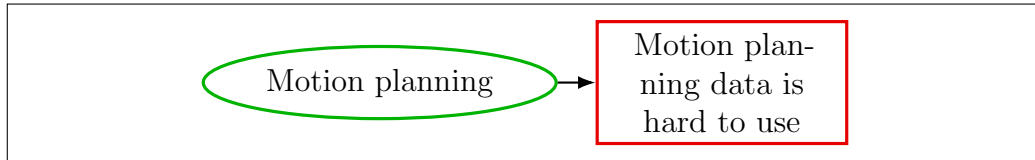
Another option is to accept the transition, but prevent any motion by setting $s' = s$. With a deterministic policy π that selects this action a in state s , the state-action value of this case is $Q^\pi(s, a) = \frac{r}{1-\gamma}$ where r is the reward of this transition.

Finally, the next-state s' can be chosen by traveling in the direction given by a until reaching the wall. This is a tricky case because s' will be exactly on the wall, and in the MDP setting no information is stored about the origin of the agent that resulted in this ambiguous state. In a simulated environment with floating-point imprecision, the agent may end up crossing the wall.

4.8 Conclusion

In this chapter, we showed experimentally that pre-loading exploration data in the replay buffer of an off-policy RL algorithm allowed the critic to reflect the

location of the reward, but failed to propagate it according to the transition rules of the environment. Then, we showed that treating the exploration data as a disconnected set of transitions caused thin walls to be virtually invisible, highlighting the need to retain some connectivity data of the exploration tree for the RL phase.



In the next chapter, we present a solution that exploits the connectivity data of the exploration in the form of a single valid trajectory from the starting position to the reward.

Exploiting exploration data as a training curriculum: backtracking

In the previous chapter, we concluded that even though Motion Planning (MP) algorithms may be able to find the sparse reward in an environment, using the resulting exploration tree is not straightforward. More specifically, splitting the exploration tree in a set of randomly shuffled transitions from which to learn incrementally causes valuable connectivity data to be lost. For instance, in mazes with thin walls this connectivity data is required to propagate the reward.

In this chapter, we contribute a novel algorithm called Plan, Backplay (PB), inspired by Backplay [Resnick et al., 2018]. A single state-space path linking the starting state to the reward is extracted from the exploration tree, and used as a curriculum for training a Reinforcement Learning (RL) algorithm on increasingly difficult portions of the full environment.

In Section 5.2, we present similar approaches to ours, and explain their similarities and differences. In Section 5.3, we present the actual PB algorithm and in Section 5.4 we describe a set of maze environments which we use to test the algorithm. Section 5.6 presents our results and an interpretation.

5.1 Introduction

RL algorithms have been used successfully to optimize policies for both discrete and continuous control problems with high dimensionality [Mnih et al., 2013; Lillicrap et al., 2015], but fall short when trying to solve difficult exploration problems [van Hasselt et al., 2018; Achiam et al., 2019; Schaul et al., 2015]. On the other hand, MP algorithms such as Rapidly-exploring Random Tree (RRT) [Lavalle, 1998] are able to efficiently explore in large

cluttered environments but, instead of trained policies, they output trajectories that cannot be used directly for closed loop control.

In this chapter, we consider environments which present a hard exploration problem with a sparse reward. In this context, a *good trajectory* is one that reaches a state with a positive reward, and we say that an environment is *solved* when a controller is able to reliably reach a rewarded state. We illustrate our approach with 2D continuous action mazes.

If one wants to obtain closed loop controllers for hard exploration problems, a simple approach is to first use an MP algorithm to find a single good trajectory τ , then optimize and robustify it using RL. However, using τ as a stepping stone for an RL algorithm is not straightforward. In this chapter, we propose PB, an approach that fits the framework of Go-Explore [Ecoffet et al., 2019], but uses a modified version of the Backplay algorithm [Resnick et al., 2018].

PB has two successive phases. First, the environment is explored until a single good trajectory is found. Then this trajectory is used to create a curriculum for training Deep Deterministic Policy Gradient (DDPG). More precisely, PB progressively increases the difficulty through a backplay process which gradually moves the starting point of the environment backwards along the trajectory resulting from exploration.

The main contribution of this chapter is a variant of the Backplay algorithm, which may seem very similar to the original, but actually lays the groundwork for the integration of skill chaining presented in Chapter 7.

5.2 Related work

To our knowledge, the closest approach to ours is the Go-Explore framework [Ecoffet et al., 2019], but in contrast to PB, Go-Explore is applied to discrete problems such as Atari benchmarks. In a first phase, a single valid trajectory is computed using an ad hoc exploration algorithm. In a second phase, a learning from demonstration (LfD) algorithm is used to imitate and improve upon this trajectory. Go-Explore uses Backplay [Resnick et al., 2018; Salimans and Chen, 2018] as the Learning from Demonstration (LfD) algorithm, with Proximal Policy Optimization (PPO) [Schulman et al., 2017] as policy optimization method.

Some approaches are similar in the sense that they learn increasingly large portions of the problem by moving the initial state backwards, but they are *undirected* in the sense that they attempt to reach the goal from *any* starting position, until they happen to reach the original initial state of the environment. These approaches include Recall Traces [Goyal et al., 2019]

in which a backtracking model is used to grow the tree backwards, Reverse Curriculum Generation [Florensa et al., 2018] in which the starting points are sampled using forward iteration of the environment and BaRC [Ivanovic et al., 2019] in which the priors are computed using an approximate physical model of the environment.

Moreire et al. [2020] present an approach that is similar to ours, and also fits the framework of Go-Explore. In phase 1, they use a guided variant of RRT, and in phase 2 they use a LfD algorithm based on Trust Region Policy Optimization (TRPO). Similarly, PB follows the same two phases as Go-Explore, with changes to both phases. In the first phase, we substituted the exploration process of Go-Explore for Ex, which we presented in Section 3.3, and in the second phase, we replaced Backplay with a variant that we found easier to control, and replaced the underlying RL algorithm PPO with the more sample efficient DDPG.

The Backplay algorithm in PB is a deterministic variant of the one proposed by Resnick et al. [2018]. In the original Backplay algorithm, the starting point of each policy is chosen randomly from a subset of the trajectory, but in our variant the starting point is deterministic: the last state of the trajectory is used until the performance of DDPG converges (more details are presented in Section 5.3), then the previous state is chosen, and so on until the full trajectory has been exploited.

5.3 Backtracking algorithm

Figure 5.1(a) describes PB. The algorithm is split in two successive phases, mirroring the Go-Explore framework. In a first phase, the environment is incrementally explored using Ex (described in Section 3.3) until a single rewarded state is found. In a second phase, a single trajectory provides a list of starting points, that are used to train a single DDPG instance on increasingly difficult portions of the full environment, until the agent is able to reliably reach the target when starting from the original starting point of the environment.

The Backplay algorithm was originally proposed in [Resnick et al., 2018]. The main difference between the original algorithm and our variant is that the original Backplay uses a sliding window along the trajectory τ , in which the starting state is chosen randomly for each environment reset. The sliding of the window is controlled by hyperparameters.

In contrast, our version of the algorithm has a deterministic starting point, therefore the stochasticity during training relies solely on the action noise added in the training process. This would be equivalent to setting the window

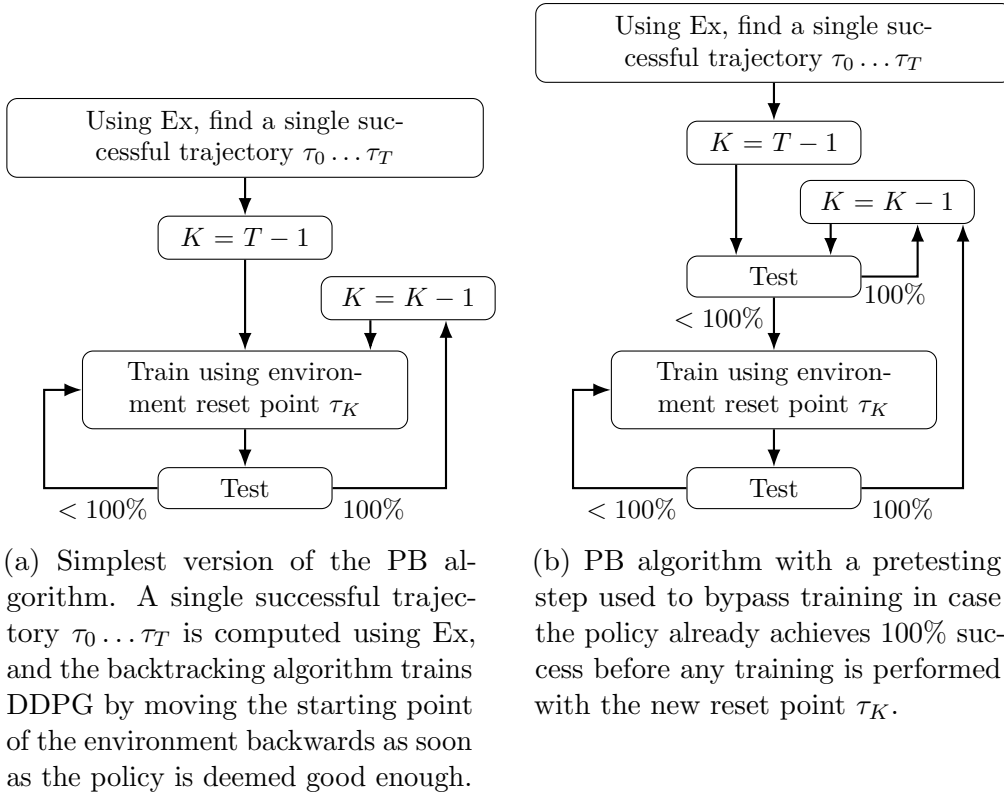


Figure 5.1: Variants of the PB algorithm.

size of the original Backplay algorithm to 1. However, the progress of the reset point is controlled by the performance of the policy instead of being preprogrammed: the reset point is changed only when the policy reaches a performance of 100%, based on periodic evaluations.

However, training the agent for each and every state in τ can be time-consuming and reduce sample efficiency, especially if the replay trajectory is very dense (which is often the case with trajectories that are produced by Ex). A simple optimization consists in adding a pretesting step before training begins. If the policy already achieves 100% success with the new starting point before the training begins, then training is skipped altogether. This optimization is presented in Fig. 5.1(b).

5.4 Experimental Setup

Our experiments are conducted in maze environments of various sizes. A maze of size N is described using the following Markov Decision Process

(MDP) with no terminal states:

$$\begin{aligned}
S &= [0, N] \times [0, N] \\
A &= [-0.1, 0.1] \times [-0.1, 0.1] \\
\text{step}(s, a) &= \begin{cases} s & \text{if } [s, s + a] \text{ intersects a wall} \\ s + a & \text{otherwise.} \end{cases} \\
R(s, a, s') &= \mathbb{1}_{\|s' - \text{target}\| < 0.2}
\end{aligned}$$

Many mazes of size 3 are trivial, therefore we always used a hard “S”-shape 3×3 maze to lower the variance of the results. Mazes of size 2 are all equivalent, and mazes of size 4 were generated independently for each test using a depth-first recursive backtracker maze generation algorithm [Github, 2018a]. Walls have a thickness of 0.1, and examples of generated mazes can be found on page 107.

The target position is $(N - .5, N - .5)$ when $N > 2$. In mazes of size 2, the target position is $(.5, 1.5)$.

We used a standard DDPG implementation [OpenAI, 2018] with default parameters. Each training session is limited to 50 episodes and an ϵ -greedy noise process is applied during rollouts: for some $0 < \epsilon < 1$, with probability ϵ the action is randomly sampled (and the actor is ignored), and with probability $1 - \epsilon$ no noise is used and the raw action is returned. In our tests, we used $\epsilon = 0.1$ (the benefits of ϵ -greedy noise over other noise processes is discussed in Section 6.2.2 on page 76).

5.5 Choice of discount factor γ

The discount factor γ is neither part of the environment nor the learning algorithm, but defines the optimality criterion. It controls the decay that is applied when evaluating the contributions of future rewards to a present choice. In our experiments, we tested two values of γ , that are $\gamma = 0.9$ and $\gamma = 0.99$.

DDPG uses a neural network Q in order to estimate the state-action value function $Q^\pi(s, a)$ of the current policy π . In the case of deterministic environments, the state-action value function is recursively defined as $Q^\pi(s, a) = R(s, a, s') + \gamma Q^\pi(s', \pi(s'))$, where $s' = \text{step}(s, a)$.

Discount factor γ too low: the policy becomes short-sighted. Reaching a sparse reward of value 1 after n steps with no reward carries a discounted value of γ^n . This implies that rewards that are reached only after many steps

have very little impact on the shape of Q . For instance, with $\gamma = 0.9$ and $n = 50$, $\gamma^n \approx 0.05$. This effectively reduces the magnitude of the training signal used by the actor update of DDPG, reducing the speed of actor updates the farther from the reward.

With this consideration, it seems that choosing γ very close to 1 solves the problem of exponential decay of the training signal. However, high γ values present their own challenges.

Discount factor γ too high: over-estimations of Q become slow to correct. The state-action critic approximator Q used by DDPG is trained on (s, a, r, s') tuples stored in an experience replay buffer, but as with any continuous approximator, it generalizes the training data to nearby (s, a) couples. In environments with positive rewards, Q can over-estimate the value of states: for instance in maze environments, the learned value can be generalized incorrectly and “leak” through walls. This mechanism is usually counter-balanced by the fact that over-estimated $Q(s, a)$ values can then be lowered. For instance, in our maze environments, hitting a wall generates a training tuple with $s' = s$ and $r = 0$. The update rule of DDPG applied to this tuple when the current policy is to go towards the wall ($\pi(s') = \pi(s) = a$) yields: $Q(s, a) \leftarrow Q(s, a)(1 + c(\gamma - 1))$ where c is the critic learning rate. Therefore, the closer γ is to 1, the slower over-estimated values will be corrected towards their actual value under the current policy which is 0.

In smaller mazes, our experiments show that reducing γ increases the performance of PB but not DDPG (Fig. 5.2). Since γ controls the rate at which the reward signal decays with distance, in small maze environments where the number of steps required to reach the reward is small, lower γ values give a better gradient in the range of relevant time horizons. Furthermore, as seen earlier this may allow for better correction of over-estimated critic values.

5.6 Results on 2D mazes and analysis

Figure 5.3 shows the performance of PB and vanilla DDPG on various 2D mazes. Both algorithms were run on the same mazes with different random seeds, and a run was considered successful if the RL algorithm was able to produce a policy reaching the target area from the environment reset point at least once, within a limit of 1M environment interactions. The environment interactions required by PB during the exploration phase are included in this count.

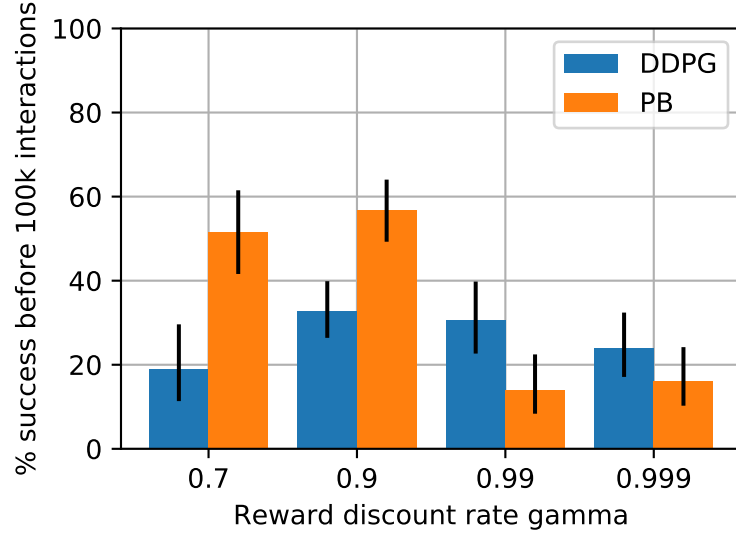


Figure 5.2: Success rate of DDPG and PB on 2×2 mazes depending on γ . Error bars are computed using Wilson score intervals [Wilson, 1927]. From left to right, $N = 117, 106, 108, 93, 183, 169, 69, 93$.

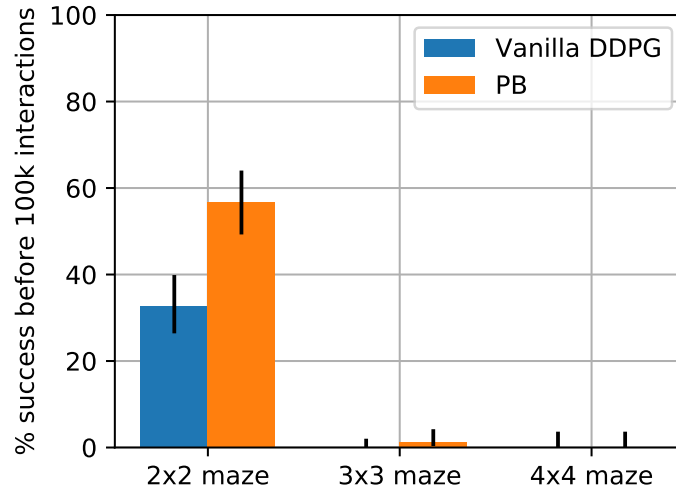


Figure 5.3: Success rate of PB on 2D mazes, when compared to DDPG. Error bars are computed using Wilson error intervals. From left to right, $N = 183, 169, 185, 168, 101, 101$.

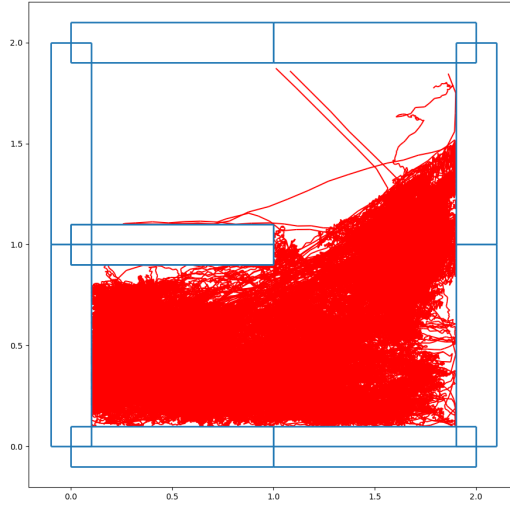
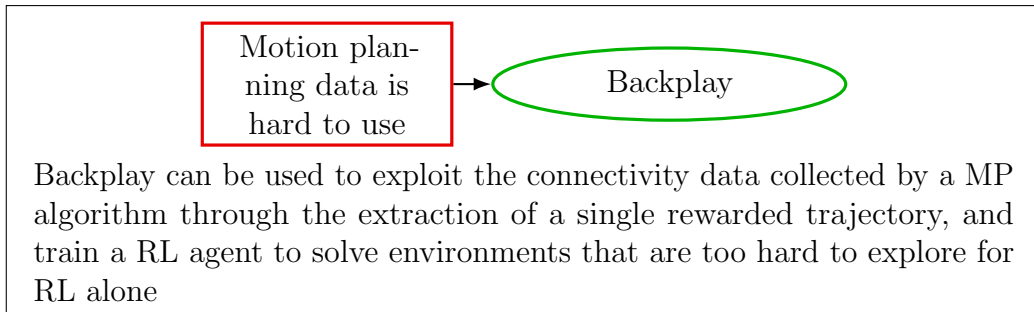


Figure 5.4: Example run of vanilla DDPG on a 2x2 maze for 100k steps. The sparse reward, which was placed in the top-left hand corner, was not found.

Figures 5.3 and 5.4 show that the exploration pattern of DDPG is often unable to find sparse rewards even in the simplest environments, as discussed in the introduction.

The results presented in Fig. 5.3 show that in small mazes, using Backplay drastically increases the performance of DDPG.



However, we can observe that:

- Even in very simple environments such as 2×2 mazes, PB regularly fails.
- The relative advantage of PB disappears in more complex mazes, hinting that other factors may be at play.

In the following chapter, we explore a fundamental problem with the DDPG algorithm that affects both DDPG and PB.

The problem of deterministic policy gradients in deterministic environments with sparse rewards

In the previous chapter, testing backtracking on simple maze environments revealed an unexpected problem in the Reinforcement Learning (RL) phase of the Plan, Backplay (PB) algorithm: even in ideal conditions with the environment difficulty being slowly increased, Deep Deterministic Policy Gradient (DDPG) regularly fails to train a policy on 2×2 mazes.

In this chapter, we simplify this problem to a 1D environment with the starting point only one step away from the reward, meaning that a single well-chosen action is enough to find the reward. Of course, without a heuristic and without any reward gradient, finding such an action can still be hard in the general case although in 1D we could still expect DDPG to succeed reliably. In order to maximize the chances of success, an ϵ -greedy noise process is applied during rollouts, with $\epsilon = 0.1$, and the environment is reset often. With these settings, we observe that the reward is indeed found by rollouts but, surprisingly, the RL algorithm still fails to train an actor to follow this very simple policy.

In Section 6.2, we isolate this problem and show that in this environment, there exists a deadlock cycle that blocks training if the reward is found very quickly. In Section 6.4, we generalize this deadlock to all deterministic environments with continuous actions and sparse rewards. In Section 6.6, we propose variations of the DDPG algorithm that are immune to this specific problem.

This chapter requires a good understanding of DDPG, therefore reading the overview of DDPG presented in Section 1.2.5 is recommended. The target networks of DDPG are mostly useful to stabilize function approximation

when learning the critic and actor networks. Since we are mostly interested in convergence phenomena, we ignore them in most of the discussions presented in this chapter. However, all our tests are still conducted with these target networks.

6.1 Related work

Issues when combining RL with function approximation have been studied for a long time [Baird and Klopff, 1993; Boyan and Moore, 1995; Tsitsiklis and Van Roy, 1997]. In particular, it is well known that deep RL algorithms can diverge when they meet three conditions coined as the “deadly triad” [Sutton and Barto, 2018b], that is when they use (1) function approximation, (2) bootstrapping updates and (3) off-policy learning. However, these questions are mostly studied in the continuous state, discrete action case. For instance, several recent papers have studied the mechanism of this instability using Deep Q-Network (DQN) [Mnih et al., 2013]. In this context, four failure modes have been identified from a theoretical point of view by considering the effect of a linear approximation of the DQN updates and by identifying conditions under which the approximate updates of the critic represent contraction mappings for some distance over Q-functions [Achiam et al., 2019]. Meanwhile, van Hasselt et al. [2018] show that, due to its stabilizing heuristics, DQN does not diverge much in practice when applied to the ATARI domain.

In contrast to these papers, here we study a failure mode specific to continuous action actor-critic algorithms. It hinges on the fact that one cannot take the maximum over actions, and must rely on the actor as a proxy for providing the optimal action instead. Therefore, the failure mode identified in this chapter cannot be reduced to any of the ones that affect DQN. Besides, the theoretical derivations provided in the appendices show that the failure mode we are investigating does not depend on function approximation errors, thus it cannot be directly related to the deadly triad.

More related to our work, several papers have studied failure to gather rewarded experience from the environment due to poor exploration [Colas et al., 2018; Fortunato et al., 2017; Plappert et al., 2017], but we go beyond this issue by studying a case where the reward is actually found but not properly exploited. Finally, like us, Fujimoto et al. [2018a] study a failure mode which is specific to DDPG-like algorithms, but the studied failure mode is different. They show under a batch learning regime that DDPG suffers from an *extrapolation error* phenomenon, whereas we are in the more standard incremental learning setting and focus on a deadlock resulting from the shape of the Q-function in the sparse reward case.

6.2 A new failure mode

In this section, we introduce a simple 1D environment called 1D-TOY. We then show that despite its simplicity DDPG occasionally fails to solve the 1D-TOY environment. We then show that these failures occur when the reward is not found early enough causing the learning process to get stuck. Besides, we show that the initial actor can be significantly modified in the initial stages before finding the first reward. We explain how the combination of these phenomena can result into a deadlock situation on 1D-TOY.

6.2.1 The 1D-Toy environment

In this section, we introduce a simplistic environment which we call 1D-TOY. It is a one-dimensional, discrete-time, continuous state and action problem, depicted in Fig. 6.1.

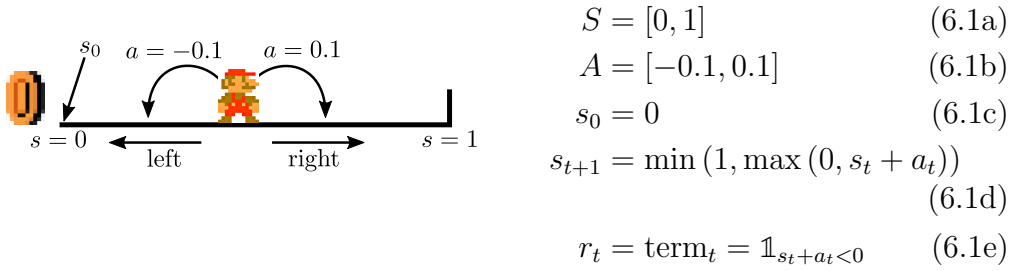
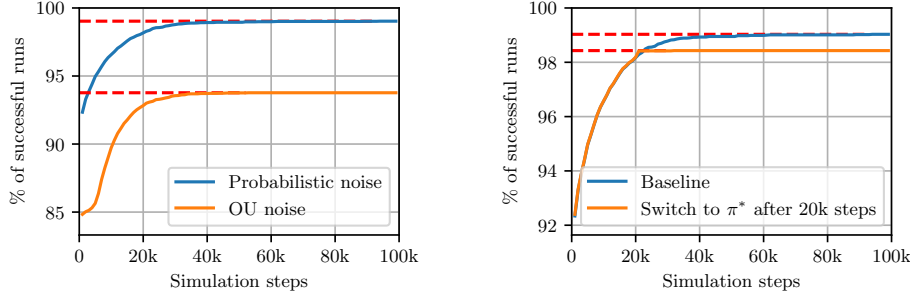


Figure 6.1: The 1D-TOY environment

Appendix Section A.4 studies a few properties of 1D-TOY. Finding analytical solutions for this discrete-time continuous-state and action Markov Decision Process (MDP) is an interesting brain teaser, but we do not use any of these findings in this chapter.

6.2.2 Residual failure to converge using different noise processes.

We start by running DDPG on the 1D-TOY environment. This environment is trivial as one infinitesimal step to the left is enough to obtain the reward, end the episode and succeed, thus we might expect a quick 100% success. The behavior of the agent during training is driven by the current policy trained by DDPG to which a noise process is applied to introduce stochasticity in the



(a) Success rate of DDPG with OU and ϵ -greedy noise. Even with ϵ -greedy noise, DDPG fails on about 1% of the seeds.
(b) Comparison between DDPG with ϵ -greedy noise and a variant in which the behavior policy is set to the optimal policy π^* after 20k steps.

Figure 6.2: Success rate of variants of DDPG on 1D-TOY over learning steps, averaged over 10k seeds.

collected data. The first attempt using an Ornstein-Uhlenbeck (OU) noise process shows that DDPG succeeds in only 94% of cases, see Fig. 6.2(a).

These failures might come from an exploration problem. Indeed, at the start of each episode the OU noise process is reset to zero and gives little noise in the first steps of the episode. In order to remove this potential source of failure, we replace the OU noise process with an ϵ -greedy strategy (also called *probabilistic noise*). For some $0 < \epsilon < 1$, with probability ϵ , the action is randomly sampled (and the actor is ignored), and with probability $1 - \epsilon$ no noise is used and the raw action is returned. In our tests, we used $\epsilon = 0.1$. This guarantees at least a 5% chance of success at the first step of each episode, for any policy. Nevertheless, Fig. 6.2(a) shows that even with ϵ -greedy noise, about 1% of seeds still fail to converge to a successful policy in 1D-TOY, even after 100k training steps. All the following tests are performed using ϵ -greedy noise.

We now focus on these failures. On all failing seeds, we observe that the actor has converged to a saturated policy that always goes to the right ($\forall s, \pi(s) = 0.1$). However, some mini-batch samples have non-zero rewards because the agent still occasionally moves to the left, due to the ϵ -greedy noise applied during rollouts. The expected fraction of non-zero rewards is slightly more than 0.1%¹. Figure 6.3(a) shows the occurrence of rewards in mini-batches taken from the replay buffer when training DDPG on 1D-TOY.

¹10% of steps are governed by ϵ -greedy noise, of which at least 2% are the first episode step, of which 50% are steps going to the left and leading to the reward.

After each rollout (episode) of n steps, the critic and actor networks are trained n times on mini-batches of size 100. So for instance, a failed episode of size 50 is followed by a training on a total of 5000 samples, out of which we expect more than 5 in average are rewarded transitions.

The constant presence of rewarded transitions in the mini-batches suggests that the failures of DDPG on this environment are not due to insufficient exploration by the behavior policy.

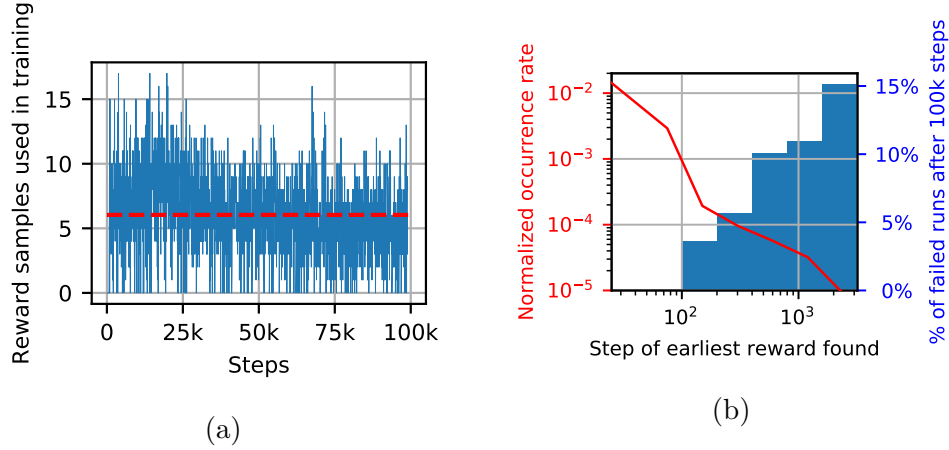


Figure 6.3: (a) Number of rewards found in mini-batches during training. After a rollout of n steps, the actor and critic are both trained on n mini-batches of size 100. The red dotted line indicates an average of 6.03 rewarded transitions present in these n mini-batches. (b) In red, normalized probability of finding the earliest reward at this step. In blue, for each earliest reward bin, fraction of these episodes that fail to converge to a good actor after 100k steps. Note that when the reward is found after one or two episodes, the convergence to a successful actor is certain.

6.3 Correlation between finding the reward early and finding the optimal policy.

We have shown that DDPG can get stuck in 1D-TOY despite finding the reward regularly. Now we show that when DDPG finds the reward early in the training session, it is also more successful in converging to the optimal policy. On the other hand, when the first reward is found late, the learning process more often gets stuck with a sub-optimal policy.

From Fig. 6.3(b), the early steps appear to have a high impact on the outcome of training. For instance, if the reward is found in the first 50 steps by the actor noise (which happens in 63% of cases), then the success rate of DDPG is 100%. However, if the reward is first found after more than 50 steps, then the success rate drops to 96%. Figure 6.3(b) shows that finding the reward later results in lower success rates, down to 87% for runs in which the reward was not found in the first 1600 steps. Therefore, we claim that there exists a critical time frame for finding the reward in the very early stages of training.

6.3.1 Spontaneous actor drift

At the beginning of each training session, the actor and critic of DDPG are initialized to represent respectively close-to-zero state-action values and close-to-zero actions. Besides, as long as the agent does not find a reward, it does not benefit from any utility gradient. Thus we might expect that the actor and critic remain constant until the first reward is found. Actually, we show that even in the absence of reward, training the actor and critic triggers non-negligible updates that cause the actor to reach a saturated state very quickly.

To investigate this, we use a variant of 1D-TOY called DRIFT where the only difference is that no rewarded or terminal transitions are present in the environment. We also use a stripped-down version of DDPG, removing rollouts and using random sampling of states and actions as mini-batches for training.

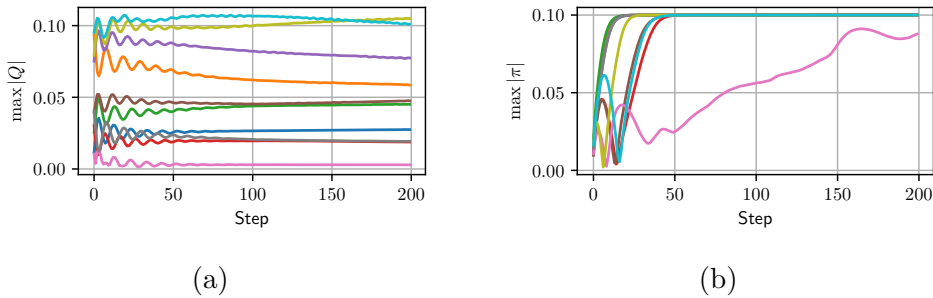


Figure 6.4: Drift of (a) $\max |Q|$ and (b) $\max |\pi|$ in the DRIFT environment, for 10 different seeds. In the absence of reward, the critic oscillates briefly before stabilizing. However, the actor very quickly reaches a saturated state, at either $\forall s, \pi(s) = 0.1$ or -0.1 .

Figure 6.4(b) shows that even in the absence of reward, the actor function

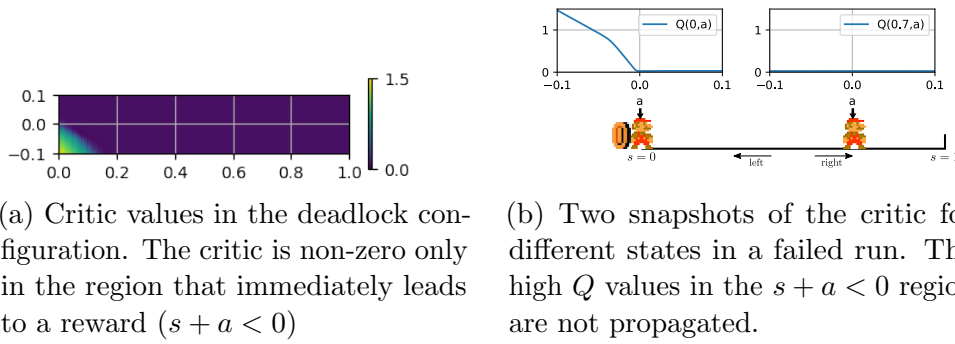


Figure 6.5: Visualization of the critic in a failing run, in which the actor is stuck to $\forall s, \pi(s) = 0.1$.

drifts rapidly (notice the horizontal scale in steps) to a saturated policy, in a number of steps comparable to the “critical time frame” identified above. The critic also has a transitive phase before stabilizing.

In Fig. 6.4(a), the fact that $\max_{s,a} |Q(s, a)|$ can increase in the absence of reward can seem counter-intuitive, since in the loss function presented in Eq. (1.2), $|y_i|$ can never be greater than $\max_{s,a} |Q(s, a)|$. However, it should be noted that the changes made to Q are not local to the mini-batch points, and increasing the value of Q for one input (s, a) may cause its value to increase for other inputs too, which may cause an increase in the global maximum of Q . This phenomenon is at the heart of the over-estimation bias when learning a critic [Fujimoto et al., 2018b], but this bias does not play a key role here.

6.3.2 Explaining the deadlock situation for DDPG on 1D-Toy

Up to now, we have shown that DDPG fails about 1% of times on 1D-TOY, despite the simplicity of this environment. We have now collected the necessary elements to explain the mechanisms of this deadlock in 1D-TOY.

Figure 6.5 shows the value of the critic in a failed run of DDPG on 1D-TOY. We see that the value of the reward is not propagated correctly outside the region in which the reward is found in a single step $\{(s, a) \mid s + a < 0\}$. The key of the deadlock is that once the actor has drifted to $\forall s, \pi(s) = 0.1$, it is updated according to $\nabla_a Q_\theta(s, a)|_{a=\pi_\psi(s)}$ (Eq. (1.5)). Figure 6.5(b) shows that for $a = \pi(s) = 0.1$, this gradient is zero therefore the actor is not updated. Besides, the critic is updated using $y_i = r(s_i, a_i) + \gamma Q(s'_i, \pi(s'_i))$ as a target. Since $Q(s'_i, 0.1)$ is zero, the critic only needs to be non-zero for directly rewarded actions, and for all other samples the target value remains

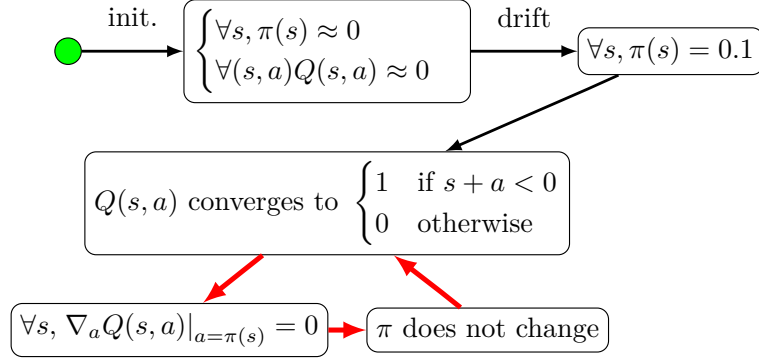


Figure 6.6: Deadlock observed in 1D-TOY, represented as the cycle of red arrows.

zero. In this state, the critic loss given in Eq. (1.2) is minimal, so there is no further update of the critic and no further propagation of the state-action values. The combination of the above two facts clearly results in a deadlock.

Importantly, the constitutive elements of this deadlock do not depend on the batches used to perform the update, and therefore do not depend on the experience selection method. We tested this experimentally by substituting the behavior policy for the optimal policy after 20k training steps. Results are presented in Fig. 6.2(b) and show that, once stuck, even when it is given ideal samples, DDPG stays stuck in the deadlock configuration. This also explains why finding the reward early results in better performance. When the reward is found early enough, $\pi(s_0)$ has not drifted too far, and the gradient of $Q(s_0, a)$ at $a = \pi(s_0)$ drives the actor back into the correct direction.

Note however that even when the actor drifts to the right, DDPG does not always fail. Indeed, because of the function approximation the shape of the critic when finding the reward for the first time varies, and sometimes converges slowly enough for the actor to be updated before the convergence of the critic.

Figure 6.6 summarizes the above process. The entry point is represented using a green dot. First, the actor drifts to $\forall s, \pi(s) = 0.1$, then the critic converges to Q^π which is a piecewise-constant function (Experiment in Fig. 6.5, proof in Theorem 1 in Section 6.3.3), which in turn means that the critic provides no gradient, therefore the actor is not updated (as seen in Eq. (1.5), more details in Theorem 2)².

²Note that Fig. 6.5 shows a critic state which is slightly different from the one presented in Fig. 6.6, due to the limitations of function approximators.

6.3.3 Formal proof of the existence of a deadlock in 1D-Toy

In this section, we prove that there exists a state of DDPG that is a deadlock in the 1D-TOY environment. This proof directly references Fig. 6.6. Let us define two functions Q and π_ψ such that:

$$\forall (s, a), Q(s, a) = \mathbb{1}_{s+a < 0} \text{ and} \quad (6.2)$$

$$\forall s \in S, \pi_\psi(s) = 0.1. \quad (6.3)$$

From now on, we will use notation $\pi := \pi_\psi$.

Theorem 1. (Q, π_ψ) is a fixed point for the critic update.

Proof. The critic update is governed by Eq. (1.2).

Let $(s_i, a_i, r_i, t_i, s'_i)$ be a sample from the replay buffer. The environment dictates that $r_i = t_i = \mathbb{1}_{s_i+a_i < 0}$.

The critic update yields

$$y_i = r_i + \gamma(1 - t_i)Q(s'_i, \pi_\psi(s'_i))$$

$$y_i = r_i + \gamma(1 - t_i)Q(s'_i, 0.1) \quad \text{by Eq. (6.3)}$$

$$y_i = r_i \quad \text{by Eq. (6.2)}$$

$$y_i = \mathbb{1}_{s_i+a_i < 0}$$

$$y_i = Q(s_i, a_i) \quad \text{by Eq. (6.2)}$$

and therefore, for each sample $y_i = Q(s_i, a_i)$, and L is null and minimal. Therefore θ will not be updated during the critic update. \square

Theorem 2. (Q, π_ψ) is a fixed point for the actor update.

Proof. The actor update is governed by Eq. (1.5).

Let $\{(s_i, a_i, r_i, t_i, s'_i)\}$ be a set of samples from the replay buffer. The environment dictates that $\forall i, r_i = t_i = \mathbb{1}_{s_i+a_i < 0}$.

$$\psi \leftarrow \psi + \alpha \sum_i \frac{\partial \pi_\psi(s_i)}{\partial \psi} \nabla_a Q(s_i, a)|_{a=\pi_\psi(s_i)}$$

Since $Q(s_i, a) = \mathbb{1}_{s_i+a < 0}$, $\nabla_a Q(s_i, a) = 0$, so ψ will not be updated during the actor update. \square

In this section, we assume that Q is any function, however in implementations Q is often a parametric neural network Q_θ , which cannot be discontinuous. Effects of this approximation are discussed in Section 6.5.

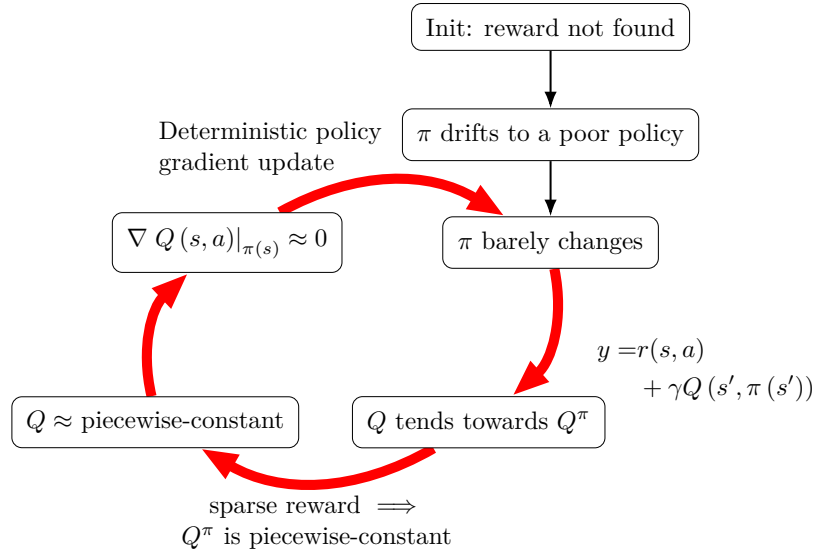


Figure 6.7: A cyclic view of the undesirable convergence process in continuous action actor-critic algorithms, in the deterministic and sparse reward case.

6.4 Generalization to all deterministic Policy Gradient algorithms in deterministic environments with sparse rewards

Our study of 1D-TOY revealed how DDPG can get stuck in this simplistic environment. We now generalize to the broader context of more general continuous action actor critic algorithms, including at least DDPG and Twin Delayed DDPG (TD3), and acting in any deterministic and sparse reward environment. The generalized deadlock mechanism is illustrated in Fig. 6.7 and explained hereafter in the idealized context of perfect approximators.

Entry point: As shown in Section 6.3.1, before the behavior policy finds any reward, training the actor and critic can still trigger non-negligible updates that may cause the actor to quickly reach a poor state and stabilize. This defines our entry point in the process.

Q tends towards Q^π : A first step into the cycle is that, if the critic is updated faster than the policy, the update rule of the critic Q given in Equation Eq. (1.2) makes Q converge to Q^π . This is presented in detail in Section 6.4.1.

Q^π is piecewise-constant: In Section 6.4.2, we then show that, in a deterministic environment with sparse terminal rewards, Q^π is piecewise-constant because $V^\pi(s')$ only depends on two things: the (integer) number of steps required to reach a rewarded state from s' , and the value of this reward state, which is itself piecewise-constant. Note that we can reach the same conclusion with non-terminal rewards, by making the stronger hypothesis on the actor that $\forall s, r(s, \pi(s)) = 0$. Notably, this is the case for the actor $\forall s, \pi(s) = 0.1$ on 1D-TOY.

Q is approximately piecewise-constant and $\nabla_a Q(s, a)|_{a=\pi(s)} \approx 0$: Quite obviously, from Q^π is piecewise-constant and Q tends towards Q^π , we can infer that Q progressively becomes almost piecewise-constant as the cyclic process unfolds. Actually, the Q function is estimated by a function approximator which is never truly discontinuous. The impact of this fact is studied in Section 6.5. However, we can expect Q to have mostly flat gradients since it is trained to match a piecewise-constant function. We can thus infer that, globally, $\nabla_a Q(s, a)|_{a=\pi(s)} \approx 0$. And critically, the gradients in the flat regions far from the discontinuities give little information on how to reach regions of higher values.

π barely changes: DDPG uses the deterministic policy gradient update, as seen in Eq. (6.4). This is an analytical gradient that does not incorporate any stochasticity, because Q is always differentiated exactly at $(s, \pi(s))$. Thus the actor update is stalled, even when the reward is regularly found by the behavior policy. This closes the loop of our process.

$$\psi \leftarrow \psi + \alpha_a \sum_i \frac{\partial \pi_\psi(s_i)^T}{\partial \psi} \nabla_a Q_\theta(s_i, a)|_{a=\pi_\psi(s_i)}. \quad (6.4)$$

6.4.1 Proof of convergence of the critic to Q^π

Notation For a state-action pair s, a , we define s_1 as the result of applying action a at state s in the deterministic environment. For a given policy π , we define a_1 as $\pi(s_1)$. Recursively, for any $i \geq 1$, we define s_{i+1} as the result of applying action a_i to state s_i , and a_i as $\pi(s_i)$.

Definition 1. Let $(s, a) \in S \times A$. If (s, a) is terminal, then we set $N = 0$. Otherwise, we set N to the number of subsequent transitions with policy π to reach a terminal state. Therefore, the transition (s_N, a_N) is always terminal. We generalize by setting $N = \infty$ when no terminal transition is ever reached.

We define the state-action value function of policy π as:

$$Q^\pi(s, a) := r(s, a) + \sum_{i=1}^N \gamma^i r(s_i, a_i)$$

Note that when $N = \infty$, the sum converges under the hypothesis that rewards are bounded and $\gamma < 1$.

If π is fixed, Q is updated regularly via approximate dynamic programming with the Bellman operator for the policy π . Under strong assumptions, or assuming exact dynamic programming, it is possible to prove [Geist and Pietquin, 2011] that the iterated application of this operator converges towards a unique function Q^π , which corresponds to the state-action value function of π as defined above. It is usually done by proving that the Bellman operator is a contraction mapping, and also applies in deterministic cases.

However, when using approximators such as neural nets, no theoretical results of convergence exists, to the best of our knowledge. In this chapter we assume that this convergence is true, and in the experimental results we did not observe any failures to converge towards Q^π . On the contrary, we observe that this convergence occurs, and can be what starts the deadlock cycle studied in Section 6.4.

6.4.2 Proof that Q^π is piecewise-constant

In this section, we show that in deterministic environments with terminal sparse rewards (that is, the only transitions with non-zero reward are also terminal), Q^π is piecewise-constant.

Definition 2. *In this chapter, for $I \subset \mathbb{R}^n$, we say a function $f : I \rightarrow \mathbb{R}$ is piecewise-constant if $\forall x_0 \in I$, either $\nabla f(x_0) = 0$, or f has no gradient at x_0 .*

Theorem 3. *In a deterministic environment with terminal sparse rewards, for any π , Q^π is piecewise-constant.*

Proof. Note that this proof can be trivialized by assuming that around any point where the gradient is defined, there exists a neighborhood in which the function is continuous. In this case, the intermediate value theorem yields an uncountable set of values of the function in this neighborhood, which contradicts the countable number of possible discounted rewards.

The crux of the following proof is that even when no such neighborhood exists, the gradient is either null or non-existent. This behavior is shown in Fig. 6.8.

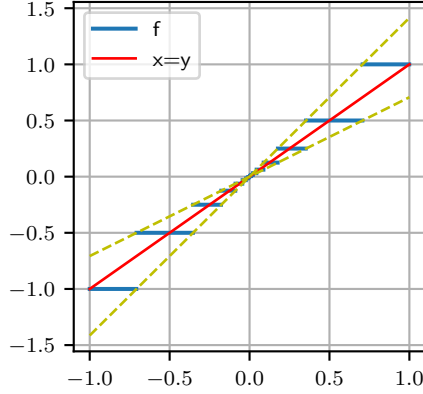


Figure 6.8: Example of a non-continuous function f with values in $\{\pm (\frac{1}{2})^n \mid n \in \mathbb{N}\}$, approximating the identity function. However, this function is not differentiable even at $x = 0$ because the difference quotient does not converge but instead oscillates between two values.

Using the notations of Definition 1 and the theorem hypothesis that rewarded transitions are also terminal, we can write $Q^\pi(s, a)$ as $Q^\pi(s, a) = \begin{cases} r(s, a) & \text{if } N = 0 \\ \gamma^N r(s_N, a_N) & \text{if } N \text{ is finite.} \\ 0 & \text{otherwise.} \end{cases}$

We promote N to a function $S \times A \rightarrow \mathbb{N} \cup \{+\infty\}$, and we define a function $u : S \times A \rightarrow \mathbb{R}$ as $u(s, a) = \begin{cases} r(s, a) & \text{if } N = 0 \\ r(s_{N(s,a)}, a_{N(s,a)}) & \text{if } N > 0 \text{ finite.} \\ 0 & \text{otherwise.} \end{cases}$

Now we have $\forall (s, a) \in S \times A, Q^\pi(s, a) = \gamma^{N(s,a)} u(s, a)$.

Let R be the finite set of possible reward values.

Therefore values of Q^π are in a set $M = \{\gamma^n r \mid n \in \mathbb{N}, r \in R\}$. Let $M^+ = M \cap \mathbb{R}^{+*}$ be the set of positive values of M . Since $R \subset \mathbb{R}$ is finite, we order all non-zero positive possible rewards in increasing order $r_1, r_2, \dots, r_{|R|}$.

Let $M_k^+ = \{\gamma^n r \mid n \in \mathbb{N}, r \in R_k\}$ where $R_k = \{r_1, \dots, r_k\}$.

We prove the following by induction over the number of possible non-zero rewards:

$$H(k) : \exists \nu_k > 0, \forall \delta > 0, \exists \text{ consecutive } b, a \in M_k^+, \delta \nu_k < a - b \text{ and } b < a < \delta$$

Initialization When $k = 1$, $M^+ = \{r_1\gamma^n \mid n \in \mathbb{N}\}$. Let $\nu_1 = \frac{\gamma(1-\gamma)}{2}$, let $\delta > 0$. Let $n = \lfloor \log_\gamma \frac{\delta}{r_1} \rfloor + 1$. We have:

$$\begin{aligned} \log_\gamma \frac{\delta}{r_1} - 1 &< n - 1 \leq \log_\gamma \frac{\delta}{r_1} \\ \log_\gamma \frac{\delta}{r_1} &< n \leq 1 + \log_\gamma \frac{\delta}{r_1} \\ \gamma \frac{\delta}{r_1} &\leq \gamma^n < \frac{\delta}{r_1} \\ \delta\gamma(1-\gamma) &\leq \gamma^n r_1(1-\gamma) < \delta(1-\gamma) \\ \delta\nu_1 &< \gamma^n r_1(1-\gamma) < \delta(1-\gamma) \\ \delta\nu_1 &< r_1\gamma^n - r_1\gamma^{n+1} \text{ and } r_1\gamma^n < \delta \end{aligned}$$

Let $a = r_1\gamma^n \in M^+$ and $b = r_1\gamma^{n+1} \in M^+$. $\delta\nu_1 < a - b$ and $b < a < \delta$ therefore $H(1)$ is verified.

Induction Let $k \geq 1$, and assume $H(k)$ is true. Let ν_k be the ν chosen for $H(k)$. Let $\nu_{k+1} = \frac{\nu_k}{2}$. Let $\delta > 0$. Let b_k, a_k a consecutive pair chosen in M_k^+ such that $\delta\nu_k < a_k - b_k$ and $b_k < a_k < \delta$.

Since R_{k+1} contains only one more element than R_k , which is larger than all elements in R_k , we know that there is either one or zero elements $c \in M_{k+1}^+$ that are strictly between a_k and b_k . If $a_k - c < c - b_k$ then let $a_{k+1} = c$ and $b_{k+1} = b_k$, otherwise $a_{k+1} = a_k$ and $b_{k+1} = c$. If a_k and b_k are still consecutive in M_{k+1}^+ , then $a_{k+1} = a_k$ and $b_{k+1} = b_k$.

This guarantees that $[b_{k+1}, a_{k+1}]$ is at least half as big as $[b_k, a_k]$. Therefore, $\frac{1}{2}(a_k - b_k) < a_{k+1} - b_{k+1}$, which means that $\delta\nu_{k+1} < a_{k+1} - b_{k+1}$ and $b_k < a_k < \delta$.

Therefore $H(k+1)$ is verified.

This also gives the general expression of ν , valid for all k : $\nu = \frac{\gamma(1-\gamma)}{2^{|R|+1}}$.

Main proof Using the result above, we prove that $Q^\pi(s, a)$ cannot have any non-null derivatives.

Trivially, Q^π cannot have a non-null derivative at a point (s, a) where $Q^\pi(s, a) = q_0 \neq 0$. Indeed, there exists a neighborhood of $q_0 \in M$ in which there is a single value.

Let $x_0 = (s, a)$ such that $Q(s, a) = 0$. Let v be a vector of the space $S \times A$. Let $f : \mathbb{R} \rightarrow \mathbb{R}$ be defined as $f(h) = Q^\pi(x_0 + hv)$. In the following, we show that $\frac{f(h)}{|h|}$ cannot converge to a non-null value when $h \rightarrow 0$.

We use the (ϵ, δ) definition of a limit. If f had a non-null derivative l at 0, we would have $\forall \epsilon > 0, \exists \delta > 0, \forall h, |h| < \delta \implies \left| \frac{f(h)}{|h|} - l \right| < \epsilon$.

Instead, we will show the opposite: $\exists \epsilon > 0, \forall \delta > 0, \exists h, |h| < \delta$ and $\left| \frac{f(h)}{|h|} - l \right| \geq \epsilon$.

Using the candidate derivative l and the ν value computed above that only depends on γ and $|R|$, we set $\epsilon = \frac{l\nu}{2}$.

Let $\delta > 0$.

There exists consecutive b, a in M such that $\delta l\nu \leq a - b$ and $b < a < \delta l$.

We set $h = \frac{a+b}{2}$. Note that $\frac{a+b}{2} < \delta l$ therefore $h < \delta$.

$f(h)$ is in M , but hl is the center of the segment $[b, a]$ of consecutive points of M . Therefore, the distance between $f(h)$ and hl is at least $\frac{a-b}{2}$.

$$|f(h) - hl| \geq \frac{a-b}{2} \geq \frac{\delta l\nu}{2}$$

Since $h < \delta$, $\frac{1}{h} > \delta$.

$$\left| \frac{f(h)}{h} - l \right| \geq \frac{l\nu}{2} = \epsilon.$$

□

6.4.3 Consequences of the convergence cycle

As illustrated with the red arrows in Fig. 6.7, the more loops performed in the convergence process, the more the critic tends to be piecewise-constant and the less the actor tends to change. Importantly, this cyclic convergence process is triggered as soon as the changes on the policy drastically slow down or stop. What matters for the final performance is the quality of the policy reached before this convergence loop is triggered. Quite obviously, if the loop is triggered before the policy gets consistently rewarded, the final performance is deemed to be poor.

The key of this undesirable convergence cycle lies in the use of the deterministic policy gradient update given in Eq. (6.4). Actually, rewarded samples found by the exploratory behavior policy β tend to be ignored by the conjunction of two reasons. First, the critic is updated using $Q(s', \pi(s'))$ and not $Q(s, \beta(s))$, thus if π differs too much from β , the values brought by β are not properly propagated. Second, the actor being updated through Eq. (6.4), i.e. using the analytical gradient of the critic with respect to the actions of π , there is no room for considering other actions than that of π . Besides, the actor update involves only the state s of the sample taken from the replay buffer, and not the reward found from this sample $r(s, a)$ or the

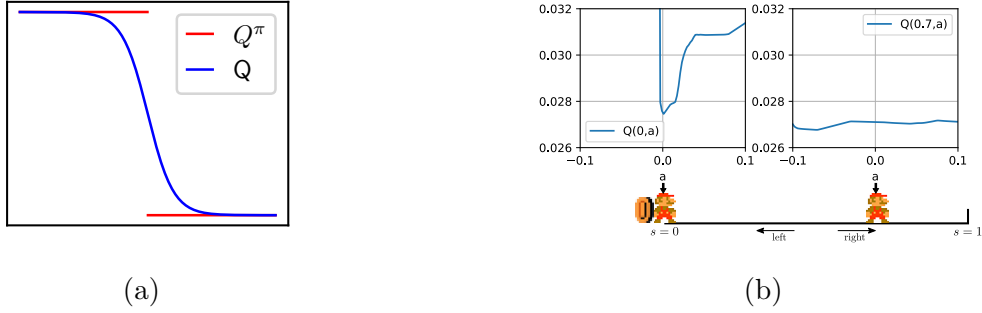


Figure 6.9: (a) Example of a monotonous function approximator. (b) Simply changing the vertical scale of the graphs presented in Fig. 6.5(b) reveals that the function approximator is not perfectly flat, and has many unwanted local extrema. Specifically, continuously moving from $\pi(0) = 0.1$ to $\pi(0) < 0$ requires crossing a significant valley in $Q(0, a)$, while $\pi(0) = 0.1$ is a strong local maximum.

action performed. For each sampled state s , the actor update is intended to make $\pi(s)$ converge to $\operatorname{argmax}_a \pi(s, a)$ but the experience of different actions performed for identical or similar states is only available through $Q(s, \cdot)$, and in DDPG it is only exploited through the gradient of $Q(s, \cdot)$ at $\pi(s)$, so the process can easily get stuck in an area with null gradient, especially if the critic tends towards a piecewise-constant function, which as we have shown happens when the reward is sparse. Besides, since TD3 also updates the actor according to Eq. (6.4), it is susceptible to the same failures as DDPG. More generally, any algorithm that attempts to update a deterministic policy using its gradient computed based on a critic is affected by this cycle.

6.5 Impact of function approximation

We have just explained that when the actor has drifted to an incorrect policy before finding the reward, an undesirable convergence process should result in DDPG getting stuck to this policy. However, in 1D-TOY, we measured that the actor drifts to a policy moving to the right in 50% of cases, but the learning process only fails 1% of times. More generally, despite the issues discussed in this chapter, DDPG has been shown to be efficient in many problems. This better-than-predicted success can be attributed to the impact of function approximation.

Figure 6.9(a) shows a case in which the critic approximates Q^π while keeping a monotonous slope between the current policy value and the reward.

In this case, the actor is correctly updated towards the reward (if it is close enough to the discontinuity). This is the most often observed case, and naturally we expect approximators to smooth out discontinuities in target functions in a monotonous way, which facilitates gradient ascent. However, the critic is updated not only in state-action pairs where $Q^\pi(s, a)$ is positive, but also at points where $Q^\pi(s, a) = 0$, which means that the bottom part of the curve also tends to flatten. As this happens, we can imagine phenomena that are common when trying to approximate discontinuous functions, such as the overshoot observed in Fig. 6.9(b). In this case, the gradient prevents the actor from improving.

6.6 Potential solutions

In Section 6.4, we have shown that actor-critic algorithms such as DDPG and TD3 could not recover from early convergence to a poor policy due to the combination of three factors whose dependence is highlighted in Fig. 6.7: the use of the deterministic policy gradient update, the use of $Q(s', \pi(s'))$ in the critic update, and the attempt to address sparse reward in deterministic environments.

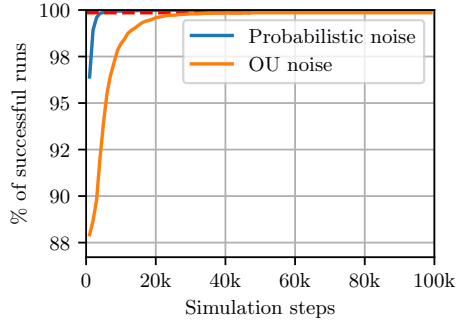
The main purpose of this chapter is to clearly identify the specific problem that leads to failures when using DDPG with deterministic environments and sparse rewards, and we leave a thorough investigation of potential solutions as future work. Nevertheless, in this section we categorize existing or potential solutions to the issue in terms of which of the above factor they remove.

6.6.1 Avoiding sparse rewards

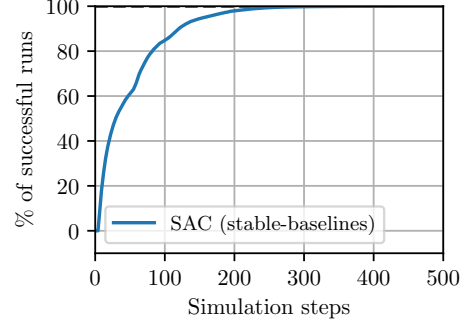
Transforming a sparse reward problem into a dense one can solve the above issue as the critic should not converge to a piecewise-constant function anymore. This can be achieved for instance by using various forms of shaping [Konidaris and Barto, 2006] or by adding auxiliary tasks [Jaderberg et al., 2016; Riedmiller et al., 2018]. We do not further investigate these solutions here, as they are mainly problem-dependent and may introduce bias when the reward transformation results in deceptive gradient or modifies the corresponding optimal policy.

6.6.2 Replacing the policy-based critic update

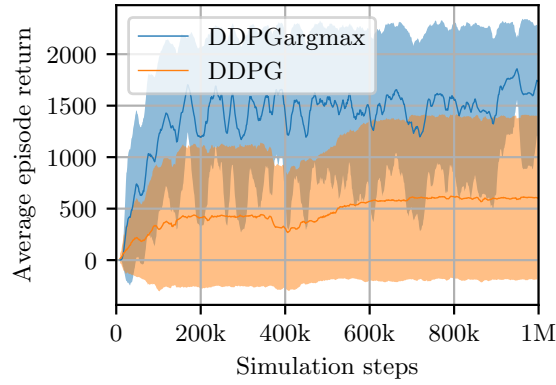
As explained above, if some transition (s, a, s') leading to a reward is found in the replay buffer, the critic update corresponding to this transition uses



(a)



(b)



(c)

Figure 6.10: (a) Applying DDPG-argmax to 1D-TOY. (b) Applying Soft Actor-Critic (SAC) to 1D-TOY. In both cases, the success rate reaches 100% quickly (notice the horizontal scale for SAC). (c) Applying DDPG and DDPG-argmax to a sparse-reward variant of the HALFCHEETAH-V2 environment. Details on the changes made to and HALFCHEETAH-V2 are available in Section 6.7.

$Q(s', \pi(s'))$, therefore not propagating the next state value that the behavior policy may have found. Of course, when using the gradient from the critic, the actor update should tend to update π to reflect the better policy such that $\pi(s') \rightarrow a'$, but the critic does not always provide an adequate gradient as shown before.

If performing a maximum over a continuous action space was possible, using $\max_a Q(s', a)$ instead of $Q(s', \pi(s'))$ would solve the issue. Several works start from this insight. Some methods directly sample the action space and look for such an approximate maximum [Kalashnikov et al., 2018; Simmons-Edler et al., 2019]. To show that this approach can fix the above issue, we applied it to the 1D-TOY environment. We take a straightforward implementation where the policy gradient update in DDPG is replaced by sampling 100 different actions, finding the argmax over these actions of $Q(s, a)$, and regressing the actor towards the best action we found. We call the resulting algorithm DDPG-argmax, and more details are available in Section 6.6.2. Results are shown in Fig. 6.10(a), in which we see that the success rate quickly reaches 100%.

Description of DDPG-argmax Instead of relying on the differentiation of $Q_\theta(s_i, \pi_\psi(s_i))$ to update ψ in order to maximize $Q(s, \pi(s))$, we begin by selecting a set of N potential actions $(b_j)_{0 \leq j < N}$. Then, we compute $Q_\theta(s_i, b_j)$ for each sample s_i and each potential action b_j , and for each sample s_i we find the best potential action $c_i = b_{\arg\max_j Q_\theta(s_i, b_j)}$. Finally, we regress $\pi_\psi(s_i)$ towards the goal c_i . This process is summarized by the following optimization problem, in which $\text{Unif}(A)$ stands for uniform sampling in A :

$$\begin{cases} (b_j)_{0 \leq j < N} \sim \text{Unif}(A) \\ c_i = b_{\arg\max_j Q_\theta(s_i, b_j)} \\ \text{minimize } \sum_i (\pi_\psi(s_i) - c_i)^2 \text{ w.r.t. } \psi \end{cases}.$$

Quite obviously, even if sampling can provide a good enough baseline for simple enough benchmarks, these methods do not scale well to large actions spaces. Many improvements to this can be imagined by changing the way the action space is sampled, such as including $\pi(s)$ in the samples, to prevent picking a worse action than the one provided by the actor, sampling preferentially around $\pi(s)$, or around $\pi(s + \epsilon)$, or just using actions taken from the replay buffer.

Interestingly, using a stochastic actor such as in the Soft Actor Critic (SAC) algorithm [Haarnoja et al., 2018] can be considered as sampling preferentially around $\pi(s + \epsilon)$ where ϵ is driven by the entropy regularization term. In

Fig. 6.10(b), we show that SAC also immediately solves 1D-TOY.

Another approach relies on representing the critic as the V function rather than the Q function. The same way $\pi(s)$ tends to approximate $\operatorname{argmax}_a Q(s, a)$, V tends to approximate $\max_a Q(s, a)$, and is updated when finding a transition that raises the value of a state. Using V , performing a maximum in the critic update is not necessary anymore. The prototypical actor-critic algorithm using a model of V as a critic is Continuous Actor-Critic Learning Automata (CACLA) [Van Hasselt and Wiering, 2007]. However, approximating V with neural networks can prove more unstable than approximating Q , as function approximation can be sensitive to the discontinuities resulting from the implicit maximization over Q values.

6.6.3 Replacing the deterministic policy gradient update

Instead of relying on the deterministic policy gradient update, one can rely on a stochastic policy to perform a different actor update. This is the case of SAC, as mentioned just above. Because SAC does not use $Q(s', \pi(s'))$ in its update rule, it does not suffer from the undesirable convergence process described here.

Another solution consists in completely replacing the actor update mechanism, using regression to update $\pi(s)$ towards any action better than the current one. This could be achieved by updating the actor and the critic simultaneously: when sampling a higher-than-expected critic value $y_i > Q(s_i, a_i)$, one may update $\pi(s_i)$ towards a_i using:

$$L_\psi = \sum_i \delta_{y_i > Q(s_i, \pi(s_i))} (\pi(s_i) - a_i). \quad (6.5)$$

This is similar to the behavior of CACLA, as analyzed in [Zimmer and Weng, 2019].

6.7 Experiments on larger benchmarks

In order to test the relevance of using DDPG-argmax on larger benchmarks, we constructed a sparse reward version of HALF-CHEETAH-v2 [Github, 2018b].

HALF-CHEETAH-v2 was modified by generating a step reward of 2 when the x component of the speed of the cheetah is more than 2. We also removed the control penalty. Since the maximum episode duration is 1000, the maximum possible reward in this modified environment is 2000.

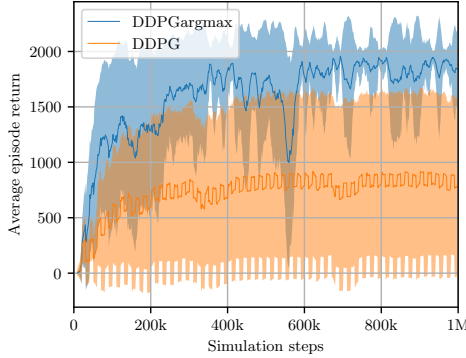


Figure 6.11: Performance of DDPG and DDPG-argmax on a sparse variant of HALF-CHEETAH-v2. To ensure exploration of the state space is not a problem, the policy is replaced with a good pre-trained policy for one episode every 20 episodes.

In both cases, the actor noise uses the default implementation of the Spinup implementation of DDPG, which is an added uniform noise with an amplitude of 0.1.

Running DDPG and DDPG-argmax on this environment yields the results shown in Fig. 6.10(c). Experiments on HALF-CHEETAH-v2 have been conducted using six different seeds, and the main curves are smoothed using a moving average covering 10 episodes (10k steps), and the shaded area represents the average plus or minus one standard deviation.

On HALF-CHEETAH-v2, both DDPG and DDPG-argmax are able to find rewards despite its sparsity. However, DDPG-argmax outperforms DDPG in this environment. Since the only difference between these algorithms is the actor update, we conclude that even in complex environments, the actor update is the main weakness of DDPG. We have shown that replacing it with a brute-force update improves performance dramatically, and further research aiming to improve the performance of deterministic actor-critic algorithms in environments with sparse rewards should concentrate on improving the actor update rule.

Figure 6.10(c) shows that DDPG is able to find the reward without the help of any exploration except the uniform noise built in the algorithm itself. However, to prove that state-space exploration is not the issue here, we constructed a variant in which the current actor is backed up and replaced with a pre-trained good actor every 20 episodes. This variant achieves episode returns above 1950 (as a reminder, the maximum episode return is 2000). In the next episode, the backed up policy is restored. This guarantees that the

replay buffer always contains all the transitions necessary to learn a good policy. We call this technique *priming*.

Results of this variant are presented in Fig. 6.11. Notice that DDPG performs much better than without priming, but the performance of DDPG-argmax is unchanged. However, DDPG still fails to completely solve the environment, proving that even when state-space exploration is made trivial, DDPG underperforms on sparse-reward environments due to its poor actor update.

However, with higher-dimensional and more complex environments, the analysis becomes more difficult and other failure modes such as the ones related to the deadly triad, the extrapolation error or the over-estimation bias might come into play, so it becomes harder to quantitatively analyze the impact of the phenomenon we are focusing on. On one hand, this point showcases the importance of using very elementary benchmarks in order to study the different failure modes in isolation. On the other hand, trying to sort out and quantify the impact of the different failure modes in more complex environments is our main objective for future work.

6.8 Conclusion

In this chapter, we were able to demonstrate that Policy Gradient (PG) algorithms such as DDPG can enter a deadlock state in any deterministic sparse-reward problem, and that this deadlock occasionally occurs even in trivial environments such as 1D-TOY.

In particular, we showed that this deadlock originates in the Deterministic Policy Gradient equation used by DDPG to train its policy efficiently using gradient descent, and presented a variant of DDPG which bypasses this problem at the cost of computational complexity. Despite its complexity, we demonstrated that this variant is able to outperform DDPG on simple tasks such as 1D-TOY, but also on environments that have a higher-dimension action space such as a sparse-reward variant of HALF-CHEETAH-V2.

However, in more complex environments, other well-studied failure modes become prevalent and the deadlock we identified cannot explain the poor behavior of DDPG. In the following chapter, we present an extension of the PB algorithm that allows it to bypass both the problem presented in this chapter and other failure modes of DDPG, and solve complex environments reliably.

Going one step further: backtracking with skill chaining

In an ideal world, Plan, Backplay (PB) alone (described in Chapter 5) would be able to bypass the weaknesses of Reinforcement Learning (RL) algorithms with respect to exploration. However, as demonstrated in Chapter 6, continuous-control RL algorithms struggle in environments with sparse rewards even when trained using PB. Solutions have been proposed, but do not scale well with the dimensionality of the action space.

In this chapter, we propose an extension of the PB algorithm, which we call Plan, Backplay, Chain Skills (PBCS).

In PBCS, skill chaining is integrated in the second phase of the algorithm: robustification. In this phase, a policy is trained to reliably reach the reward by moving the starting point of the environment backwards along a trajectory. In this context, skill chaining is used as a fail-safe, a last-resort to be used when the training performance decreases below a set threshold. More precisely, we use the fact that even if Backplay eventually fails, it is still able to solve some subset of the problem. Therefore, a partial policy is saved, and the remainder of the problem is solved recursively until the full environment can be solved reliably.

Therefore, the ultimate output of the PBCS algorithm is a sequence of policies (a chain of skills) that can be executed sequentially to reach the reward.

The addition of skill chaining can seem straightforward, but it adds a new challenge: in PB, the RL algorithm is trained on a version of the environment in which the reset point gradually changes, but the reward function is kept identical to the one in the underlying environment. In contrast, in PBCS the RL algorithm needs to be trained to reach arbitrary states in the environment

since any state of the Phase 1 trajectory can be used as the boundary between skills. In order to guide the agent towards an arbitrary state, we use a reward shaping strategy described in Section 7.2.3. This strategy solves two problems:

1. It allows the training of an agent to reach an arbitrary state of the environment, instead of the intended environment goal.
2. It replaces a sparse reward with a dense one, which suppresses the problem underlined in Chapter 6.

This gives rise to a form of paradox: we use skill chaining in order to solve the instability of RL algorithms, but then we need to introduce reward shaping which also solves the instability of RL by removing sparse rewards. A legitimate question would then be: why do we need skill chaining at all, since reward shaping solves the instability of RL?

This question has two answers: reward shaping can be misleading, and sparse rewards are not the only source of instability in RL. Introducing reward shaping is useful when the reward can be found by simply following the generated reward gradient. This may be the case for simple tasks, however in larger environments such as mazes, simple reward shaping strategies such as rewarding straight-line progress towards the goal fall short and tend to mislead the agent, in other words it provides deceptive gradients. Using reward shaping in conjunction with skill chaining limits the complexity of the sub-policies that need to be learned, and diminishes the flaws of reward shaping. Second, many works have demonstrated that RL algorithms can be unstable especially in environments with continuous states and actions. Therefore, the use of skill chaining is warranted even though reward shaping could in theory solve the problem of sparse rewards alone. This is summarized in Fig. 1 on page 18.

7.1 Related work

Skill chaining. The process of *skill chaining* was explored in different contexts by several research papers. Konidaris and Barto [2009] present an algorithm that incrementally learns a set of skills using classifiers to identify changepoints, while the method proposed by Konidaris et al. [2010] builds a skill tree from demonstration trajectories, and automatically detects changepoints using statistics on the value function.

In Bagaria and Konidaris [2019]; Slivinski et al. [2020], a *policy over options* is used to select from several learned skills. This approach uses a classifier to define the activation set of each policy, however, it does not rely on exploration data in the same way as our proposed method.

In the domain of Motion Planning (MP), sequencing local controllers has been used through pre-image backchaining [Kaelbling and Lozano-Pérez, 2017] and LQR-Trees [Tedrake, 2009] although these approaches do not fit the model-free paradigm.

To our knowledge, our approach is the first to use Backplay to build a skill chain. We believe that it is more reliable and minimizes the number of changepoints because the position of changepoints is decided using data from the RL algorithm which trains the policies involved in each skill.

Reward shaping. Many RL environments are designed so that the *natural* reward is continuous or near-continuous, for instance the score in a game, the speed of a virtual robot, the distance to a target. These environments are constructed so that reward shaping is not needed. However, some environments such as *Montezuma’s Revenge* provide hard exploration challenges. In this case, reward shaping is often used to encourage exploration. However, adding rewards to the environment can not only bias the policy search (which is the intended effect), but also bias the optimal policy. This can be avoided by adding a value to the environment reward $r(s, a, s')$ which is in the form of $\Psi(s') - \Psi(s)$ where Ψ is any function. Therefore, Ψ can be seen as a heuristic function that guides the agent towards the goal. Ng et al. [1999] prove that this is a necessary and sufficient condition for the optimal policy to be unbiased.

7.2 Methods

In this section, we introduce the PBCS algorithm, which builds on PB but is more robust to the instabilities of the underlying RL algorithm, and to the decay of the reward signal with longer training sessions.

We define $B_\epsilon(x)$ as the ball of radius ϵ centered around x in Euclidean distance, i.e.: $B_\epsilon(x) = \{y \mid \|x - y\| \leq \epsilon\}$.

7.2.1 Skill chaining algorithm

Algorithm 6 presents the skill chaining process. It uses a **Backplay** function which is inspired by the work done in Chapter 5 and detailed in Section 7.2.2. This function takes as input a trajectory $\tau_0 \dots \tau_T$, and returns a policy π and an index $K < T$ such that running policy π repeatedly on a state from $B_\epsilon(\tau_K)$ always leads to a state in $B_\epsilon(\tau_T)$.

The main loop of the skill chaining algorithm builds a chain of skills that roughly follows trajectory τ , but is able to improve upon it. Specifically, acti-

Algorithm 6: Skill chaining algorithm

Input : $\tau_0 \dots \tau_N$ the output of phase 1
Output : $\pi_0 \dots \pi_n$ a chain of policies with activation sets $A_0 \dots A_n$

```
1  $T \leftarrow N$ 
2  $n \leftarrow 0$ 
3 while  $T > 0$  do
4    $\pi_n, T \leftarrow \text{Backplay}(\tau_0 \dots \tau_T)$ 
5    $A_n \leftarrow B_\epsilon(\tau_T)$ 
6    $n \leftarrow n + 1$ 
7 end
8 Reverse lists  $\pi_0 \dots \pi_n$  and  $A_0 \dots A_n$ 
```

vation sets A_n are centered around points of τ but policies π_n are constructed using a generic RL algorithm that optimizes the path between two activation sets. The list of skills is then reversed, because it was constructed backwards.

7.2.2 Adapted backplay algorithm

In this section, we build on the backtracking algorithm presented in Section 5.3, which is itself a variant of an algorithm proposed by [Resnick et al., 2018].

The **Backplay** function (Algorithm 7) takes as input a segment $\tau_0 \dots \tau_T$ of the trajectory $\tau_0 \dots \tau_N$ obtained in phase 1, and returns a pair (K, π) where K is an index on trajectory τ , and π is a policy trained to reliably attain $B_\epsilon(\tau_T)$ from $B_\epsilon(\tau_K)$. The policy π is trained using DDPG to reach $B_\epsilon(\tau_T)$ from starting point $B_\epsilon(\tau_K)$ ¹, where K is initialized to $T - 1$, and gradually decremented in the main loop.

At each iteration, the algorithm evaluates the feasibility of a skill with target $B_\epsilon(\tau_T)$, policy π and activation set $B_\epsilon(\tau_K)$. If the measured performance is 100% without any training (Line 5), the current skill is saved and the starting point is decremented. Otherwise, a training loop is executed until performance stabilizes (Line 8). This is performed by running Algorithm 8 repeatedly until no improvement over the maximum performance is observed α times in a row. We ran our experiments with $\alpha = 10$, and a more in-depth discussion of hyperparameters is available in Section 7.5.

Then the performance of the skill is measured again (Line 9), and three cases are handled:

¹More details on why the starting point needs to be $B_\epsilon(\tau_K)$ instead of τ_K are available in Section 7.2.4

- **The skill is always successful (Line 10).** The current skill is saved and the index of the starting point is decremented.
- **The skill is never successful (Line 13).** The last successful skill is returned, unless no successful skill was ever found, in which case training continues on the same segment until a better policy is found.
- **The skill is sometimes successful.** The current skill is not saved, and the index of the starting point is decremented. In our maze environment, this happens when $B_\epsilon(\tau_K)$ overlaps a wall: in this case some states of $B_\epsilon(\tau_K)$ cannot reach the target no matter the policy.

Algorithm 7: The Backplay algorithm

Input : $(\tau_0 \dots \tau_T)$ a state-space trajectory
 α the stability criterion.

Output : π_s a trained policy
 K_s the index of the starting point of the policy

```

1  $K \leftarrow T - 1$ 
2 Initialize a DDPG architecture with policy  $\pi$ 
3 while  $K > 0$  do
4   Test performance of  $\pi$  between  $B_\epsilon(\tau_K)$  and  $B_\epsilon(\tau_T)$  over  $\beta$  episodes
5   if  $performance = 100\%$  then
6      $\pi_s \leftarrow \pi, K_s \leftarrow K$ 
7   else
8     Run Train (Algorithm 8) repeatedly until performance
      stabilizes (no improvement over  $\alpha$  sessions).
9     Test performance of  $\pi$  between  $B_\epsilon(\tau_K)$  and  $B_\epsilon(\tau_T)$  over  $\beta$ 
      episodes
10    if  $performance = 100\%$  then
11       $\pi_s \leftarrow \pi, K_s \leftarrow K$ 
12    end
13    if  $performance = 0\%$  and  $K_s$  exists then
14      return  $(K_s, \pi_s)$ 
15    end
16  end
17   $K \leftarrow K - 1$ 
18 end
19 return  $(\pi_s, K_s)$ 

```

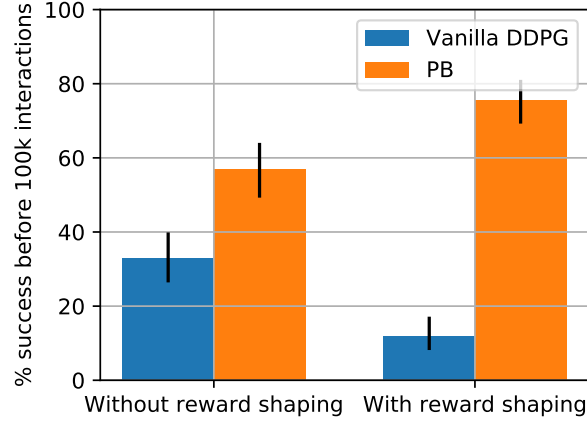


Figure 7.1: Performance of DDPG and PB on 2×2 mazes with and without reward shaping

7.2.3 Reward shaping

With reward shaping, we bypass the reward function of the environment, and train DDPG to reach any state τ_T . Ng et al. [1999] proved that the only way to avoid biasing the optimal policy when adding a shaped reward is to use a potential function. Therefore we define a potential function $\Phi(s) = \frac{1}{d(s, \tau_T)}$, where $d(s, \tau_T)$ is the L2 distance between s and the current training goal τ_T . We then define our shaped reward $R_{\text{shaped}}(s, a, s') = \begin{cases} 10 & \text{if } d(s, \tau_T) \leq \epsilon \\ \Phi(s') - \Phi(s) & \text{otherwise.} \end{cases}$.

In theory any positive constant can be used for the reward in $B_\epsilon(\tau_T)$. The potential function guarantees that only advances towards the goal are rewarded, and coupled with the discounted reward this progress is encouraged to happen as early as possible. Therefore once the threshold of $B_\epsilon(\tau_T)$ is reached, no further reward can be gained without entering the target area.

Reward shaping is an integral part of PBCS, however in the general case of RL it can create deceptive rewards. Indeed, even on a 2×2 maze, adding this shaped reward decreases the performance by about half. However, as shown in Fig. 7.1, adding reward shaping to the PB algorithm slightly increases its performance. Backplay eliminates the drawbacks of reward shaping because the agent starts close to the reward, therefore no further exploration is required and the deceptively-shaped gradient is overpowered by the reward signal of the target which has already been discovered.

Algorithm 8 shows how this reward function is used in place of the environment reward. This training function runs β episodes of up to `max_steps`

Algorithm 8: Training process with reward shaping

Input : τ_K the source state
 τ_T the target state
Output : The average performance p

```
1  $n \leftarrow 0$ 
2 for  $i = 1 \dots \beta$  do
3    $s \sim B_\epsilon(\tau_K)$ 
4   for  $j = 1 \dots \text{max\_steps}$  do
5      $a \leftarrow \pi(s) + \text{random noise}$ 
6      $s' \leftarrow \text{step}(s, a)$ 
7     if  $d(s', \tau_T) \leq \epsilon$  then
8        $r \leftarrow 10$ 
9     else
10       $r \leftarrow \frac{1}{d(s', \tau_T)} - \frac{1}{d(s, \tau_T)}$ 
11    end
12    DDPG.train( $s, a, s', r$ )
13     $s \leftarrow s'$ 
14    if  $d(s', \tau_T) \leq \epsilon$  then
15       $n \leftarrow n + 1$ 
16      break
17    end
18  end
19 end
20  $p \leftarrow \frac{n}{\beta}$ 
21 return  $p$ 
```

steps each, and returns the fraction of episodes that were able to reach the reward. β is a hyperparameter that we set to 50 for our test, and more details on this choice are available in Section 7.5.

Note that contrary to the backtracking algorithm presented in Fig. 5.1(b) on page 69, Line 3 of Algorithm 8 resets the environment to a state chosen randomly in $B_\epsilon(\tau_K)$ and not simply τ_K . This small change is necessary to be able to chain skills, as detailed in Section 7.2.4. A side effect of this change is reaching a performance of 100% is not always possible, even with long training sessions, because the starting point is selected in $B_\epsilon(\tau_K)$, and some of these states may be inside obstacles for instance.

7.2.4 Need for Resetting to Unseen States

As a reminder, for the Backplay algorithm and our variant, a single trajectory $\tau_0 \dots \tau_T$ is provided, and training is performed by changing the starting point of the environment to various states.

In the original Backplay algorithm, the environment is always reset to a visited state τ_K , where K is an index chosen randomly in a sliding window of $[0, T]$. The sliding window is controlled by hyperparameters, but the main idea is that in the early stages of training, states near T are more likely to be selected, while in later stages states near 0 are more likely to be selected.

However, we found that this caused a major issue when combined with continuous control and the skill chaining process. With skill chaining, the algorithm creates a sequence of activation sets (A_n) , and a sequence of policies (π_n) such that when the agent reaches a state in A_n , it switches to policy π_n . Each activation set A_n is a ball of radius ϵ centered around a state τ_K for some K .

The policy needs to be trained not only on portions of the environment that are increasingly long, it also needs to account for the uncertainty of its starting point. When executing the skill chain, the controller switches to policy π_n as soon as the state reaches the activation set A_n , which is $B_\epsilon(\tau_K)$ for some K . Even if A_n is relatively small, we found it caused systematic issues on maze environments, as presented in Fig. 7.2.

In our variant of the Backplay algorithm, we found it was necessary to train DDPG on starting points chosen randomly in $B_\epsilon(\tau_K)$, to ensure that the policy is trained correctly to solve a portion of the environment with any starting point in this volume.

This also means that we need to reset the environment to unseen states, and can cause problems when these states are unreachable (in our maze examples this is usually because they are inside walls, but in higher dimensions we assume this could be more problematic).

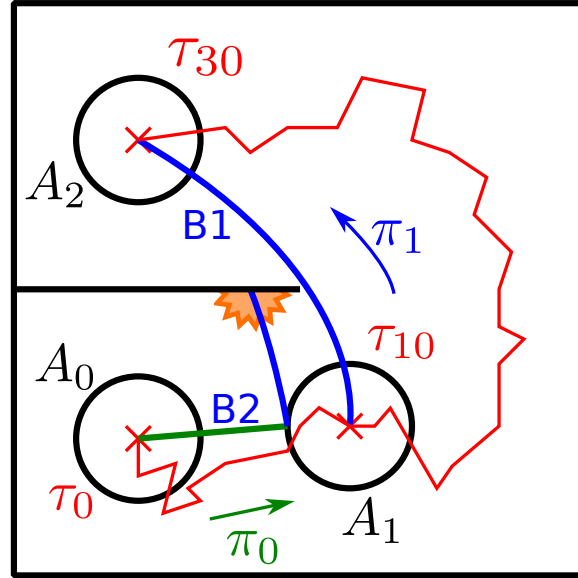


Figure 7.2: Policy π_1 was trained using starting points $\tau_{30} \dots \tau_{10}$ without any added noise. Therefore, τ_{30} is reachable from τ_{10} using π_1 (trajectory $B1$), but not necessarily from any point in A_1 . In maze environments, the optimal policy is usually close to walls, and provides little margin for perturbations. The trajectory $B2$ (that starts in green and ends in blue) results from the execution of the skill chain. The controller switches from π_0 to π_1 as soon as the agent reaches A_1 , and then hits the wall (trajectory $B2$). This problem persists even when ϵ is reduced.

When possible, a solution would be to run the environment backwards from τ_K with random actions to generate these samples (while ensuring that they still lie within $B_\epsilon(\tau_K)$). Another solution, especially in high-dimension environments, would be to run the environment backwards for a fixed number of steps, and use a classifier to define the bounds of A_n , instead of using the L2 distance.

7.3 Results on 2D mazes

We perform experiments in the same continuous maze environments as in Section 5.4 for the PB algorithm. For a maze of size N , the state space is the position of a point mass in $[0, N]^2$ and the action describes the speed of the point mass, in $[-0.1, 0.1]^2$. Therefore, the step function is simply $s' = s + a$, unless the $[s, s']$ segment intersects a wall. The only reward is -1 when hitting a wall and 1 when the target area is reached.

A maze of size N is generated using a maze generation algorithm with wall thickness 0.1 , the target position is $(N - .5, N - .5)$ when $N > 2$, or $(.5, 1.5)$ when $N = 2$. The Markov Decision Process (MDP) used for training is:

$$\begin{aligned} S &= [0, N] \times [0, N] \\ A &= [-0.1, 0.1] \times [-0.1, 0.1] \\ \text{step}(s, a) &= \begin{cases} s & \text{if } [s, s + a] \text{ intersects a wall} \\ s + a & \text{otherwise.} \end{cases} \\ R(s, a, s') &= \mathbb{1}_{\|s' - \text{target}\| < 0.2} - \mathbb{1}_{[s, s+a] \text{ intersects a wall}} \end{aligned}$$

We used a standard DDPG implementation [OpenAI, 2018] with default parameters. Each training session is limited to 50 episodes and an ϵ -greedy noise process is applied during rollouts, with $\epsilon = 0.1$ (ϵ -greedy noise is introduced in Section 6.2.2 on page 76).

7.4 Analysis of results

As expected, standard RL algorithms (DDPG and TD3) were unable to solve all but the simplest mazes. These algorithms have no mechanism for state-space exploration other than uniform noise added to their policies during rollouts. Therefore, in the best-case scenario they perform a random walk and, in the worst-case scenario, their actors may actively hinder exploration.

Figure 7.3: Results of various algorithms on maze environments. For each test, the number of environment steps performed is displayed with a red background when the policy was not able to reach the target, and a green one when training was successful.

In "Vanilla" experiments, the red paths represent the whole area explored by the RL algorithm. In "Backplay" experiments, the trajectory computed in phase 1 is displayed in red, and the "robustified" policy or policy chain is displayed in green. Activation sets A_i are displayed as purple circles. Enlarged images are presented in Fig. 7.4.

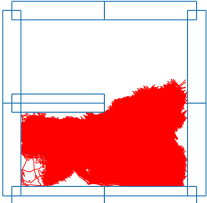
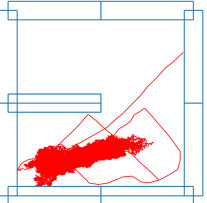
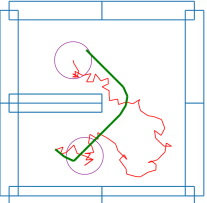
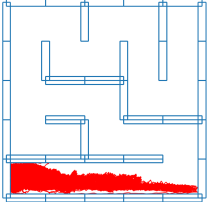
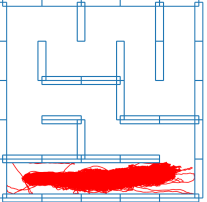
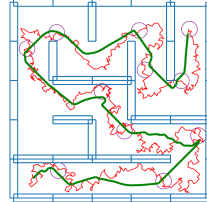
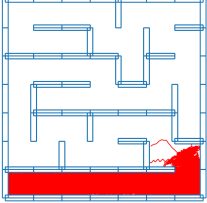
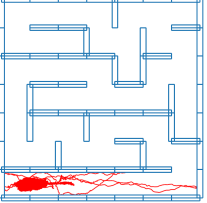
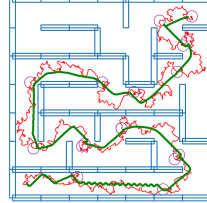
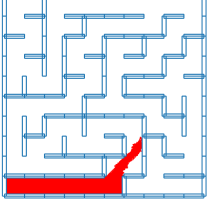
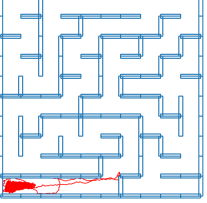
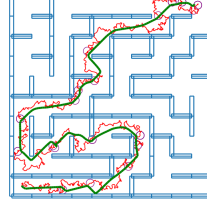
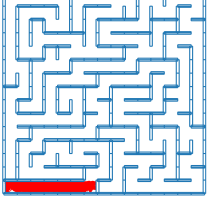
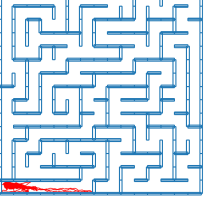
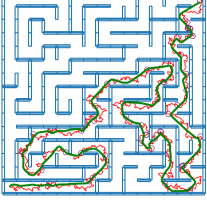
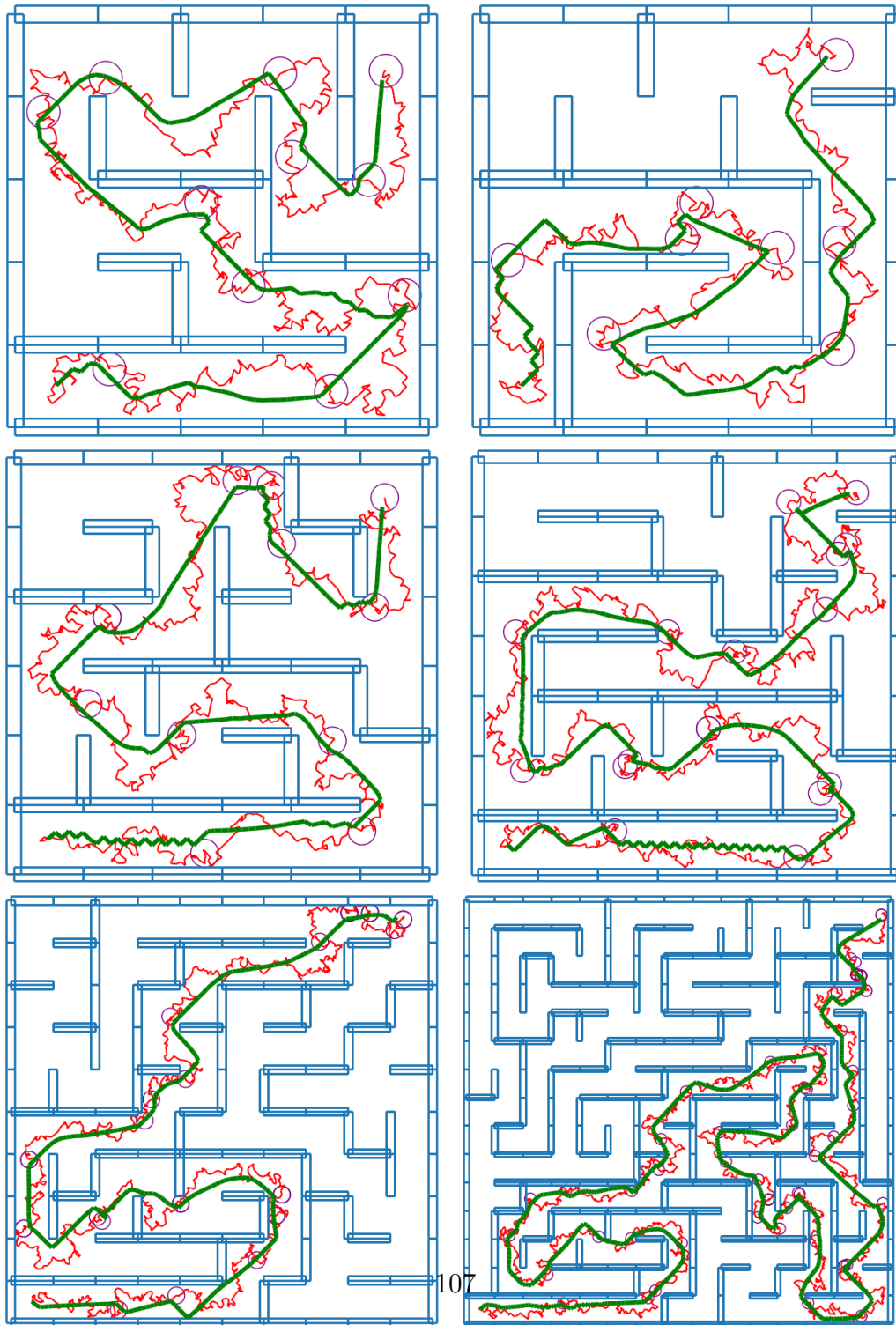
Vanilla		PBCS
DDPG	TD3	DDPG
 1M	 1M	 321k
 1M	 1M	 5M
 1M	 1M	 6M
 1M	 1M	 8M
 1M	 1M	 22M

Figure 7.4: Enlarged view of the results of PBCS on mazes of different sizes. The trajectory computed in phase 1 is displayed in red, and the "robustified" policy chain is displayed in green. Activation sets A_i are displayed as purple circles.



As seen in Chapter 5, backtracking alone (in the form of the PB algorithm) is not sufficient to solve mazes more complex than 2×2 , a failure that we attributed to inherent limitations of DDPG such as the one studied in Chapter 6.

However with the addition of skill chaining, PBCS is able to overcome these issues by limiting the length of training sessions of DDPG, and is able to solve complex mazes up to 7×7 , by chaining several intermediate skills.

7.5 Influence of hyperparameters

PBCS uses three hyperparameters α , β and ϵ .

The parameter α represents the number of consecutive non-improvements of the training performance required to assume training is finished. In our experiments, this value was set to 10, and we summarize here what can be expected if this parameter is set too high or too low.

- Setting α too high results in longer training sessions, in which the policy keeps being trained despite being already successful. The time and sample performance of PBCS is impacted, but the algorithm should still be able to build a policy chain.
- Setting α too low may cause training to stop early. In benign cases, the policy is simply sub-optimal, but in some cases this may lead to the creation of many changepoints, and prevent PBCS from improving at all upon the phase 1 trajectory. If the activation conditions overlap too much, PBCS may output a skill chain that is unable to navigate the environment.

The parameter β represents the number of samples used to evaluate the performance of a skill. In our experiments, this value was set to 50.

- Setting β too high would increase the time and sample complexity of PBCS, but would not impact the output.
- The main risk of setting β too low is that PBCS may incorrectly compute that a skill has a performance of 100%. If this skill is then selected, the output skill chain may be unable to navigate the environment.

The parameter ϵ corresponds to the radius of the targets used during skill chaining.

7.6 Conclusion

RL is unstable
in long sessions

→

Skill chaining

The results presented in Section 7.3 show that the addition of skill chaining to PB allows to train RL algorithms on short sections of the environment, therefore mitigating their instability in long training sessions.

Skill chaining

→

Need to
train towards
arbitrary targets

The addition of skill chaining requires the algorithm to be able to train to reach arbitrary states, instead of the natural goal of the environment.

Need to
train towards
arbitrary targets

→

Reward shaping

Section 7.2.3 presents a solution to this problem in the form of reward shaping, which allows us to guide a RL algorithm towards any state.

Sparse re-
wards cause
failures of RL

→

Reward shaping

As an added benefit, using a continuous shaped reward eliminates potential deadlocks in RL which we presented in Chapter 6.

Reward shaping

→

Deceptive
gradients

However, the introduction of continuous reward shaping comes at a cost: the reward gradient may be deceptive and prevent the agent from finding the ultimate goal

Deceptive
gradients



Backplay

Fortunately, as detailed in Section 7.2.3 this is mitigated by the use of backplay, which eliminates any possible dead-ends by ensuring the ultimate reward is immediately known and used to bootstrap the value estimates of the RL algorithm.

Conclusion

In this work, we studied the use of Reinforcement Learning (RL) algorithms for solving simulated robotics tasks. We encountered a major issue that is exploring a large-dimension environment to find a goal state, a task that RL algorithms are not designed to accomplish.

We demonstrated that Motion Planning (MP) algorithms were suited to this task although they require the use of an unusual primitive for RL which is the ability to reset to any previously-visited state. MP algorithms are often designed to take advantage of invertible models of the robot behavior, however some algorithms such as Rapidly-exploring Random Tree (RRT) and Expansive Spaces Tree (EST) can be adapted to operate without any knowledge of the robot or environment. We also proposed a novel algorithm called Ex that outperforms RRT on environments where the sampling space is hard to specify in a way that makes sampling planners efficient.

Next, we found that the data provided by MP algorithms was not well-suited to robotics control because they are in the form of a tree of transitions and not a controller, which makes generalization difficult especially in the absence of an environment model or local planner. This called for the use of a second phase where an RL algorithm uses the exploration data to overcome the difficulty of exploration, while outputting a robust policy. However, even so-called *off-policy* algorithms such as Deep Deterministic Policy Gradient (DDPG) are unable to learn from this exploration data alone (this would require *offline learning*, which is an ongoing research topic). With additional rollouts, DDPG can take advantage of the exploration data preloaded in its replay buffer, however this process does not scale well, and more generally we argued that the connectivity information in the exploration data is crucial in some environments such as mazes with thin walls, and therefore any algorithm

that relies solely on a disconnected set of transitions is bound to fail in these environments.

Therefore, we used the connectivity data of the exploration tree in the form of a single trajectory from the environment reset point to a goal state that is used as a learning curriculum through the use of backtracking. We called this technique Plan, Backplay (PB) and demonstrated that it outperforms DDPG on small mazes. PB also uses the reset-anywhere primitive, mirroring the exploration process. However, this technique does not scale to larger maze environments, in part due to the inherent instability of DDPG and other off-policy RL algorithms.

This led us to identify a fundamental problem of DDPG, which takes the form of a deadlock that can occur when the critic of DDPG converges to a value that is expected, but provides little to no information to the actor update, which leads the actor to be stuck to a sub-optimal policy. We demonstrated this issue on a trivial environment, and showed that it could be generalized to any sparse-reward environment, although the side effects of function approximation can mitigate this issue.

Finally, we presented a new algorithm called Plan, Backplay, Chain Skills (PBCS) that solves this deadlock by introducing reward shaping and therefore removing the sparse reward. Usually, this would doom the exploration capabilities of the underlying RL algorithm by providing a deceptive gradient, but when used in conjunction with backtracking, this is not an issue. PBCS also mitigates the instability of DDPG by introducing *skill chaining*, a technique that fits especially well with the backtracking process. We demonstrated that with this technique, DDPG was able to reliably traverse large mazes, up to 15×15 cells.

Limitations

Despite our promising results, several limitations restrict the scope of our study and proposed approaches. Some of these limitations can be mitigated, while others are inherent to the technique and may be more difficult to overcome.

Too many skills in mazes. In our experiments with PBCS on 2D mazes, the expected result was to let the underlying RL algorithm handle most of the navigation, and use skill chaining only as a last resort when either the reward horizon was too far to propagate any useful signal, or the RL algorithm diverges due to inherent limitations. However, we observed that a new skill was triggered very often, meaning that the RL algorithm was able to learn

the correct action to traverse the maze in a straight line or a slight turn, but did not generalize well to complex paths. We see this as a limitation of the underlying RL algorithm, and hope that the next generation of off-policy learning algorithms will benefit our approach as well.

Reset-anywhere in non-holonomic systems. The activation set of a learned skill cannot be a single state, because reaching a given state is nearly impossible in a continuous environment. Therefore, in PBCS we define activation sets as balls with a fixed radius. This alone should be worrisome: in environments with non-holonomic constraints or thin obstacles, there may be no short trajectory linking two states that are close in state space. A potential solution to this problem is the use of a reverse-time simulator: the set of states from which s can be reached in n steps can be sampled by running the simulation backwards from s during n steps with random actions. This sampling could be directly used to train a tentative skill, however when performing rollouts the controller needs to decide when to switch skills. This could for instance be performed by training a classifier on the samples that were generated by backwards simulation, and use it to decide whether a state is in the activation set of the skill.

Limitations on the environment. In our work, we focused on deterministic environments with a single start area and a single goal area. We made the case that non-sparse rewards are often used improperly in environments where a sparse reward would be more logical. However, there certainly are cases in which a dense reward is pertinent such as maximizing speed, minimizing cost, combining several objectives. RL algorithms are well suited to these tasks, and do not necessarily suffer from the exploration challenge on which we focused. Some tasks are hybrid in the sense that it is hard to find any reward, but once some reward is found it needs to be maximized. This is an area of research that we do not cover in this work. The real world is often modeled as deterministic, however stochastic environments are useful to describe uncertainty in the environment. Our work only deals with deterministic environments, restricting its scope.

Limitations on the observations. Our work mostly deals with simulations and therefore assumes that the state is known, however many earlier RL algorithms function in environments where only partial observations are available, either because of uncertainty on the environment, or because the observation is obtained through sensors. Using the reset-anywhere primitive negates this capability of RL, which is a big limitation since RL is well-known

for its capabilities in image and sensor data processing.

Perspectives

We believe that the use of reward-driven RL has allowed for incredible results when performing complex motions that are limited in time, such as throwing a ball, or simple walk cycles. However, the time horizons required for more intricate sequences of actions such as traversing an environment show a fundamental limit with the concept of discounted rewards. The exponential nature of the discounted reward after n steps γ^n allows for a constant reward gradient along a trajectory, but this requires tuning γ , and this gradient gets smaller with longer time-horizons.

Furthermore, RL algorithms that use an experience replay buffer sample from this buffer either uniformly, or with strategies that did not prove beneficial in our benchmarks. Combined with the fact that updates to the policy are often non-local, this makes learning a long trajectory challenging.

We found that integrating exploration data in RL was challenging mostly because common RL algorithms are not *offline*, in the sense that they need to interact with the environment themselves, and cannot solely rely on prior rollouts. Offline learning is an active topic [Fu, 2020] which may yield promising results although some cases are likely to remain problematic (see Chapter 4).

We believe that splitting the learning process into exploration and exploitation components is beneficial, however in more challenging environments, performing a full exploration of the state-space will likely be intractable. In this case, more communication may be required between the exploration and RL components. This could take the form of alternating between the two behaviors, or more intricate approaches as explored by [Brown et al., 2020; Chen et al., 2020].

Recent advances have improved the stability and speed of RL algorithms, but we believe that the next breakthrough in the domain of robotics control using RL will require some kind of compromise, both on the RL and MP sides. On the MP side, this could be through the use of RL agents for sub-tasks such as local planning or torque control. On the RL side, this could be achieved either through modeling, expert input, or allowing the RL agent to reset to arbitrary states (which is the path we took within this thesis).

The field of MP is an obvious neighbor when using RL algorithms to control virtual or real-world robots, therefore we believe that bridging the gap between these domains will be the key to future major advances.

Bibliography

- Joshua Achiam and Miguel Morales. Part 2: Kinds of RL algorithms — Spinning Up documentation, November 2018. URL https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html. [Cited on page 31.]
- Joshua Achiam, Ethan Knight, and Pieter Abbeel. Towards characterizing divergence in Deep Q-Learning. *arXiv:1903.08894 [cs]*, March 2019. URL <http://arxiv.org/abs/1903.08894>. [Cited on pages 15, 66, and 75.]
- Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight Experience Replay. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5048–5058. Curran Associates, Inc., 2017. [Cited on page 37.]
- Akhil Bagaria and George Konidaris. Option discovery using deep skill chaining. In *International Conference on Learning Representations*, September 2019. URL <https://openreview.net/forum?id=B1gqipNYwH>. [Cited on page 97.]
- Leemon C. Baird and A. Harry Klopff. Reinforcement learning with high-dimensional, continuous actions. 1993. doi: 10.21236/ada280844. [Cited on page 75.]
- Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuron-like adaptive elements that can solve difficult learning control problems.

- IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, September 1983. ISSN 2168-2909. doi: 10.1109/TSMC.1983.6313077. [Cited on page 34.]
- Richard Bellman. A Markovian Decision Process. *Indiana University Mathematics Journal*, 6(4):679–684, 1957. ISSN 0022-2518. doi: 10.1512/iumj.1957.6.56038. URL <http://www.iumj.indiana.edu/IUMJ/fulltext.php?artid=56038&year=1957&volume=6>. [Cited on page 131.]
- Homanga Bharadhwaj, Animesh Garg, and Florian Shkurti. LEAF: Latent Exploration Along the Frontier. *arXiv:2005.10934 [cs]*, June 2020. URL <http://arxiv.org/abs/2005.10934>. [Cited on page 37.]
- Nick Bostrom. What happens when our computers get smarter than we are?, 2015. URL https://www.ted.com/talks/nick_bostrom_what_happens_when_our_computers_get_smarter_than_we_are. [Cited on page 14.]
- Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in neural information processing systems*, pages 369–376, 1995. [Cited on page 75.]
- Noam Brown, Anton Bakhtin, Adam Lerer, and Qucheng Gong. Combining Deep Reinforcement Learning and Search for Imperfect-Information Games. *arXiv:2007.13544 [cs]*, July 2020. URL <http://arxiv.org/abs/2007.13544>. [Cited on page 114.]
- Fanfei Chen, John D. Martin, Yewei Huang, Jinkun Wang, and Brendan Englot. Autonomous Exploration Under Uncertainty via Deep Reinforcement Learning on Graphs. *arXiv:2007.12640 [cs]*, July 2020. URL <http://arxiv.org/abs/2007.12640>. [Cited on page 114.]
- Hao-Tien Lewis Chiang, Jasmine Hsu, Marek Fiser, Lydia Tapia, and Aleksandra Faust. RL-RRT: Kinodynamic motion planning via learning reachability estimators from RL policies. *arXiv:1907.04799 [cs]*, July 2019. URL <http://arxiv.org/abs/1907.04799>. [Cited on page 37.]
- Geoffrey Cideron, Thomas Pierrot, Nicolas Perrin, Karim Beguir, and Olivier Sigaud. QD-RL: efficient mixing of quality and diversity in reinforcement learning. *arXiv:2006.08505 [cs]*, June 2020. URL <http://arxiv.org/abs/2006.08505>. [Cited on page 36.]
- Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. GEP-PG: Decoupling exploration and exploitation in deep reinforcement learning algorithms.

- arXiv:1802.05054 [cs.LG]*, February 2018. URL <https://arxiv.org/abs/1802.05054>. [Cited on pages 36, 62, 75, and 131.]
- L. E. Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, 79(3):497–516, 1957. ISSN 0002-9327. doi: 10.2307/2372560. URL <https://www.jstor.org/stable/2372560>. [Cited on page 22.]
- Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Go-Explore: A new approach for hard-exploration problems. *arXiv:1901.10995 [cs, stat]*, January 2019. URL <http://arxiv.org/abs/1901.10995>. [Cited on pages 17, 67, and 131.]
- Paul Erdős and Samuel James Taylor. Some intersection properties of random walk paths. *Acta Mathematica Academiae Scientiarum Hungarica*, 11(3): 231–248, September 1960. ISSN 1588-2632. doi: 10.1007/BF02020942. URL <https://doi.org/10.1007/BF02020942>. [Cited on page 52.]
- Lawrence H. Erickson and Steven M. Lavalle. Survivability: Measuring and ensuring path diversity. *2009 IEEE International Conference on Robotics and Automation*, 2009. doi: 10.1109/ROBOT.2009.5152773. [Cited on page 36.]
- Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is All You Need: Learning skills without a reward function. *arXiv:1802.06070 [cs]*, October 2018. URL <http://arxiv.org/abs/1802.06070>. [Cited on page 36.]
- Mahdi Fakoore, Amirreza Kosari, and Mohsen Jafarzadeh. Revision on fuzzy artificial potential field for humanoid robot path planning in unknown environment. *International Journal of Advanced Mechatronic Systems*, 6(4):174, 2015. ISSN 1756-8412, 1756-8420. doi: 10.1504/IJAMECHS.2015.072707. URL <http://www.inderscience.com/link.php?id=72707>. [Cited on page 24.]
- Aleksandra Faust, Oscar Ramirez, Marek Fiser, Kenneth Oslund, Anthony Francis, James Davidson, and Lydia Tapia. PRM-RL: long-range robotic navigation tasks by combining reinforcement learning and sampling-based planning. *arXiv:1710.03937 [cs]*, May 2018. URL <http://arxiv.org/abs/1710.03937>. [Cited on page 37.]

- Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. Reverse curriculum generation for reinforcement learning. *arXiv:1707.05300 [cs]*, July 2018. URL <http://arxiv.org/abs/1707.05300>. [Cited on page 68.]
- Sébastien Forestier, Yoan Mollard, and Pierre-Yves Oudeyer. Intrinsically motivated goal exploration processes with automatic curriculum learning. *arXiv:1708.02190 [cs]*, August 2017. URL <http://arxiv.org/abs/1708.02190>. [Cited on pages 36 and 131.]
- Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration. *arXiv:1706.10295 [cs, stat]*, June 2017. URL <http://arxiv.org/abs/1706.10295>. [Cited on page 75.]
- Justin Fu. D4RL: Building Better Benchmarks for Offline Reinforcement Learning, June 2020. URL <http://bair.berkeley.edu/blog/2020/06/25/D4RL/>. [Cited on page 114.]
- Scott Fujimoto, David Meger, and Doina Precup. Off-policy deep reinforcement learning without exploration. *arXiv:1812.02900 [cs, stat]*, December 2018a. URL <http://arxiv.org/abs/1812.02900>. [Cited on page 75.]
- Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *International Conference on Machine Learning*, 2018b. [Cited on pages 15, 16, 80, and 132.]
- Matthieu Geist and Olivier Pietquin. Parametric value function approximation: A unified view. In *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning 2011*, pages 9–16, Paris, France, April 2011. doi: 10.1109/ADPRL.2011.5967355. URL <https://hal-supelec.archives-ouvertes.fr/hal-00618112>. [Cited on page 85.]
- Github. jostbr/pymaze, February 2018a. URL <https://github.com/jostbr/pymaze>. [Cited on page 70.]
- Github. openai/gym. OpenAI, January 2018b. URL <https://github.com/openai/gym>. [Cited on page 93.]
- Github. erlerobot/gym-gazebo. Erle Robotics, July 2020a. URL <https://github.com/erlerobot/gym-gazebo>. [Cited on page 134.]

- Github. openai/baselines. OpenAI, September 2020b. URL <https://github.com/openai/baselines>. [Cited on page 57.]
- Github. openai/rosbridge. OpenAI, February 2020c. URL <https://github.com/openai/rosbridge>. [Cited on page 134.]
- Github. protocolbuffers/protobuf. Protocol Buffers, August 2020d. URL <https://github.com/protocolbuffers/protobuf>. [Cited on page 134.]
- Peter Goldsborough. A Promenade of PyTorch, February 2018. URL <http://www.goldsborough.me/ml/ai/python/2018/02/04/20-17-20-a-promenade-of-pytorch/>. [Cited on page 133.]
- Anirudh Goyal, Philemon Brakel, William Fedus, Soumye Singhal, Timothy Lillicrap, Sergey Levine, Hugo Larochelle, and Yoshua Bengio. Recall traces: Backtracking models for efficient reinforcement learning. *arXiv:1804.00379 [cs, stat]*, January 2019. URL <http://arxiv.org/abs/1804.00379>. [Cited on page 67.]
- Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv:1801.01290 [cs, stat]*, January 2018. URL <http://arxiv.org/abs/1801.01290>. [Cited on pages 36, 92, and 132.]
- Jakob J. Hollenstein, Erwan Renaudo, and Justus Piater. Improving exploration of deep reinforcement learning using planning for policy search. *Submitted to the International Conference on Learning Representations*, September 2019. URL <https://openreview.net/forum?id=rJe7CkrFvS>. [Cited on page 62.]
- Ionel-Alexandru Hosu and Traian Rebedea. Playing atari games with deep reinforcement learning and human checkpoint replay. *arXiv:1607.05077 [cs]*, July 2016. URL <http://arxiv.org/abs/1607.05077>. [Cited on page 30.]
- David Hsu, Jean-Claude Latombe, and Rajeev Motwani. Path planning in expansive configuration spaces. In *Proceedings of International Conference on Robotics and Automation*, volume 3, pages 2719–2726 vol.3, April 1997. doi: 10.1109/ROBOT.1997.619371. [Cited on pages 45, 46, and 131.]
- Serena Ivaldi, Jan Peters, Vincent Padois, and Francesco Nori. Tools for simulating humanoid robot dynamics: a survey based on user feedback. In *IEEE-RAS International Conference on Humanoid Robots (Humanoids)*, Madrid, Spain, 2014. URL <https://hal.archives-ouvertes.fr/hal-01116148>. [Cited on page 134.]

- Boris Ivanovic, James Harrison, Apoorva Sharma, Mo Chen, and Marco Pavone. BaRC: backward reachability curriculum for robotic reinforcement learning. *2019 International Conference on Robotics and Automation (ICRA)*, 2019. doi: 10.1109/ICRA.2019.8794206. [Cited on page 68.]
- Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z. Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016. [Cited on page 90.]
- Leslie Kaelbling and Tomás Lozano-Pérez. Pre-image backchaining in belief space for mobile manipulation. In *Robotics Research: The 15th International Symposium*, volume 100, pages 383–400. January 2017. ISBN 978-3-319-29362-2. doi: 10.1007/978-3-319-29363-9_22. [Cited on page 98.]
- Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, and Vincent Vanhoucke. Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation. *arXiv preprint arXiv:1806.10293*, 2018. [Cited on page 92.]
- Lydia Kavraki, Petr Svestka, Jean-Claude Latombe, and M.H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *Robotics and Automation, IEEE Transactions on*, 12:566–580, September 1996. doi: 10.1109/70.508439. [Cited on page 40.]
- Joseph Klann. Klann linkage in 4 different positions throughout the cycle., 2005. URL <https://commons.wikimedia.org/wiki/File:F1-positions.gif>. [Cited on pages 11 and 20.]
- Ross A. Knepper and Matthew T. Mason. Path diversity is only part of the problem. *2009 IEEE International Conference on Robotics and Automation*, 2009. doi: 10.1109/ROBOT.2009.5152696. [Cited on page 36.]
- George Konidaris and Andrew Barto. Autonomous shaping: Knowledge transfer in reinforcement learning. In *Proceedings of the 23rd international conference on Machine learning*, pages 489–496, 2006. [Cited on page 90.]
- George Konidaris and Andrew G. Barto. Skill discovery in continuous reinforcement learning domains using skill chaining. In Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 1015–1023. Curran Associates, Inc., 2009. [Cited on page 97.]

- George Konidaris, Scott Kuindersma, Roderic Grupen, and Andrew G. Barto. Constructing skill trees for reinforcement learning agents from demonstration trajectories. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 1162–1170. Curran Associates, Inc., 2010. [Cited on page 97.]
- Yaqing Lai, Wufan Wang, Yunjie Yang, Jihong Zhu, and Minchi Kuang. Hindsight planner. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '20, pages 690–698, Auckland, New Zealand, May 2020. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-1-4503-7518-4. [Cited on page 37.]
- Sascha Lange, Thomas Gabel, and Martin Riedmiller. Batch reinforcement learning. In Marco Wiering and Martijn van Otterlo, editors, *Reinforcement Learning: State-of-the-Art*, Adaptation, Learning, and Optimization, pages 45–73. Springer, Berlin, Heidelberg, 2012. ISBN 978-3-642-27645-3. doi: 10.1007/978-3-642-27645-3_2. URL https://doi.org/10.1007/978-3-642-27645-3_2. [Cited on page 59.]
- Jean-Claude Latombe. *Robot motion planning*. The Springer International Series in Engineering and Computer Science. Springer US, 1991. ISBN 978-0-7923-9206-4. doi: 10.1007/978-1-4615-4022-9. URL <https://www.springer.com/gp/book/9780792392064>. [Cited on page 19.]
- Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, Iowa State University, 1998. [Cited on pages 66 and 132.]
- Steven M. Lavalle and James Kuffner. Rapidly-exploring random trees: Progress and prospects. *Algorithmic and computational robotics: New directions*, January 2000. [Cited on pages 42 and 132.]
- Joel Lehman and Kenneth O. Stanley. Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 211–218, Dublin, Ireland, July 2011. Association for Computing Machinery. ISBN 978-1-4503-0557-0. doi: 10.1145/2001576.2001606. URL <https://doi.org/10.1145/2001576.2001606>. [Cited on pages 36 and 131.]

- Jan Leike, David Krueger, Tom Everitt, Miljan Martic, Vishal Maini, and Shane Legg. Scalable agent alignment via reward modeling: a research direction. *arXiv:1811.07871 [cs, stat]*, November 2018. URL <http://arxiv.org/abs/1811.07871>. [Cited on page 15.]
- Jure Leskovec, Anand Rajaraman, and Jeff Ullman. Finding Similar Items. In *Mining of Massive Datasets*. Chapter 3, January 2020. ISBN 978-1-108-47634-8. URL <http://www.mmds.org/>. [Cited on page 50.]
- Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems. *arXiv:2005.01643 [cs, stat]*, May 2020. URL <http://arxiv.org/abs/2005.01643>. [Cited on page 59.]
- Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv:1509.02971 [cs, stat]*, September 2015. URL <http://arxiv.org/abs/1509.02971>. [Cited on pages 15, 16, 27, 66, and 130.]
- Kevin M. Lynch and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, USA, 1st edition, 2017. ISBN 978-1-107-15630-2. [Cited on page 19.]
- Guillaume Matheron, Nicolas Perrin, and Olivier Sigaud. The problem with DDPG: Understanding failures in deterministic environments with sparse rewards. *arXiv:1911.11679 [cs, stat]*, November 2019. URL <http://arxiv.org/abs/1911.11679>. [Cited on pages 16, 17, and 130.]
- Guillaume Matheron, Nicolas Perrin, and Olivier Sigaud. PBCS: Efficient exploration and exploitation using a synergy between reinforcement learning and motion planning. *arXiv:2004.11667 [cs, stat]*, April 2020. URL <http://arxiv.org/abs/2004.11667>. [Cited on pages 17, 131, and 132.]
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. *arXiv:1312.5602 [cs]*, December 2013. URL <http://arxiv.org/abs/1312.5602>. [Cited on pages 15, 29, 66, 75, and 131.]
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dhharshan Kumaran, Daan Wierstra,

- Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. ISSN 1476-4687. doi: 10.1038/nature14236. URL <https://www.nature.com/articles/nature14236>. [Cited on page 27.]
- Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *arXiv:1602.01783 [cs]*, June 2016. URL <http://arxiv.org/abs/1602.01783>. [Cited on page 35.]
- Philippe Morere, Gilad Francis, Tom Blau, and Fabio Ramos. Reinforcement learning with probabilistically complete exploration. *arXiv:2001.06940 [cs, stat]*, January 2020. URL <http://arxiv.org/abs/2001.06940>. [Cited on page 68.]
- Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv:1504.04909 [cs, q-bio]*, April 2015. URL <http://arxiv.org/abs/1504.04909>. [Cited on page 36.]
- Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, International Conference on Machine Learning '99, pages 278–287, June 1999. ISBN 978-1-55860-612-8. [Cited on pages 98 and 101.]
- OpenAI. Spinning up in Deep RL, November 2018. URL <https://blog.openai.com/spinning-up-in-deep-rl/>. [Cited on pages 53, 70, and 105.]
- Hugo Penedones, Damien Vincent, Hartmut Maennel, Sylvain Gelly, Timothy Mann, and Andre Barreto. Temporal difference learning with neural networks - study of the leakage propagation problem. *arXiv:1807.03064 [cs, stat]*, July 2018. URL <http://arxiv.org/abs/1807.03064>. [Cited on page 64.]
- J.M. Phillips, N. Bedrossian, and L.E. Kavraki. Guided Expansive Spaces Trees: a search strategy for motion- and cost-constrained state spaces. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, volume 4, pages 3968–3973 Vol.4, April 2004. doi: 10.1109/ROBOT.2004.1308890. [Cited on page 46.]
- Matthias Plappert, Rein Houthoofd, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin

- Andrychowicz. Parameter space noise for exploration. *arXiv preprint arXiv:1706.01905*, 2017. [Cited on page 75.]
- David Pratt. The curse of dimensionality: A visual approach, March 2018. URL <https://blog.midnightmechanism.com/post/dimensionality-curse/>. [Cited on page 52.]
- Justin K. Pugh, Lisa B. Soros, and Kenneth O. Stanley. Quality diversity: a new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3, 2016. ISSN 2296-9144. doi: 10.3389/frobt.2016.00040. URL <https://www.frontiersin.org/articles/10.3389/frobt.2016.00040/full>. [Cited on page 36.]
- James A. Reeds and Larry A. Shepp. Optimal paths for a car that goes both forwards and backwards. 1990. doi: 10.2140/pjm.1990.145.367. [Cited on page 21.]
- Cinjon Resnick, Roberta Raileanu, Sanyam Kapoor, Alexander Peysakhovich, Kyunghyun Cho, and Joan Bruna. Backplay: "Man muss immer umkehren". *arXiv:1807.06919 [cs, stat]*, December 2018. URL <http://arxiv.org/abs/1807.06919>. [Cited on pages 17, 66, 67, 68, 99, and 131.]
- Martin Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degraeve, Tom Van de Wiele, Volodymyr Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing - solving sparse reward tasks from scratch. *arXiv:1802.10567 [cs, stat]*, February 2018. URL <http://arxiv.org/abs/1802.10567>. [Cited on page 90.]
- Tim Salimans and Richard Chen. Learning Montezuma’s Revenge from a single demonstration. *arXiv:1812.03381 [cs, stat]*, December 2018. URL <http://arxiv.org/abs/1812.03381>. [Cited on pages 30 and 67.]
- Samir. Using Reinforcement Learning to Perform Motion Planning for a YuMi Robot, December 2016. URL <https://robosamir.github.io/DDPG-on-a-Real-Robot/>. [Cited on page 134.]
- Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, July 1959. ISSN 0018-8646. doi: 10.1147/rd.33.0210. URL <https://doi.org/10.1147/rd.33.0210>. [Cited on page 19.]
- Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay. *arXiv:1511.05952 [cs]*, November 2015. URL <http://arxiv.org/abs/1511.05952>. [Cited on page 66.]

- John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust Region Policy Optimization. *arXiv:1502.05477 [cs]*, February 2015. URL <http://arxiv.org/abs/1502.05477>. [Cited on pages 15, 30, 34, and 132.]
- John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]*, July 2017. URL <http://arxiv.org/abs/1707.06347>. [Cited on pages 34, 67, and 132.]
- Alexander Shkolnik, Matthew Walter, and Russ Tedrake. Reachability-guided sampling for planning under differential constraints. In *2009 IEEE International Conference on Robotics and Automation*, pages 2859–2865, May 2009. doi: 10.1109/ROBOT.2009.5152874. [Cited on page 39.]
- David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic Policy Gradient algorithms. In *International Conference on Machine Learning*, pages 387–395, January 2014. URL <http://proceedings.mlr.press/v32/silver14.html>. [Cited on page 27.]
- David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529 (7587):484–489, January 2016. ISSN 1476-4687. doi: 10.1038/nature16961. URL <https://www.nature.com/articles/nature16961>. [Cited on pages 19 and 30.]
- Riley Simmons-Edler, Ben Eisner, Eric Mitchell, Sebastian Seung, and Daniel Lee. Q-learning for continuous actions with cross-entropy guided policies. *arXiv:1903.10605 [cs]*, July 2019. URL <http://arxiv.org/abs/1903.10605>. [Cited on page 92.]
- Matthew Slivinski, George Konidaris, and Lauren E. Marshall. Robust deep skill chaining. M.Sc. Project Report, 2020. [Cited on page 97.]
- Héctor J. Sussmann and Guoqing Tang. Shortest paths for the Reeds-Shepp car: a worked out example of the use of geometric techniques in nonlinear optimal control. Technical Report SYCON-91-10, January 1991. [Cited on page 22.]

- Richard S. Sutton and Andrew G. Barto. 4.4 Value iteration. In *Reinforcement Learning: An Introduction*. MIT Press, November 2018a. ISBN 978-0-262-03924-6. [Cited on page 26.]
- Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, November 2018b. ISBN 978-0-262-03924-6. [Cited on pages 16, 32, and 75.]
- Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. #Exploration: A study of count-based exploration for deep reinforcement learning. *arXiv:1611.04717 [cs]*, November 2016. URL <http://arxiv.org/abs/1611.04717>. [Cited on pages 36, 52, and 130.]
- Russ Tedrake. LQR-Trees: Feedback motion planning on sparse randomized trees. In *Robotics: Science and Systems*, University of Washington, Seattle, USA, June 2009. doi: 10.15607/RSS.2009.V.003. [Cited on page 98.]
- Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, October 2012. doi: 10.1109/IROS.2012.6386109. [Cited on page 134.]
- John N. Tsitsiklis and Benjamin Van Roy. Analysis of temporal-difference learning with function approximation. In *Advances in neural information processing systems*, pages 1075–1081, 1997. [Cited on page 75.]
- Rasmus K. Ursem. Diversity-guided evolutionary algorithms. In Juan Julián Merelo Guervós, Panagiotis Adamidis, Hans-Georg Beyer, Hans-Paul Schwefel, and José-Luis Fernández-Villacañás, editors, *Parallel Problem Solving from Nature - PPSN VII*, pages 462–471, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45712-1. [Cited on page 36.]
- Hado Van Hasselt and Marco A. Wiering. Reinforcement learning in continuous action spaces. In *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 272–279. IEEE, 2007. [Cited on pages 93 and 130.]
- Hado van Hasselt, Yotam Doron, Florian Strub, Matteo Hessel, Nicolas Sonnerat, and Joseph Modayil. Deep reinforcement learning and the deadly triad. *arXiv:1812.02648 [cs]*, December 2018. URL <http://arxiv.org/abs/1812.02648>. [Cited on pages 66 and 75.]

- Vojtech Vonásek and Martin Saska. Increasing diversity of solutions in sampling-based path planning. In *International Conference on Robotics and Artificial Intelligence 2018*, 2018. doi: 10.1145/3297097.3297114. [Cited on page 36.]
- Caleb Voss, Mark Moll, and Lydia E. Kavraki. A heuristic approach to finding diverse short paths. *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 2015. doi: 10.1109/ICRA.2015.7139774. [Cited on page 36.]
- Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD Thesis, King’s College, Oxford, 1989. [Cited on pages 29, 34, and 132.]
- Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992696. URL <https://doi.org/10.1007/BF00992696>. [Cited on page 32.]
- Edwin B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, June 1927. ISSN 0162-1459. doi: 10.1080/01621459.1927.10502953. [Cited on page 72.]
- Albert Wu, Sadra Sadraddini, and Russ Tedrake. R3T: Rapidly-exploring Random Reachable Set Tree for Optimal Kinodynamic Planning of Non-linear Hybrid Systems. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4245–4251, May 2020. doi: 10.1109/ICRA40945.2020.9196802. [Cited on page 39.]
- Jun Yamada. Motion planner augmented action spaces for reinforcement learning, June 2020. URL <https://www.junjunggoal.tech/publication/mopa/>. [Cited on page 37.]
- Matthieu Zimmer and Paul Weng. Exploiting the sign of the advantage function to learn deterministic policies in continuous domains. *arXiv preprint arXiv:1906.04556*, 2019. [Cited on page 93.]

Acronyms

CACLA Continuous Actor-Critic Learning Automata. 93, *Glossary:* CACLA

DDPG Deep Deterministic Policy Gradient. 12, 17, 19, 26–29, 32–36, 56–64, 67–84, 89–92, 94, 95, 101, 105, 106, 108, 111, 112, 130–132, *Glossary:* DDPG

DDPG-argmax Deep Deterministic Policy Gradient - argmax. 12, 91–95, *Glossary:* DDPG-argmax

DQN Deep Q-Network. 26, 27, 29, 32–35, 75, *Glossary:* DQN

EST Expansive Spaces Tree. 45, 46, 48, 50, 52, 53, 111, 131, *Glossary:* EST

GEP Goal Exploration Processes. 36, *Glossary:* GEP

LfD Learning from Demonstration. 67

MDP Markov Decision Process. 19, 23–25, 38, 48, 57, 64, 69, 76, 105, 131, 132, *Glossary:* MDP

ML Machine Learning. 19

MP Motion Planning. 15–17, 19, 22–24, 30, 31, 36–38, 45, 54–56, 59, 66, 67, 73, 98, 111, 114, 131, 132, 134

NSLC Novelty Search with Local Competition. 36, *Glossary:* NSLC

OU Ornstein-Uhlenbeck. 77

PB Plan, Backplay. 12, 17, 66–69, 71–74, 95, 96, 98, 101, 105, 108, 109, 112, 132, *Glossary*: PB

PBCS Plan, Backplay, Chain Skills. 17, 96, 98, 101, 106–108, 112, 113, *Glossary*: PBCS

PG Policy Gradient. 17, 32, 95

PPO Proximal Policy Optimization. 32, 67, 68, *Glossary*: PPO

RL Reinforcement Learning. 11, 15–17, 19, 23–25, 27, 28, 30–33, 35–38, 44, 46, 52, 54–56, 59, 63–68, 71, 73–75, 96–98, 101, 109–114, 130–134

RRT Rapidly-exploring Random Tree. 11, 23, 31, 38–46, 48, 50, 53, 54, 66, 68, 111, 132, 134, *Glossary*: RRT

RRT-NH Rapidly-exploring Random Tree Non-Holonomic. 11, 40, 42–46, 48, 52–59, *Glossary*: RRT-NH

SAC Soft Actor-Critic. 12, 32, 34–36, 91–93, *Glossary*: SAC

TD3 Twin Delayed Deep Deterministic Policy Gradient. 32–35, 83, 89, 90, 106, *Glossary*: TD3

TRPO Trust Region Policy Optimization. 32, 34, 68, *Glossary*: TRPO

Glossary

#Exploration is a method of penalizing already-seen areas in a more traditional Reinforcement Learning (RL) algorithm. In order to efficiently detect whether a state is in a densely-explored area, the state space is divided in bins which all have a counter. States are then penalized using reward shaping [Tang et al., 2016]. 36, 46, 52

CACLA is an class of actor-critic algorithms² that only use the sign of the Time-Difference error to determine the update to a stochastic actor, instead of using the exact error. The actor is only updated when the TD-error is positive [Van Hasselt and Wiering, 2007]. 93

DDPG is an actor-critic reinforcement learning that trains a deterministic actor, using the chain derivation rule to train the actor towards the best action under the current policy and critic values [Lillicrap et al., 2015]. 17, 19, 35, 56, 67, 74, 111, 130

DDPG-argmax is a variant of Deep Deterministic Policy Gradient (DDPG) in which the actor update is based on a sampled argmax operation instead of gradient descent. This alleviates some convergence problems at the cost of additional sampling [Matheron et al., 2019]. 91

DQN is an extension of Q-Learning to continuous state spaces, while retaining the discrete action space. Q is stored as a parametric neural network, and updated by performing a regression using the Q-Learning

²see our Taxonomy of Reinforcement Learning algorithms on page 32 for more information on actor-critic algorithms

rule [Mnih et al., 2013]. The main difference with DDPG, which uses a continuous actions space is the use of an explicit maximum operation, since Q can be evaluated for each possible action. 26, 34, 75

EST is a Motion Planning (MP) algorithm that explores a state space by prioritizing areas of the tree that have a low density, and expanding in a set disc around the chosen node [Hsu et al., 1997]. 45, 111, 131

Ex is a motion planning algorithm inspired by Expansive Spaces Tree (EST), and can build an exploration tree in a Markov Decision Process (MDP). It can be seen as a random walk that resets to a visited state when it encounters itself, and uses counters and spatial hashing so that each exploration step can be performed in $O(1)$ time [Matheron et al., 2020]. 17, 39, 44, 46–63, 68, 69, 111, 131

GEP is an algorithm that finds a set of diverse policies by performing a goal exploration process similar to Ex in an *outcome space* that is based on the policy [Forestier et al., 2017]. These diverse policies can then be used to bootstrap DDPG in a variant called GEP-PG [Colas et al., 2018].. 36

Go-Explore is a RL algorithm that trains a policy on a continuous MDP by using a two-steps approach. In a first step, a single valid path to the reward is found using an exploration algorithm. In a second step, this path is converted to a robust policy using backtracking. This algorithm was mostly tested on Atari benchmarks [Ecoffet et al., 2019]. 17, 67, 68

MDP A MDP is a discrete-time stochastic control problem. It can be defined as a tuple (S, A, P_a, R_a) in which S is the state space, A the action space, $P_a(s, s')$ is the probability that action a in state s will lead to state s' at the next time step. $R_a(s, s')$ is the immediate reward received for the same transition [Bellman, 1957]. 19, 38, 57, 70, 76, 105, 131

NSLC is an algorithm that finds a set of policies that are both diverse and perform well with respect to an objective. When two tentative policies are too similar, a secondary objective function is used to determine which to keep and which to forget. [Lehman and Stanley, 2011]. 36

PB is an two-part algorithm in which first, a valid trajectory is found using MP, then a second part inspired by Backplay [Resnick et al., 2018] helps the training of a RL algorithm by moving the starting point of

the environment backwards along this valid trajectory, until it reaches the original start of the environments. 17, 66, 74, 96, 112, 132

PBCS is an extension of Plan, Backplay (PB) which detects when the RL process is stuck or has diverged, and is able to restore it to a sane state, and exploit the progress achieved to reduce the problem to a simpler one and solve it recursively [Matheron et al., 2020]. 17, 96, 112

PPO is a class of RL algorithms that perform a policy gradient descent while limiting the change in policy at each training step by applying a penalty depending on the difference between the parameters of consecutive policies [Schulman et al., 2017]. 34, 67

Q-Learning is a Reinforcement Learning algorithm suited to finite MDPs, and is always able to find an optimal policy on such environments [Watkins, 1989]. 19, 26–29, 32–34, 130

RRT is a MP algorithm which biases the expansion of a search tree towards large unexplored areas by uniformly sampling targets in a predefined space, and expanding from the tree node that is nearest the target [Lavalle, 1998]. 11, 23, 31, 38, 66, 111, 132, 134

RRT-NH is a variant of Rapidly-exploring Random Tree (RRT) which is more suited to RL and non-holonomic problems because the expansion is done with random actions instead of trying to determine the best action to advance towards the target state [Lavalle and Kuffner, 2000]. 11, 40, 56

SAC is a RL algorithm that trains a stochastic actor while ensuring it does not converge to a deterministic policy by adding an entropy maximization constraint [Haarnoja et al., 2018]. 35, 91

TD3 is an expansion of DDPG which adds several features that speed up convergence and increase stability, namely the use of two critic networks, less frequent actor updates and adding noise to the actions used for training the critic network [Fujimoto et al., 2018b]. 35, 83

TRPO Is a RL algorithm that improves on simple Policy Gradient by estimating bounds for the size of each gradient descent step, and is able to guarantee monotonic improvements under certain conditions [Schulman et al., 2015]. 34, 68



Appendix: tooling and other contributions

A.1 Reinforcement Learning toolchain

In our work, we used Tensorflow with Python 3.7 and Python 3.8. It was run on Ubuntu and we used this workflow to run experiments on a set number of different parameters or seeds:

1. Create a single script that takes all of the relevant parameters as arguments
2. Create a BASH script `auto.sh` that generates a list of all commands to run. Each command has different parameters, for instance `seed=3`, `algo=ddpg`, and an *experiment name* such as `ddpg_03`. Each command is stored in a separate bash script in the `commands` directory, for instance `commands/exp_ddpg_03.sh`.
3. Run script `auto.sh` that generates the list of experiments to perform.
4. Use a *Makefile* to run parallel experiments using the `-j` argument of `make`. Each command can either pipe its log to a separate file that is used as a target for Make, or produce a `csv` or `pkl` results file which can be used as a target for Make.
5. Once each experiment is complete, use a python script to read all the results and either produce the figures or aggregate them in a single file which can then be exploited to produce figures.

Another popular Python library for Reinforcement Learning (RL) that we investigated is PyTorch [Goldsborough, 2018].

A.2 Gazebo simulator

All of the experiments which are presented in this document use either a custom simulator (for extremely simple environments such as 1D-TOY or mazes), or the MuJoCo simulator [Todorov et al., 2012]. Other options have been surveyed by Ivaldi et al. [2014], but the main criticism of MuJoCo is its "soft" simulation of contact forces, that benefits RL approaches by smoothing forces and observations.

An early project of my thesis was using the ROS Gazebo simulator with RL algorithms. Gazebo is well known in the Motion Planning (MP) community for simulating the behavior of actual robots, therefore testing RL algorithms on Gazebo is a first step to bridge the two domains.

Most RL algorithms are implemented in Python, therefore we implemented a Gazebo module that accepts commands in the form of Protobuf Github [2020d] messages on a socket and opens an API to Gazebo that can be used from Python. This interface allowed for about 400 simulation steps per second on a typical computer.

Other projects also attempt to run RL algorithms on simulated environments using Gazebo [Github, 2020a,c; Samir, 2016].

A.3 RRT pitfalls

During our research, finding a good Rapidly-exploring Random Tree (RRT) implementation proved harder than expected. The main time-complexity of RRT comes from the `nearest_neighbor` function, and is usually solved using a data structure called a k-d Tree (approximate methods also exist). However, implementations usually do not re-balance the search tree regularly. If points were inserted in a random order, the average complexity would still be logarithmic (for a single lookup) in the size of the tree, however the structure of the insertion order in RRT causes catastrophic performance. We implemented our own version of RRT which re-balances the k-d tree when it detects an imbalance.

A.4 Analysis of 1D-Toy

In this section, we study a few properties of environment 1D-TOY. Modeling the behavior of stochastic rollouts (in this case a deterministic agent with ϵ -greedy noise) on discrete-time environments is usually intractable, however the 1D-TOY environment is simple enough to analytically study the experience generated by rollouts when the learned policy is saturated.

This analysis is not strictly required to our work in Chapter 6, but is an interesting brain teaser.

A.4.1 Probability of success for an unbiased random walk with sink

In this section, we model the behavior of an agent that performs actions sampled uniformly in $[-0.1, 0.1]$. The following definition formalizes this process.

Definition 3 (Unbiased random walk with sink). *An unbiased random walk with sink is a sequence of random variables X_i where $\Pr(X_0 = 0) = 1$, and $X_{i+1} = X_i + U_i$, where U_i has the Probability Density Function (PDF)*

$$u(x) = \begin{cases} 5 & \text{if } -\frac{1}{10} < x < \frac{1}{10} \\ 0 & \text{otherwise} \end{cases}.$$

However the states $]-\infty, 0]$ act as a sink: X_{i+1} is only defined if $X_i > 0$ (otherwise the reward is found and the episode stops).

Theorem 4. *For each $i > 0$, there exists a set of polynoms $P_i^{(k)}$ such that X_i has PDF $x_i(z) = \begin{cases} P_i^{(k)} & \text{if } \frac{k-1}{10} \leq z < \frac{k}{10} \end{cases}$.*

Proof. **Initialization**

$X_1 = X_0 + U_0$ therefore $x_1(z) = \begin{cases} 5 & \text{if } -\frac{1}{10} < z < \frac{1}{10} \end{cases}$ and $P_1^{(0)} = P_1^{(1)} = 5$.

Induction

For all i and k , let $Q_i^{(k)} = \int P_i^{(k)}$.

$$\text{Let } u(z - t) = \begin{cases} 5 & \text{if } z - \frac{1}{10} < t < z + \frac{1}{10} \\ 0 & \text{otherwise} \end{cases}.$$

Consider the hypothesis holds true for i . We use the relation $X_{i+1} = X_i + U_i$, however we only consider the cases in which $X_i > 0$. In order to achieve this, we add a normalization coefficient α and only integrate positive arguments for x_i .

$$\begin{aligned} x_{i+1}(z) &= \alpha \int_0^{+\infty} x_i(t) u(z - t) dt \\ &= 5\alpha \int_{\max(0, z - \frac{1}{10})}^{z + \frac{1}{10}} x_i(t) dt \end{aligned}$$

Let $z > -\frac{1}{10}$ and $k = \lfloor 10z + 1 \rfloor$. We then have $\frac{k-1}{10} \leq z < \frac{k}{10}$ and $k \geq 0$.

The integration interval can be split in three segments $[\max(0, z - \frac{1}{10}), \max(0, \frac{k-1}{10})]$, $[\max(0, \frac{k-1}{10}), \frac{k}{10}]$ and $[\frac{k}{10}, z + \frac{1}{10}]$.

$$\begin{aligned}
x_{i+1}(z) &= 5\alpha \left[\int_{\max(0, z - \frac{1}{10})}^{\max(0, \frac{k-1}{10})} x_i(z) dt + \int_{\max(0, \frac{k-1}{10})}^{\frac{k}{10}} x_i(z) dt + \int_{\frac{k}{10}}^{z + \frac{1}{10}} x_i(z) dt \right] \\
&= 5\alpha \left[\int_{\max(0, z - \frac{1}{10})}^{\max(0, \frac{k-1}{10})} P_i^{(k-1)}(z) dt + \int_{\max(0, \frac{k-1}{10})}^{\frac{k}{10}} P_i^{(k)}(z) dt + \int_{\frac{k}{10}}^{z + \frac{1}{10}} P_i^{(k+1)}(z) dt \right] \\
&= 5\alpha \left[Q_i^{(k-1)} \left(\max \left(0, \frac{k-1}{10} \right) \right) - Q_i^{(k-1)} \left(\max \left(0, z - \frac{1}{10} \right) \right) \right. \\
&\quad \left. + Q_i^{(k)} \left(\frac{k}{10} \right) - Q_i^{(k)} \left(\max \left(0, \frac{k-1}{10} \right) \right) \right. \\
&\quad \left. + Q_i^{(k+1)} \left(z + \frac{1}{10} \right) - Q_i^{(k+1)} \left(\frac{k}{10} \right) \right].
\end{aligned}$$

$\max(0, X - \frac{1}{10})$ is piecewise-polynomial therefore $x_{i+1}(z)$ is piecewise-polynomial.

Note that $z - \frac{1}{10}$ is either positive or negative on the whole interval $\frac{k-1}{10} \leq z < \frac{k}{10}$. Therefore, $\max(0, z - \frac{1}{10})$ is piecewise-polynomial with the same pieces as the rest of the polynomials, and x_{i+1} can be written as $x_{i+1}(z) = \begin{cases} P_{i+1}^{(k)} & \text{if } \frac{k-1}{10} \leq z < \frac{k}{10}. \end{cases}$

Computation of the normalization coefficient α

$$1 = \int_{-\infty}^{+\infty} x_{i+1}(z) dz = \int_{-\infty}^{+\infty} \alpha \int_0^{+\infty} x_i(t) u(z-t) dt dz = \alpha \int_0^{+\infty} x_i(t) dt$$

Therefore,

$$\begin{aligned}
\alpha &= \frac{1}{\int_0^{+\infty} x_i(t) dt} = \frac{1}{1 - \int_{-\infty}^{(0)} x_i(t) dt} = \frac{1}{1 - \int_{-\frac{1}{10}}^0 P_i^{(0)} dt} \\
&= \frac{1}{1 - Q_i^{(0)}(0) + Q_i^{(0)}(-\frac{1}{10})}
\end{aligned}$$

This shows that the coefficients of $P_i^{(k)}$ can be computed in efficient time using dynamic programming.

Computation of $Pr(X_i \leq 0)$

$$Pr(X_i \leq 0) = Q_i^{(0)}(0) - Q_i^{(0)}(-\frac{1}{10})$$

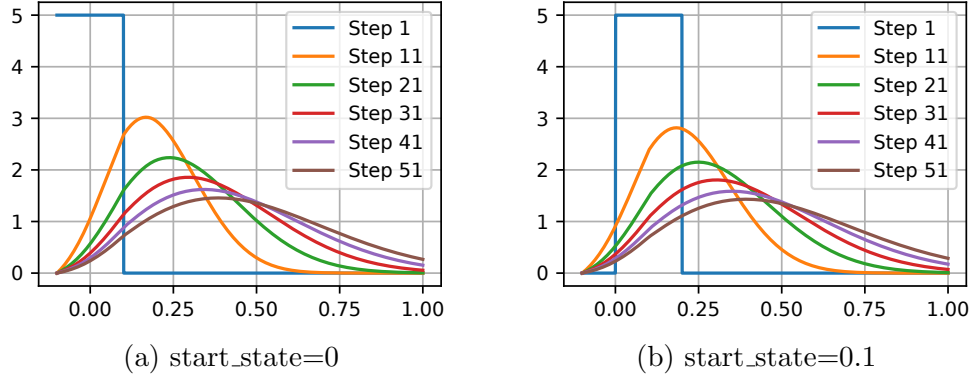


Figure A.1: PDF of state after n steps of the unbiased random walk with sink

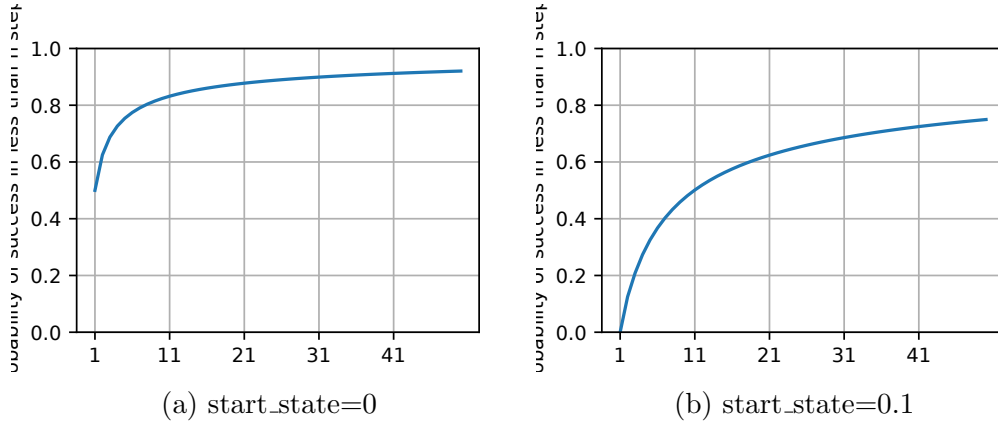


Figure A.2: Cumulative success rate after n steps of the unbiased random walk with sink

□

A.4.2 Probability of success for unbiased p -greedy random walk with sink

In this section, we change the problem to fit closer to the definition of 1D-TOY. We state that a random action is performed only with probability p , in other cases the state is unchanged. Since the initial state has a Dirac delta distribution, we can't compute it as a piecewise-polynomial. To alleviate this constraint, we assume that the agent is always greedy on the first step.

The definition of the new problem follows:

Definition 4 (Unbiased p-greedy random walk with sink). *We model the behavior of the agent as a sequence of random variables Y_i where Y_1 has PDF*

$$y_1(z) = \begin{cases} 5 & \text{if } -0.1 \leq z < 0.1 \\ 0 & \text{otherwise} \end{cases},$$

and for $i \geq 2$, $Y_{i+1} = \begin{cases} Y_i + U_i & \text{with probability } p \\ Y_i & \text{with probability } 1 - p \end{cases}$, where U_i has PDF

$$u(z) = \begin{cases} 5 & \text{if } -\frac{1}{10} < z < \frac{1}{10} \\ 0 & \text{otherwise} \end{cases}.$$

Theorem 5. *For each $i > 0$, there exists a set of polynoms $R_i^{(k)}$ such that Y_i has PDF $y_i(z) = \begin{cases} R_i^{(k)} & \text{if } \frac{k-1}{10} \leq z < \frac{k}{10} \end{cases}$.*

Proof. This can be noted by simply applying a transformation to P computed earlier.

For $k > 0$, $R_i^{(k)} = \gamma \left((1-p)P_i^{(k)} + pP_{i+1}^{(k)} \right)$ and $R_i^{(0)} = \gamma p P_{i+1}^{(0)}$. The normalization coefficient γ can be computed as $\frac{1}{\gamma} = 1 - (1-p) \left(Q_i^{(0)} - Q_i^{(0)} \left(-\frac{1}{10} \right) \right)$. \square

A.4.3 Sanity check

We ran simulations of the unbiased random walk with sink and unbiased p-greedy random walk with sink models, and compared the experimental results to the curves predicted by our analysis (Figures A.4 and A.3).

We then compared these results to the earliest reward found in the actual 1D-TOY experiments. Since the maximum episode length is 50 steps, and the actor is updated at the end of the episode, we expect to see this model match the unbiased p-greedy random walk (we showed the initial actor's output is very small). This claim is tested in Figure A.5.

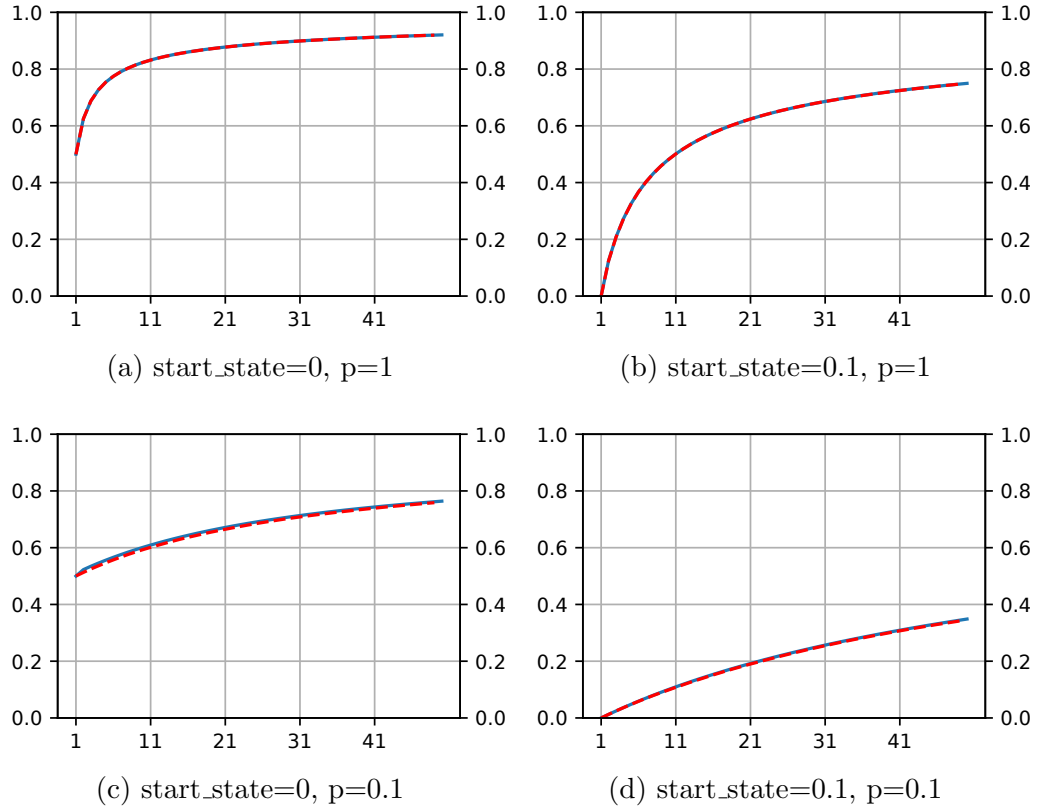


Figure A.3: Cumulative success rate after n steps of the p -greedy unbiased random walk with sink. Blue: analytical predictions. Red: simulation results ($N=10M$)

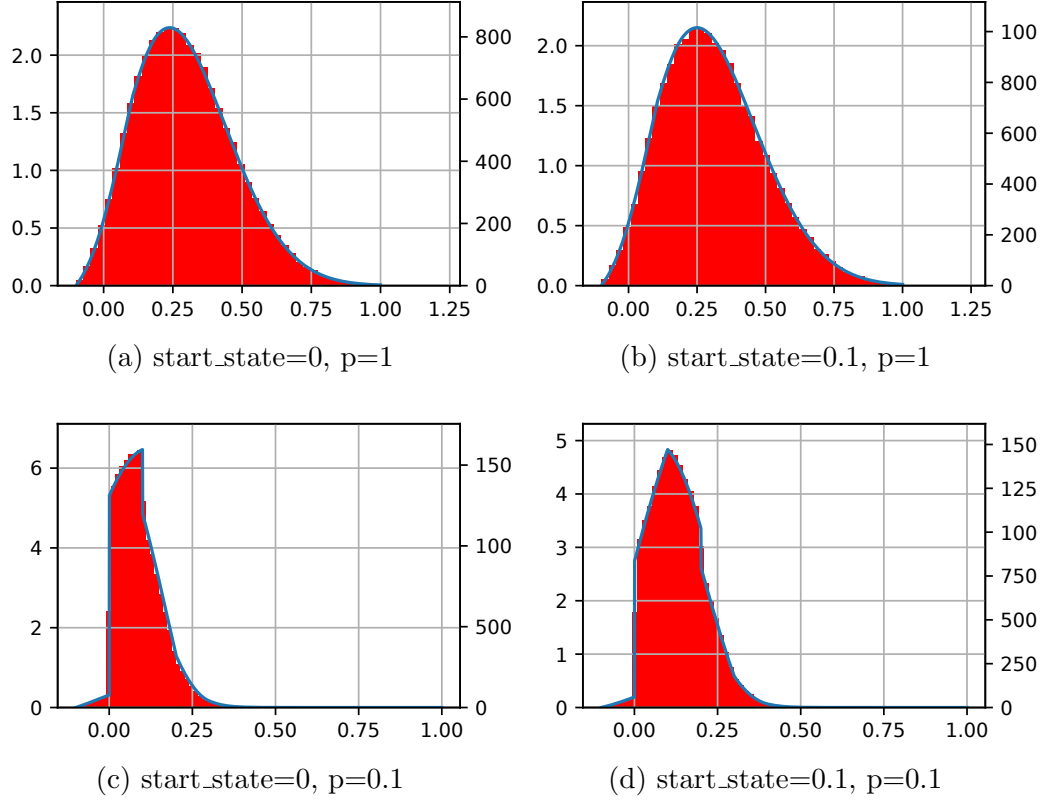


Figure A.4: PDF of state after 21 steps of the unbiased random walk with sink. Blue: analytical predictions. Red: simulation results ($N=10M$)

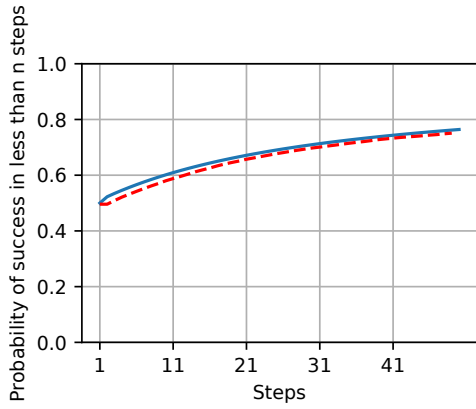


Figure A.5: Cumulative success rate after n steps of the 0.1-greedy unbiased random walk with sink. Blue: analytical predictions. Red: 1D-TOY results ($N=19k$). $\text{start_state}=0$