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

Guiding the Exploration Strategy of a Reinforcement Learning Agent

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

Hélène Plisnier

Promotors: Prof. Dr. Ann Nowé (Vrije Universiteit Brussel)

Print: Silhouet, Maldegem

2019 Uitgeverij VUBPRESS Brussels University Press
VUBPRESS is an imprint of ASP nv (Academic and Scientific Publishers nv)
Keizerslaan 34
B-1000 Brussels
Tel. +32 (0)2 289 26 50
Fax +32 (0)2 289 26 59
E-mail: info@vubpress.be
www.vubpress.be

ISBN 978 90 5718 448 2
NUR 958
Legal deposit D/2016/11.161/049

Home is where I want to be
But I guess I'm already there
I come home, she lifted up her wings
Guess that this must be the place
I can't tell one from another
Did I find you, or you find me?
There was a time before we were born
If someone asks, this is where I'll be

This must be the place, David Byrne

Abstract

Reinforcement Learning (RL) is a Machine Learning method mimicking the way humans learn to perform new tasks, specifically when it involves a great amount of trial and error. By nature, it is a progressive process that requires many interactions with the environment, and therefore time, before it can exhibit a satisfactory behavior. Consequently, an important challenge faced by current RL techniques is sample-efficiency. While most approaches to reduce the amount of samples needed focus on extracting a maximum amount of information out of each sample, we explore ways to improve the exploration strategy of the learner. Pushing the learning agent towards fruitful areas of the search space, and preventing it from wasting its time in undesirable areas, helps the agent reach a good policy faster and more efficiently. We present the Actor-Advisor, a general-purpose Policy Shaping method, allowing an external advisory policy to influence the actions selected by an RL agent. We extend our main contribution to a wide range of settings, such as discrete and continuous actions spaces, using on or off-policy Reinforcement Learning algorithms. We design the *learning correction* to let Policy Gradient-based methods benefit from off-policy external guidance, despite their strong on-policyness.

We evaluate the Actor-Advisor in two important RL sub-fields: learning from human intervention, and Transfer Learning. Although almost any source can be used as an advisor of an RL agent following the Actor-Advisor framework, the focus of this thesis is applying the Actor-Advisor to several novel Transfer Learning problems. Transfer Learning is resolutely related to sample-efficiency, since it aims at making the learning of new tasks faster by smartly reusing knowledge acquired in previous tasks. Finally, we introduce Self-Transfer, a learning trick inspired by Transfer Learning, in which an RL agent can easily improve its sample-efficiency by using an advisor pre-trained for a short while on the same task. We hope that our contributions will help promote the use of Reinforcement Learning methods in future real-life problems.

Acknowledgement

This is the uncomfortable and potentially cheesy part of this thesis. I'll keep it as short as possible, yet that does not mean I'm not extremely grateful to everyone mentioned below. Firstly, I'm thanking my two supervisors, Ann Nowé and Diederik Roijers for their unconditional support. They quickly understood my needs as a student, and provided me the freedom and sympathetic ear allowing me to work optimally. I also realize how lucky I am to have been welcomed by the Artificial Intelligence lab of the VUB. There, I have met people that I would have never met elsewhere, that are smart both intellectually and emotionally. I would like to thank, in no particular order, Tim Brys, Gill Balcaen, Pieter Libin, Johan Loeckx, Roxana Radulescu, Lara Mennes, Timothy Verstraeten, Arno Moonens, Eugenio Bargiacchi, Youri Coppens, Marta Marta De Lluvia, Mathieu Reymond (my Dungeons and Dragons friends), Audrey Sanctorum, Mathieu Reymond, Leander Schietgat, Jelena Grujic, Marjon Blondeel, Felipe Gomez Marulanda, Elias Fernandez, Isel Grau, Kyriakos Efthymiadis, Paul Van Eecke.

My family always stayed present throughout my life. When my older brother Aurélien and I were young, very little space was left for our dreams, aspirations and personality development, yet we managed to grow up as functional adults, supporting one another. My son, Cassandre Steckelmacher, objectively the most beautiful baby on earth, recently joined our family and helps me stay optimistic about the future. Finally, the most important person in my life, the person who is keeping me alive, really, is Denis Steckelmacher. Since we met, he made one of his lifetime goals to boost my self-esteem and confidence; we also tried to serve each other as personal supervisors on a day to day basis. His help, both on my work and my mental health, has been invaluable.

Acronyms

A3C : Asynchronous Advantage Actor Critic
ABCDQN : Agggressive Bootstrapped Clipped Deep Q-Network
BDPI : Bootstrapped Dual Policy Iteration
DDPG : Deep Deterministic Policy Gradient
DPD : Dual Policy Distillation
DQN : Deep Q-Network
GPSARSA : Gaussian Process State–action–reward–state–action
LC : Learning Correction
MDP : Markov Decision Process
PG : Policy Gradient
PI : Policy Intersection
PPO : Proximal Policy Optimization
PPR : Probabilistic Policy Reuse
RL : Reinforcement Learning
SAC : Soft Actor-Critic
TL : Transfer Learning

Table of Contents

1 | Introduction

Machine Learning is a sub-field of Artificial Intelligence, and can be divided into three main types of algorithms: Supervised Learning, Unsupervised Learning and Reinforcement Learning [Abu-Mostafa et al., 2012]. In Supervised Learning, the algorithm is provided with a set of inputs, and with each input, a corresponding output. The goal of the algorithm is to learn the function which produced the outputs given the inputs, in order to be able to predict the output of unseen inputs. In contrast, an Unsupervised Learning algorithm only receives the inputs, and attempts to cluster them in some way, allowing for patterns that may have been undetected by human experts to be discovered. Finally, a Reinforcement Learning algorithm takes an input, predicts an output, and then receives a scalar grading, or *reward*, indicating how well it did; the objective of the Reinforcement learner is to maximize the rewards it gets.

Reinforcement Learning (RL) is attractive when the solution to a problem is unknown or too difficult to implement by hand, but that it is possible to tell whether a solution is good or bad. It has already been proven useful in a wide range of applications, such as industrial and healthcare problems: order dispatching [Kuhnle et al., 2019], datacenter cooling [Lazic et al., 2018], personalized pharmacological anemia management [Gaweda et al., 2005], etc. Nevertheless, due to the progressive nature of the RL process and the need to sufficiently explore the solution space, most existing RL algorithms tend to require a prohibitive amount of samples to learn well, which makes them difficult to apply in real-life settings. To mitigate this problem, work has been pursued to extract as much value from each sample as possible in order to decrease the amount of samples needed [Schulman et al., 2015, 2017; Hessel et al., 2018; Steckelmacher, 2020].

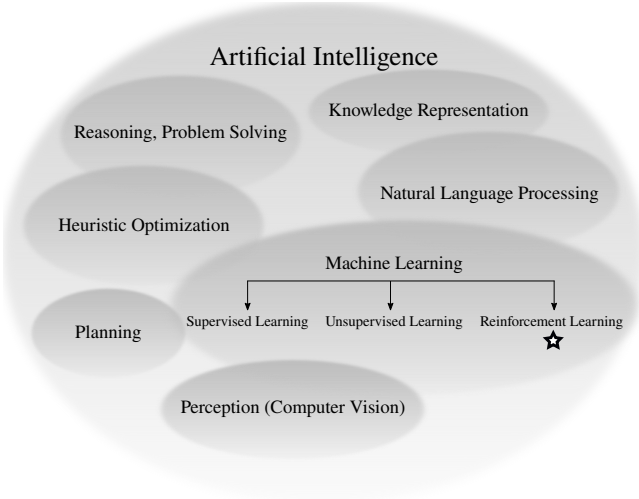

Figure 1.1: A non-exhaustive overview of Artificial Intelligence methods: this thesis focuses on Reinforcement Learning, a Machine Learning sub-field.

Another direction is to improve the exploration carried out by the algorithm by guiding it using some extra information; this approach belongs to the realm of Policy Shaping. In a Reinforcement Learning context, Policy Shaping methods let an external advisory policy influence or determine the actions chosen by the agent during action selection time. We use the term "extra information", which is deliberately vague, because many different sources can be used to advise the reinforcement learner: a human teacher [Griffith et al., 2013], a sub-optimal heuristic hand-coded by a designer, a proven-safe shield [Alshiekh et al., 2018], another reinforcement learner [Lai et al., 2020], etc. We are specifically interested in Transfer Learning approaches [Taylor and Stone, 2009], which exploit solutions learned in previous problems to learn new problems faster; this thesis mostly contains applications of Transfer Learning.

Settings compatible with Policy Shaping :

	continuous actions	discrete actions
critic-only	✓ (GPSARSA, fitted Q-iteration)	✓ (Q-Learning)
actor-only	✗ (PG)	✗ (PG)
actor-critic	✗ (SAC, PPO)	✓ (BDPI)

Settings compatible with Policy Shaping with our contributions :

	continuous actions	discrete actions
critic-only	✓ (GPSARSA, fitted Q-iteration)	✓ (Q-Learning)
actor-only	✓ (PG)	✓ (PG)
actor-critic	✓ (SAC, PPO)	✓ (BDPI)

However, leveraging external knowledge to guide an agent's exploration through tasks in the above-mentioned fields was not always feasible, due to fundamental algorithmic limitations. Some algorithms simply do not tolerate their action selection strategy to be directly influenced by an external force, such as actor-only or some actor-critic reinforcement learning algorithms. Yet, these algorithms tend to be the only ones applicable to tasks where the action space is continuous, as it is often the case for robotic tasks. As a result, the first part of this thesis (mainly, Chapter 3) extends a well-known Policy Shaping technique to make it compatible with state-of-the-art RL algorithms. Our main contribution, which we call the Actor-Advisor framework, is a general framework in which an RL actor is helped through its learning process by an advisor, using a Policy Shaping method. We dedicate Chapter 3 to the maximisation of the variety of problem settings in which Policy Shaping can be applied to improve the agent's performance.

The second part of this thesis is focused on applying our contribution developed in the first part to different problem settings, such as:

- learning from human intervention (Chapter 4)
- transferring knowledge across robotic platforms equipped with different sensors (Chapter 5, Section 5.2)
- transferring knowledge from different experts to one student (Chapter 5, Section 5.3)
- learning while transferring knowledge in parallel (Chapter 5, Section 5.5)
- self-transfer learning to easily boost sample-efficiency (Chapter 6)

Below, the reader will find a more exhaustive list of the contributions presented in this thesis. Most, if not all of the results listed revolve around our main contribution, the Actor-Advisor framework.

The Actor-Advisor with Policy Gradient Chapter 3, Section 3.1.2

Policy Gradient methods, a big family of RL algorithms, do not tolerate the

influence of an external force. However, these methods cannot simply be disregarded, as they are sometimes better suited than other RL algorithms to learn certain tasks. We present the learning correction, a method that makes Policy Gradient methods compatible with external guidance.

The Actor-Advisor with BDPI Chapter 3, Section 3.1.3
We implement a learning correction adapted to Bootstrapped Dual Policy Iteration (BDPI) [Steckelmacher et al., 2019], a very different algorithm than Policy Gradient-based algorithms. Although BDPI does not theoretically need our learning correction to accept external guidance, in contrast to Policy Gradient methods, we empirically show that its learning can still be improved thanks to it.

Extension of the Actor-Advisor to continuous action spaces Chapter 3, Section 3.2
Our Actor-Advisor is based on Policy Intersection [Griffith et al., 2013], a technique that guides the RL exploration strategy. However, originally, this method assumes discrete actions, significantly restricting its scope of application. We show two approaches to extend the Actor-Advisor to tasks with continuous actions.

Learning from a simulated human Chapter 4
We evaluate the Actor-Advisor in a learning from a simulated human teacher setting, on a difficult to explore grid-world with sparse rewards. We compare two teaching methods, advice and feedback, and show that teaching via advice requires significantly less interventions than providing feedback. In addition, we observe that the Actor-Advisor lets the Policy Gradient agent recover in the presence of misleading advice, and when the teacher suddenly stops giving advice to the agent.

Transfer between differently equipped drones Chapter 5, Section 5.2
We tackle a novel Transfer Learning setting, in which two Proximal Policy Optimization (PPO) [Schulman et al., 2017] agents controlling simulated drones with different sensors help each other learn the same navigation task. We compare the Actor-Advisor with transfer via shaping the reward of the agent in that setting, and show that our contribution significantly improves sample-efficiency at the beginning of learning in contrast to reward shaping.

Using multiple advisors to identify changes Chapter 5, Section 5.3
We introduce a method to allow several advisors to vote on which action should be taken by a fresh BDPI agent on a new task. This method is evaluated on a large grid-world with sparse rewards, and changing environmental dynamics.

Leveraging multiple advisors instead of one makes it possible to predict parts of the new task that are likely to vary from the previous tasks, and parts that are likely to be the same as before.

Transfer in an air compressor management problem Chapter 5, Section 5.4
We distill knowledge from several previously learned controllers into a new one while it trains on a new variant of an air compressor management setting. This approach outperforms simply loading an old controller, and significantly improves performance in the long run.

Transfer between multiple agents learning a task in parallel Chapter 5, Section 5.5
We introduce Shepherd, an RL agent as a service allowing the users of a web application to train PPO agents that automatically share knowledge with each other and help improve each other. We present a novel environment to evaluate the Shepherd framework on in future work: a tour guide web application designed for visitors of Brussels.

Self-Transfer to improve sample-efficiency Chapter 6
We introduce Self-Transfer, a learning trick for RL agents consisting in training an advisor agent on a task for a short while, then freezing it and using it to advise a fresh agent learning the same task until reaching a good policy. We observe that this simple idea results in a significant improvement in sample-efficiency of a Soft Actor-Critic (SAC) [Haarnoja et al., 2018] agent on multiple continuous environments, and of a BDPI agent on a difficult to explore environment with discrete actions mimicking a large office space.

2 | Background

Learning by reinforcement is a method constantly used by humans to develop intelligence, for a wide variety of tasks. Learning to ice skate for the first time, for example, is a pure learning by reinforcement task, that requires hours of practice to master. At a given instant, the ice skater pushes on their left foot, then senses their current balance on the blades, their speed and how close they are to the fence of the ice rink. Their action might slightly offset their balance, and lead to them falling down. Little by little, the ice skating beginner learns to stay longer on the ice without hurting themselves. In an Artificial Intelligence context, Reinforcement Learning (RL) is the appropriation by computer programs of that learning mechanism. Researchers and engineers design idealized situations, formulated as Markov decision processes (MDPs), in which the learner, called an *agent*, repeatedly selects an *action*, observes the current *state* of its *environment* after the action has been executed, and (perhaps) receives a *reward* (a numerical value) [Sutton and Barto, 2018, ch. 3]. The objective of the RL agent is to learn a behavior, or *policy*, that leads to the best rewards possible.

Some Reinforcement Learning algorithms are said to be *model-based* [Sutton, 1990], in the sense that they not only use a policy, but also a model of some sort of their environment. A model is a form of reversible access to the dynamics of the MDP representing the environment [Moerland et al., 2020], which allows the agent to plan forward before committing to a particular action. In contrast, *model-free* RL methods [Sutton and Barto, 2018, p.13 and 14] cannot predict how their environment will "react" to their action in advance. Model-free algorithms can be preferable to model-based ones when the environment is difficult to accurately model. In this thesis, we focus on model-free RL methods.

An important challenge in Reinforcement Learning is *exploration*. To learn a good policy, the agent must have had the chance to try out a potentially large number of actions and observe their consequences, otherwise the agent may stay stuck in a local optimum. However, not only may a satisfactory amount of exploration take time, but it will probably lead to the agent making mistakes, which, depending on the environment, can be disastrous. To make learning faster and less hazardous, there exist techniques that guide the exploration strategy of the agent towards fruitful, safer areas [García and Fernández, 2015]. The Actor-Advisor, our main contribution, is one such technique, and is based on Policy Intersection [Griffith et al., 2013], which we review in this chapter.

In the next section, we formally introduce Markov Decision Processes (MDPs). We then describe the RL algorithms mostly used in our experiments, namely Policy Gradient, PPO and Bootstrapped Dual Policy Iteration (BDPI). Note that we also leverage Soft Actor-Critic in some of our experiments, however, to harmoniously incorporate external guidance in their learning process, BDPI and Policy Gradient methods require the most important modifications, which are detailed in Chapter 3. Hence, we describe their inner-workings in this chapter, to better grasp our contributions to them. Typically, the choice of which algorithm to apply is done after having determined the problem to be solved. To our knowledge, there does not yet exist one algorithm that fits all reinforcement learning applications, hence why we could not stick to one algorithm for all our experiments, and had to diversify. With the adoption of different RL algorithms came the need to adapt our contribution, the Actor-Advisor (which we detail in Chapter 3), to the core mechanisms of these algorithms. An important trait that fundamentally impacts the structure of an RL algorithm is whether it is applicable to environments with discrete or continuous actions.

Most RL algorithms can roughly be categorized in two families: value-based algorithms [Watkins and Dayan, 1992], and policy-based algorithms [Sutton et al., 2000]. A policy search method explicitly learns a policy function π or *actor*, which, at each time-step t, takes a state s_t and samples an action $a_t \sim \pi(a_t|s_t)$ from its current state-dependent policy [Sutton et al., 2000; Peters and Bagnell, 2010]. In opposition, a value-based method learns a "quality" function Q, also called a *critic*, which takes state action pairs s_t, a_t as input, and estimates how much reward can be expected if a_t is chosen in s_t. At acting time, actions can be selected using a heuristic based on the Q function. Although value-based methods are not impossible to use in environments with continuous actions [Engel et al., 2005; Antos et al., 2007], computing the Q function becomes impractical. Policy Gradient, PPO and SAC are the preferred methods when dealing with continuous action spaces; BDPI uses both a policy function π and one or several Q functions, yet it is restricted to discrete actions.

Finally, we review in Section 2.5 two algorithms allowing an external policy to guide the exploration strategy of an RL agent: Probabilistic Policy Reuse (PPR) [Fernández and Veloso, 2006], and Policy Intersection [Griffith et al., 2013].

2.1 The Reinforcement Learning Setting

An environment, which lets the agent execute actions, observe states, and sometimes gives it rewards, can be formulated using a Markov Decision Process (MDP) defined by the tuple $\langle S, A, R, T \rangle$ [Bellman, 1957], with the time discretized in *time-steps*. At each time-step t, the agent takes an action $a_t \in A$ based on the current state $s_t \in S$, receives a new state from the environment $s_{t+1} \in S$ and a scalar reward r_t, returned by a reward function $R(s_t, a_t, s_{t+1}) \in \mathbb{R}$. For each state transition, a transition function $T(s_{t+1}|s_t, a_t) \in [0, 1]$ taking as input a state-action pair (s_t, a_t) returns a probability distribution over new states s_{t+1}; T is generally unknown to the agent. Both the actions set A and states set S can be finite or infinite. For instance, in a grid-world environment, which consists of a map of I rows times J columns, a state is typically described as the $i \in I, j \in J$ coordinates of the cell the agent is currently in; there are four actions at the agent's disposal: either going one cell up, down, left or right. Another example is a robot navigation task, in which the state can be the output of a distance sensor, and the agent sends a target speed to the controller of its motor(s). In the grid-world example, there is only a finite number of states and actions, whereas in the latter case, both the state and action spaces are infinite because continuous.

The amount of rewards received in the long run is formally defined as the discounted return $\mathcal{R}_t = \sum_t \gamma^t r_t$, where $\gamma \in [0, 1]$ is called the discount factor. The closer γ is to 1, the more the agent will prefer long-term rewards to short-terms ones. The agent selects actions based on its policy π, which maps a state to a probability distribution over actions, with $\pi(a_t|s_t) \in [0, 1]$ denoting the probability of taking action a_t given state s_t. The objective of an RL algorithm is to progressively modify π until the optimal policy π^* is found, which maximizes $E_{\pi^*}[\mathcal{R}_t]$, the expected discounted return [Sutton et al., 2000, ch. 3].

In this thesis, we focus on episodic tasks, in which time is broken down into *episodes*; typically, an episode ends and a new one starts once the agent has reached the goal, or once a given number of time-steps have passed. Similarly to playing multiple games of chess, episodic tasks assume an initial state $s_i \in S$ to which the agent is put back in at the beginning of each episode. The performance of an RL agent, and ultimately of the algorithm it executes, is evaluated mainly through looking at its *learning curve*. The reader will find multiple learning curves throughout this thesis; all of them show, for each episode, the sum of all rewards collected during the episode, see an example in Figure 2.1. The higher the curve goes, the better the policy learned

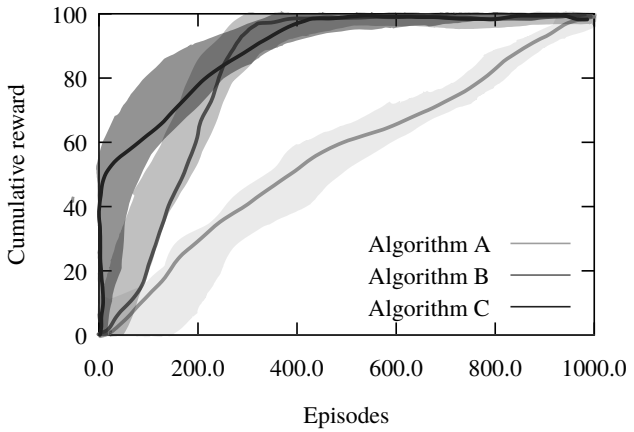

Figure 2.1: Three fabricated learning curves, each corresponding to an imaginary algorithm executed for N runs of X episodes (here, 1000) on an imaginary task. The curves shown in this thesis are obtained by computing for episode $i \in 1, 2, ...X$ the moving average of the mean over N episodic rewards. The shadows around the curves show the moving average of the standard deviation. Algorithm A and B learn a similarly good policy (given that the optimal episodic reward is 100), however, algorithm B is more sample-efficient than A, as it reaches the good policy faster than A. Algorithm C obtains even better results since it shows a significant *jumpstart* compared to A and B; the agent shows an almost good behaviour right from the beginning of learning. Jumpstarts like these tend to occur notably when the agent is helped by an external advisory policy, as it is the case in Transfer Learning settings, for instance.

is. In addition, an experiment often consists in several *runs* of an RL algorithm learning a task from beginning to end, for a number of episodes. In our results sections, we never show the learning curve of only one run; our curves are always averages of several runs, at least 4, often 8 runs per curve.

In addition to the value of the rewards obtained by the agent towards the end of learning, another important criterion to assess the quality of an RL algorithm is the amount of episodes, or more generally the amount of time, that the agent took to learn a good policy. This criterion is often referred as the *sample-efficiency* of an algorithm; each interaction the agent has with its environment consists in a data sample. The less samples the algorithm needs until reaching a policy achieving high rewards, the quicker the algorithm is at learning the task, which can determine whether or not the algorithm can actually be used in a given setting. Sample-efficiency is paramount

in robotic applications, for instance, in which each time-step can take up to several seconds.

Unfortunately, most RL algorithms need the agent to sufficiently explore the environment to manage to learn a good policy, which can in turn be costly in samples. Guiding the exploration strategy of an agent aims at making the exploration process more efficient, by advising it on which actions to select; this can potentially reduce the number of samples needed. The two methods we mainly evaluate in this thesis (see Section 2.5) are both under the Policy Shaping umbrella [Najar and Chetouani, 2021] (see Section 2.5), as they directly alter the agent's action choice.

2.2 Improving Exploration

In this thesis, we focus on a Policy Shaping approach to improve the exploration of the agent of its environment, and indirectly improve its sample-efficiency. Nevertheless, there exist other very different methods to improve exploration, as it consists in a general goal that can be approached from multiple angles; below, we briefly review two notable examples of such methods for the curious reader. One can design their agent to have an intrinsic motivation to perform a task, allowing it to generate its own rewards, other than the ones provided by the environment [Oudeyer and Kaplan, 2009]. In reinforcement learning literature, the intrinsic motivation of an agent to explore is often called *curiosity*. Curiosity-driven exploration techniques [Still and Precup, 2012; Frank et al., 2014; Pathak et al., 2017; Burda et al., 2018] form a fairly young RL sub-domain, and include visitation counts, which approximate the visitation frequency of states, and *novelty* or *surprise*, which rewards the agent when it visits less know states. A common method to implement the surprisal of an agent to see a given state is by leveraging a model of the environment, predicting the next state based on the previous state and action; the agent is trained to maximise the prediction error, hence favouring highly surprising states [Achiam and Sastry, 2017].

Properly balancing exploration and exploitation consists in a well-known reinforcement learning challenge; the agent explores by deliberately choosing actions that were not tried out yet in a given state, and it exploits by choosing actions it believes to lead to optimal rewards. In opposition to intrinsic motivation which aims attention on rewards, Thompson Sampling [Thompson, 1933; Russo et al., 2018] and Upper Confidence Bound [Garivier and Moulines, 2011] perform the action selection in a way that better adjust the amount of exploration of the agent.

Note that the above-mentioned techniques do not reappear in the rest of this thesis, as they cannot be used for applications such as learning from human intervention and Transfer Learning, in contrast to Policy Shaping.

2.2.1 The Challenge of Guiding Exploration Through Policy Shaping

Most RL algorithms can be decomposed into two main parts: an acting part, and a learning part. The acting part is executed at each time-step: it is when the agent selects an action to be executed according to its current policy or Q values; the process of choosing an action can be simply sampling a probability distribution over actions, or leveraging a sophisticated heuristic. The learning part, often called a *training epoch*, updates the policy or Q values based on the rewards obtained by executing actions, following an *update rule*. In contrast to the acting part, the learning part does not necessarily have to happen at every time-step, e.g., Policy Gradient updates its policy only after a whole episode has passed. In actor-critic algorithms, such as SAC and BDPI, there are actually two different updates per training epoch: the update of the critic, using the critic update rule, and the update of the actor, using the actor update rule.

To allow their exploration to be guided by an external advisory policy without jeopardizing learning, some RL algorithms require the advice produced by the advisor to be included not only in the acting part, but also in the learning part of the agent. The inclusion of advice in the acting part is often straightforward, and is already implemented by existing Policy Shaping methods (see Section 2.5). The addition of advice in the learning part, on the other hand, can be tricky, and heavily relies on the inner-workings of the RL algorithm, specifically its actor update rule. In addition, not all Policy Shaping methods are compatible with continuous actions, although many applications require a continuous action space. In Chapter 3, we extend our Actor-Advisor framework to allow Policy Shaping to be applied to algorithms that would not otherwise tolerate it, and we extend Policy Intersection, a well-investigated Policy Shaping technique, to settings with continuous actions.

When adapting an RL algorithm to Policy Shaping, one only needs to consider the actor update rule. Therefore, we are not concerned with the training of the critic, since it is the actor that is responsible of the agent's exploration strategy. In the next sections, we review the specific RL algorithms we use the most in our experiments throughout this thesis: Policy Gradient, PPO and BDPI. Note that we do not hold the ambition to teach the reader about these algorithms, but rather highlight the part in each of them that requires our attention when combining it with our contribution; this part is called the actor update rule.

2.3 Policy Gradient methods

Policy Gradient [Williams, 1992; Sutton et al., 2000] explicitly learns a policy π_θ parametrized by a weights vector θ; θ is updated at each training epoch $h \in \{0, 1, 2, ...\}$:

$$\begin{aligned}\theta_{h+1} &= \theta_h + \alpha \nabla \mathcal{L}^{PG}(\pi_\theta) \\ &= \theta_h + \alpha \nabla (-\sum_{t=0}^{T} \mathcal{R}_t \log(\pi_\theta(a_t|s_t)))\end{aligned} \tag{2.1}$$

where α is a small learning rate, $\mathcal{L}(\pi_\theta)$ is the *loss*, and $\mathcal{R}_t = \sum_t \gamma^t r_t$ denotes the discounted return at time t. Intuitively, the loss leads to the probability of past actions with the highest return to be executed more often in the future, leading to a constantly improving policy. Because the policy $\pi_\theta(s_t)$ is a probability vector that sums to 1, if the probability of one action is increased, the probability of the other actions is decreased. Concretely, at each time-step t, the agent takes the current state s_t as input, outputs a state-dependent policy $\pi_\theta(s_t) \in \mathbb{R}^A$ (a probability distribution over the set of actions A, summing to 1), samples an action $a_t \sim \pi_\theta(s_t)$ according to the policy, and receives a reward r_t from the environment after having executed a_t. The *experience tuple* (s_t, a_t, r_t) is appended to a list of experiences. Once the episode is finished, these experiences are used to compute new tuples $(s_t, a_t, \mathcal{R}_t)$, with $\mathcal{R}_t$ the return at time t, and the loss $\mathcal{L}^{PG}(\pi_\theta)$ is computed. Policy Gradient can perform only one policy update on a batch of experiences; after that, the batch must be discarded as Policy Gradient always requires fresh experiences to update its policy, which is relatively sample-inefficient. In all algorithms we use, the policy π is represented by a neural network, which adjusts the weights θ to minimize the loss.

Similarly to Policy Gradient, Proximal Policy Optimization (PPO) [Schulman et al., 2017] directly learn a policy π_θ, however it uses a different loss than the one in Equation 2.1 to update the parameters θ:

$$\begin{aligned}\theta_{h+1} &= \theta_h + \alpha \nabla \mathcal{L}^{CLIP}(\pi_\theta) \\ &= \theta_h + \alpha \nabla (\min(p_t(\theta)\mathcal{R}_t, \text{clip}(p_t(\theta), 1-\epsilon, 1+\epsilon)\mathcal{R}_t))\end{aligned} \tag{2.2}$$

where $p_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta\,\text{old}}(a_t|s_t)}$ is an *importance sampling* term, ϵ is a fixed parameter ($\epsilon = 0.2$ in [Schulman et al., 2017]), and $\mathcal{R}_t = \sum_t \gamma^t r_t$ is the discounted return, as in Equation 2.1 [1]. In PPO, the use of the importance sampling term $p_t(\theta)$ allows for

[1] In the original paper, the return $\mathcal{R}_t$ is replaced by an estimator of the advantage function $\mathcal{A}_t$ to improve learning stability. Here, we keep the return term in our equations for simplicity, but we invite interested readers to consult the original paper [Schulman et al., 2017] for more details.

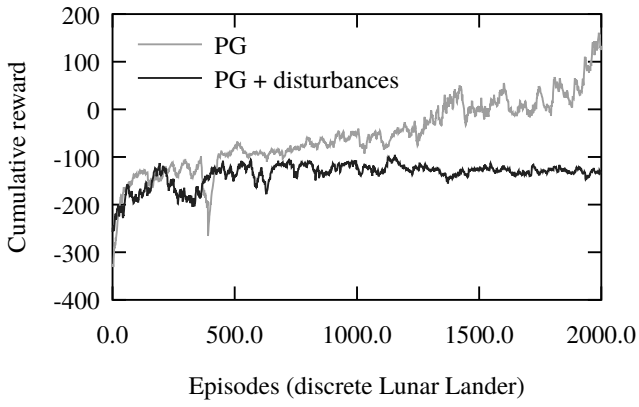

Figure 2.2: Policy Gradient refuses to learn the task if even a small amount of actions are not selected from its policy. The agent executes an arbitrary action instead of sampling its policy with a probability $\epsilon = 0.01$. These disturbances prevent Policy Gradient from learning Lunar Lander [Brockman et al., 2016] with discrete actions, a task it normally learns in approximately 2000 episodes.

multiple training iterations to be performed on the same batch of experience tuples, which increases sample-efficiency, in comparison to Policy Gradient. In a nutshell, the clip term encourages stable convergence by ensuring that $p_t(\theta)$ remains in the interval $[1-\epsilon, 1+\epsilon]$, which prevents the new policy π_θ from changing "too much" from the old policy $\pi_{\theta_{\text{old}}}$, during a training epoch. Taking a too large step away from the original policy can destroy the policy altogether.

In addition to whether it is value-based or policy-based, an RL algorithm can also be classified depending on whether it is *on*-policy or *off*-policy: "On-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data" [Sutton and Barto, 2018, p.124]. Policy Gradient is said to be strongly on-policy: it only converges if the actions executed by the agent are drawn from its current policy π_θ. This is illustrated in Figure 2.2: we force the agent to execute an arbitrary action other than one sampled from the current policy with probability $\epsilon = 0.01$, which is enough to prevent Policy Gradient from learning a task it is normally capable of learning. PPO does not show the same sensitivity to disturbances in the action selection as Policy Gradient does, perhaps because of its ability to update its policy multiple times on the same batch of experiences. Figure 2.3 shows the robustness

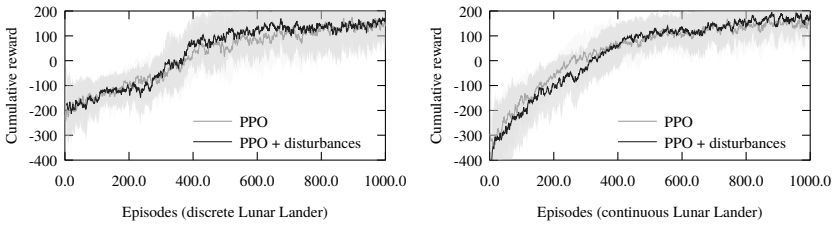

Figure 2.3: Proximal Policy Optimization, a sophisticated upgrade of Policy Gradient, does not present the same sensitivity to disturbances in the action selection as Policy Gradient does, whether it is facing an environment with discrete (*right*) or continuous actions (*left*). A very slight decrease in performance can be noticed in the continuous actions case at the beginning of learning, however, it is quickly compensated towards the end. The agent executes an arbitrary action instead of sampling its policy with a probability $\epsilon = 0.01$.

of PPO when the agent is forced to execute a random action instead of sampling its policy with a probability $\epsilon = 0.01$.

Guiding the exploration strategy of an agent often involves the intervention of external, off-policy advice at action selection time. Unfortunately, vanilla Policy Gradient is incapable of tolerating such interference, which disturbs its learning to the point of jeopardizing convergence. We present in Chapter 3 a solution to overcome this limitation, by modifying the loss of Policy Gradient to directly include external off-policy advice, without any convergence issue.

Although the definition of the loss may change, Policy Gradient is at the basis of most actor-only and actor-critic algorithms, such as PPO and SAC. A notable exception is BDPI, which uses a form of Conservative Policy Iteration [Pirotta et al., 2013] for its actor, instead of a Policy Gradient actor.

2.4 Bootstrapped Dual Policy Iteration

Bootstrapped Dual Policy Iteration [Steckelmacher et al., 2019, BDPI] is an actor-critic method, with one actor and multiple critics $N_c > 1$. Critics are trained using a value-based method called Aggressive Bootstrapped Clipped DQN (ABCDQN) [Steckelmacher et al., 2019], a version of Clipped DQN [Fujimoto et al., 2018]. Each critic maintains two Q-functions, Q^A and Q^B; at each training epoch, a batch b_i is sampled for each critic $i \in [1, N_c]$ from an experience buffer B. Then, the critics update their

Q^A function $N_t > 1$ times, and swap their Q^A and Q^B functions in between iterations. At each iteration, Q^A is updated using Equation 2.3 on the batch b_i.

$$\begin{aligned} Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) \; + \; & \alpha \left(r_{t+1} + \gamma \, V(s_{t+1}) \right) \\ & - Q_k(s_t, a_t) \\ V(s_{t+1})) = \min_{l=A,B} & Q^l(s_{t+1}, \text{argmax}_{a'} \, Q^A(s_{t+1}, a')) \end{aligned} \tag{2.3}$$

The actor π, the most interesting part to us, is trained using a variant of Conservative Policy Iteration [Pirotta et al., 2013]. Every training epoch, after the critics have been updated for a number N_t of times, the actor is trained towards the greedy policy of all its critics. This is achieved by sequentially applying Equation 2.4 N_c times, each iteration updating the actor based on a different critic (Equation 2.4 is applied as many times as there are critics).

$$\pi(s) \leftarrow (1 - \lambda) \, \pi(s) + \lambda \, \Gamma(Q^{A,i}_{k+1}(s, \cdot)) \qquad \forall s \in b_i \tag{2.4}$$

where $\lambda = 1 - e^{-\delta}$ is the actor learning rate, computed from the maximum allowed KL-divergence δ, and Γ the *greedy* policy. Given a state, the greedy policy returns the action believed to yield the best reward; $\Gamma(Q^{A,i}_{k+1}(s, \cdot))$ denotes the greedy policy according to the Q^A function of the i-th critic. By applying Equation 2.4, the actor is moved one small step towards the average greedy policy of the critics. To allow the actor to also be guided by an external advisor, in Chapter 3, we modify Equation 2.4 to explicitly include the intervention of the advisor.

2.5 Policy Shaping by Altering the Exploration Strategy

Existing Policy Shaping methods let an external advisory policy π_A alter or determine the agent's behavior at acting time, either sporadically or continuously throughout learning. Since π_A can be of any source, Policy Shaping consists in a general approach used in many different RL subfields, such as Safe RL [Alshiekh et al., 2018; García and Fernández, 2015; García and Fernández, 2019], learning from human feedback [Griffith et al., 2013], and Transfer Learning [Fernández and Veloso, 2006]. In this thesis, we are mainly interested in ways in which agents can transfer their policies to help each other learn new tasks faster (see Chapters 5 and 6), although we start in Chapter 4 with a learning from human intervention task.

Below, we review two methods to shape an RL agent's policy: Probabilistic Policy Reuse [Fernández and Veloso, 2006; García and Fernández, 2019, PPR], and Policy Intersection [Griffith et al., 2013; Cederborg et al., 2015, originally called "Policy Shaping"]. The first work we know of that introduced the term "Policy Shaping" is Griffith et al. [2013]. However, the method presented (which we detail below) is very specific, while the term Policy Shaping could apply to a wide variety of methods [MacGlashan et al., 2017; Harutyunyan et al., 2014]. Since Griffith et al. [2013] introduce an element-wise multiplication between discrete probability distributions, and that this operation represents taking the intersection between the distributions, we re-baptize Griffith's formula "Policy Intersection" in this thesis for clarity. Similarly, a "Policy Union" algorithm can be achieved by summing the probability distributions (which we non-exhaustively evaluate in Section 3.1.2).

2.5.1 Probabilistic Policy Reuse

First introduced by Fernández and Veloso [2006], Probabilistic Policy Reuse (PPR) samples an action at each time-step either from an external advisory policy $\pi_A(s_t)$ (or advisor) with probability ψ, or from the currently learned policy $\pi_L(s_t)$ (or actor) with probability $1 - \psi$:

$$a_t \sim \begin{cases} \pi_A(s_t) & \text{with probability } \psi \\ \pi_L(s_t) & \text{with probability } 1 - \psi \end{cases} \tag{2.5}$$

where $\pi_L(s_t)$ is the state-dependent policy learned by the agent, $\pi_A(s_t)$ is the state-dependent advice, and ψ is the probability of sampling an action a_t from the advisor π_A rather than from π_L. Even though PPR only *sometimes* executes an action from π_A, when it does so, π_A fully determines the action executed by the agent. Moreover, PPR has no provision for the advisor to choose, state for state, when to advise the agent and when to let it choose an action itself.

2.5.2 Policy Intersection

This second approach to Policy Shaping multiplies the π_L and π_A vectors, then normalizes the resulting policy vector at each time-step:

$$a_t \sim \pi_L(s_t) \times \pi_A(s_t) = \frac{\overbrace{\pi_L(s_t)\,\pi_A(s_t)}^{\text{element-wise product}}}{\underbrace{\pi_L(s_t) \cdot \pi_A(s_t)}_{\sum_{a \in A} \pi_L(a|s_t)\pi_A(a|s_t)}} \tag{2.6}$$

where $\pi_L(s_t) \cdot \pi_A(s_t)$ is the dot product of the two policies. At acting time, the agent samples an action a_t from the mixture $\pi_L \times \pi_A$ of the agent's current learned policy $\pi_L(s_t)$ and the external advisory policy $\pi_A(s_t)$, instead of sampling only from $\pi_L(s_t)$. The product $\pi_L(s_t) \times \pi_A(s_t)$ amounts to taking the intersection between what the current agent's policy π_L wants to do, and what the external advisory policy π_A would do in state s_t. As a result, although this formula was first introduced by Griffith et al. [2013] as "Policy Shaping", we re-name it Policy Intersection in this thesis for clarity.

Probabilistic Policy Reuse Versus Policy Intersection

We identify two desirable properties when combining the actor's policy with the advisor's policy:

1) the advisor must have a non-negligible power of influence over the advisee, especially when its advice is relevant;

2) the advisee must be able to ignore, or progressively learn to ignore the advisor's bad advice

Point 1 is motivated by the fact that, when faced with situations on which the advisor has some good recommendations, it is preferable that the actor effectively selects actions recommended by the advisor, and does not waste time exploring instead. Point 2 becomes paramount in cases where the advisor is not optimal. This is prevalent in Transfer Learning settings, in which the advisor has learned a first task, similar yet distinct to the second task the advisee is about to tackle. In such settings, the advisee must be capable of following the advisor when its knowledge about the first task is relevant to the second task, and ignore it otherwise. This is a particularly difficult challenge in the realm of Transfer Learning, which we propose a solution for in Chapter 5, Section 5.3. Nevertheless, a part of the solution resides in the method used to combine the actor's and advisor's policies.

PPR makes the action selection choice at a given time-step fully either the actor's or the advisor's. Regarding point 1 mentioned above, if the advisor happens to know better than the actor in a given situation, but it just so happens to not be "its turn", then the opportunity to exploit good advice is wasted. Similarly, if the turn is given to the advisor in a situation where its advice happens to be irrelevant, then the agent is bound to take a wrong action.

In contrast to PPR, Policy Intersection allows π_L and π_A to more cooperatively select actions, with π_A able to increase or decrease the probability of actions, but without fully determining the action. When sampling actions from the product of the actor's and advisor's policy, the agent is forced to select an action in the subset of actions that both the actor and the advisor agree on. The advisor cannot force the

agent to take actions that the actor has a zero probability for. Moreover, by adding, at each time-step, a very small non-zero probability to all zero probabilities of the advisor's vector, the actor can always push for the augmentation of the probability of an action that the advisor normally "does not agree with". At the beginning of learning, the actor is likely to have all its probabilities set to non-zero values, as its behavior is initially random. As long as the state of the agent's policy is to be open to anything, guiding it using an external source of knowledge is possible. As the advisor suggests actions to the actor, some of which revealing themselves to be bad, the actor adjusts its probabilities over actions according to the consequences these actions have on the environment. Towards the end of learning, the actor becomes deterministic , hence the agent stops listening to the advisor; in cases where the actor and the advisor disagree, the actor's probabilities are very high for one action, and almost zero for the ones the advisor has a high probability for.

Both PPR and Policy Intersection can be used to guide the exploration strategy of an agent, since they both directly alter its policy. Nevertheless, they let the advisory policy π_A get involved only at acting time, not in the learning part. Moreover, Equation 2.6 can only be applied to tasks for which there is a finite set of actions, excluding environments with a continuous action space. PPR, on the other hand, is trivial to use with continuous actions. Our main contribution, the Actor-Advisor, leverages Policy Intersection in the acting part, and manages to also include the advisory policy in the learning part of the agent if need be. As we illustrated in Section 2.3, Policy Gradient does not tolerate its actions to be sometimes overridden by an external force, which happens to correspond with the definition of Policy Shaping. In Chapter 3, Section 3.1.3, we empirically show that incorporating the advisory policy in the actor update rule can make the use of Policy Shaping with Policy Gradient possible, and can improve the stability of the learning process for BDPI. In addition, we extend Policy Intersection to environments with continuous actions in Section 3.2.

In Chapter 4, we use the Actor-Advisor to help a Policy Gradient agent learn a grid-world task while being advised by a simulated human teacher. After that brief venture into the realm of learning from human interventions, we focus all of our attention on various Transfer Learning tasks in Chapter 5: we notably introduce how not one but several advisory policies can intervene at the same time to help a fresh agent, how agents with different sensors can share their knowledge to learn the same task faster in a flying drone simulation, and how multiple users of a web-application can easily train the same RL agent. Finally, in Chapter 6, we present our last application idea for the Actor-Advisor to significantly improve the sample-efficiency of an agent, using a simple yet effective trick: Self-Transfer.

3 | The Actor-Advisor

The theoretical contributions presented in this chapter were first introduced in Plisnier, Steckelmacher, Brys, Roijers and Nowé, *Directed Policy Gradient for Safe Reinforcement Learning with Human Advice*, European Workshops on Reinforcement Learning (EWRL), 2018; Plisnier, Steckelmacher, Roijers and Nowé, *Transfer Reinforcement Learning Across Environment Dynamics With Multiple Advisors*, Benelux Conference on Artificial Intelligence (BNAIC), 2019; and Plisnier, Steckelmacher and Nowé, *Self-Transfer Learning*, Adaptive and Learning Agents (ALA) 2020.

We introduce this thesis main contribution, the Actor-Advisor, a Policy Shaping method that integrates external knowledge about which action should be taken in the Reinforcement Learning process, regardless of whether the action space is discrete or continuous. We provide all necessary extensions making the Actor-Advisor compatible to RL algorithms designed to deal with environment with discrete actions (e.g., BDPI) or continuous actions (e.g., Soft-Actor Critic); and that are either on-policy (e.g., Policy Gradient) or off-policy (e.g., BDPI). In the next chapters, we demonstrate the versatility of the Actor-Advisor by evaluating it to a variety of Reinforcement Learning problems, such as learning from a simulated human teacher, allowing simulated drones with different sensors to help each other learn the same task, and allowing multiple advisors to pitch in to solve one navigation task in a grid-world environment.

The Actor-Advisor is a framework in which a reinforcement learning actor policy is influenced by an advisor policy. We do not consider RL algorithms that do not include an actor in this thesis. Note that Policy Shaping techniques can still be used, and have predominantly been used with critic-only algorithms until now [Griffith et al., 2013; Fernández and Veloso, 2006]; however, the most sample-efficient and popular

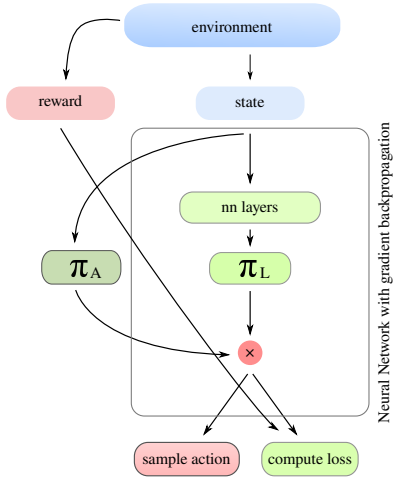

Figure 3.1: In the case of Policy Gradient, the Actor-Advisor involves the advisory policy both during the acting part and the learning part of the agent: the mixed policy $\pi_L \times \pi_A$ is sampled both for action selection (vanilla Policy Intersection), and when the loss or actor update rule is computed (learning correction). Note that the network training (or optimization) problem is not made harder by the use of π_A, since π_A is not fed to the trainable part of the network, is not parametric, and as such cannot be considered as an extension of the state-space. This neural architecture, inspired by how variable action-spaces are implemented in Steckelmacher et al. [2018], meets all the Policy Gradient and PPO assumptions, yet the behavior of the agent can be directly altered by an external advisory policy from any source.

algorithms at time of writing, such as PPO and SAC, happen to always include an actor, and very little work currently exists on Policy Shaping used with actor-only and actor-critic algorithms.

We divide our different extensions of the Actor-Advisor into two main sections: contributions that are specific to RL algorithms compatible with environments with discrete actions (see Section 3.1), and contributions that are specific to algorithms that are compatible with environments with continuous actions (see Section 3.2). This classification criterion is motivated by the fact that depending on whether the action space is discrete or continuous, the design of the RL algorithm used can fundamentally differ. Although the Actor-Advisor is a generic method with few assumptions, its implementation can be impacted by the underlying RL algorithm. The two main sections of this chapter are further decomposed as follows:

- Section 3.1: we present the two forms of discrete advice considered in this thesis, namely stochastic and deterministic (see Section 3.1.1); we adapt the loss of Policy Gradient and PPO, two on-policy algorithms, to harmoniously integrate off-policy advice without jeopardizing convergence (see Section 3.1.2); we adapt the actor-learning rule of an off-policy actor-critic algorithm, namely BDPI, and empirically show that the learning stability of the agent is improved (see Section 3.1.3;
- Section 3.2: we extend Policy Intersection, the Policy Shaping algorithm the Actor-Advisor is based on, to be used with Soft Actor-Critic (SAC) [Haarnoja et al., 2018], an off-policy algorithm designed for continuous actions spaces. In contrast to selecting an action when the actions are discrete, choosing a continuous action while being advised follows a completely different process, that must be carefully though out.

Policy Shaping techniques reviewed in Chapter 2 only intervene in the acting part of the reinforcement learning agent, i.e., they influence or determine which action is executed by the agent at action selection time. The Actor-Advisor, on the other hand, allows an external advisory policy to intervene both in the acting part, following the Policy Intersection formula (see Section 2.5), and learning part of the agent, using what we call a *learning correction*. Historically, this learning correction first came into existence when we used a Policy Gradient actor for the first time, and wished to guide its policy, using off-policy advice. Due to its strong on-policyness, this made Policy Gradient's learning diverge, until we designed a learning correction as detailed in Section 3.1.2.

In our experience, a learning correction is mandatory when using an actor-only on-policy algorithm (such as Policy Gradient), and can improve the stability of the learning of some off-policy algorithms (such as BDPI), although it is not required to learn [1].

3.1 Discrete Action Spaces

We first describe the two main shapes a state-dependent discrete advisory policy can take in the learning problems explored in this thesis. We then detail how we modified Policy Gradient, PPO and BDPI to let them incorporate off-policy advice harmoniously into their learning process.

[1] Off-policy RL algorithms do not need a learning correction, since overriding the policy generating the data does not jeopardize their convergence, in contrast to Policy Gradient methods.

3.1.1 Stochastic and Deterministic Advice

In the discrete actions case, the Actor-Advisor implements the Policy Intersection formula first introduced by Griffith et al. [2013], and detailed in Section 2.5.2 of Chapter 2. In that formula, the advice vector $\pi_A(s_t)$ is a probability distribution over actions summing to 1, and is either stochastic or deterministic, leading to distinct potential uses of the advice in real-world applications:

Stochastic advice ensures that each action has a non-zero probability, and can thus be selected by the agent with an arbitrarily high or low probability, depending on the agent's current policy π_L. Stochastic advice can be seen as an optional, *soft* advice, as it will merely bias the action selection rather than determining it. Stochastic advice can be used to allow a heuristic, a human teacher (in the case of suggestions, not commands), or a previously learned policy, in a transfer learning setting, to help the agent learn the task. In the above-mentioned cases, stochastic advice can be sub-optimal, hence it is desirable that the agent learns to ignore it whenever following the advice endangers its performance. A property we believe inherent to Policy Gradient and actor-critic algorithms, on top of which we implement the Actor-Advisor in the sections below, is that the higher the entropy of the advice, the easier it is for the agent to learn to ignore it. Our experiments in the next chapters tend to empirically confirm this intuition.

Deterministic advice has one 1 for a particular action, and a zero probability for all other actions. Deterministic advice can be used to enforce proven-safe mandatory guidelines that must be respected by the agent, in a safe RL task. In the case of a human user, it can also express the act of providing orders that, from the perspective of the user, must absolutely be executed, such as wanting their assistive robot to suddenly resume what it is currently doing. [2]

3.1.2 On-policyness and the Learning Correction

We were first introduced to Policy Shaping when working with an implementation of Policy Gradient, applied to environments with discrete actions. Policy Gradient [Sutton et al., 2000] is an actor-only Reinforcement Learning algorithm which explicitly represents and maintains a policy, instead of Q-Values. In contrast to Q-Learning-based algorithms, Policy Gradient does not need an extra exploration strategy to introduce exploration; directly sampling its actor naturally leads to sporadic exploration until convergence. Policy Shaping methods, on the other hand, consist in extra

[2] In a safety-critical application, such as a motorized wheelchair learning to navigate, a hierarchy between advisors might have to be considered, in which a proven-safe backup policy always has priority. This is to prevent cases in which the human user tries to make the wheelchair perform dangerous actions, such as driving down stairs.

exploration mechanisms by design, and have originally only been used with Q-Learning [Griffith et al., 2013; Fernández and Veloso, 2006]. Therefore, it is a counter-intuitive idea to want to leverage Policy Shaping with a Policy Gradient agent. More importantly, vanilla Policy Gradient methods will simply not allow for an exploration strategy to influence their action selection without convergence issues, due to their strong on-policyness. Our first extension of the Actor-Advisor architecture consists of a solution to this problem, allowing the policy executed by the agent to be directly influenced by an external advisory policy, without impairing convergence. We empirically show in Figure 3.3 how necessary the learning correction is for Policy Gradient to be able to exploit off-policy advice: a Policy Gradient is tackling Lunar Lander, Five Rooms or Cart Pole [Brockman et al., 2016] [3], while receiving advice at every time-step from a well-trained advisor on the same task.

We implement the vanilla Policy Intersection formula by Griffith et al. [2013] at acting time: an action to be executed is sampled from the mixed policy $\pi_\theta = \pi_{\theta L} \times \pi_A$ between the agent state-dependent policy vector $\pi_{\theta L}(s_t)$ and an advice vector $\pi_A(s_t)$ produced by the advisory policy π_A. At learning time, both the agent and the advisor produce their policy vector, $\pi_{\theta L}(s_t)$ and $\pi_A(s_t)$ respectively. The loss is then computed based on the mixed policy $\pi_\theta(s_t, \pi_A(s_t)) = \pi_{\theta L}(s_t) \times \pi_A(s_t)$, instead of on $\pi_{\theta L}(s_t)$ alone:

$$\begin{aligned}
\theta_{h+1} &= \theta_h + \alpha \nabla \mathcal{L}^{PG+Advice}(\pi_\theta) \\
&= \theta_h + \alpha \nabla (-\sum_{t=0}^{T} \mathcal{R}_t \log(\pi_\theta(a_t|s_t, \pi_A(s_t)))) \\
&= \theta_h + \alpha \nabla (-\sum_{t=0}^{T} \mathcal{R}_t \log(\pi_\theta(a_t|s_t) \times \pi_A(a_t|s_t))) \qquad (3.1)
\end{aligned}$$

This makes the Policy Gradient actor *aware* of the action it has actually executed in the environment, while being influenced by the advice at acting time, instead of updating its policy with action probabilities *that would be off-policy*. Including $\pi_A(s_t)$ in the loss satisfies the need of Policy Gradient to see the actions selected being sampled from the policy it is currently learning (as discussed in Chapter 2, Section 2.3); we meet this need by integrating the mixed policy in the Policy Gradient upgrade, to match up with the actions sampled from the mixed policy at action selection time.

We call the integration of the advice term $\pi_A(s_t)$ in the loss a *learning correction*, and it can be implemented in the clip loss of PPO, similarly to Equation 3.1. Instead

[3] Lunar Lander, Cart Pole, Pendulum and Acrobot are benchmark environments commonly used by the RL research community, made available by OpenAI [Brockman et al., 2016]. Five Rooms is our own custom environment, inspired by Four Rooms [Precup et al., 1998]

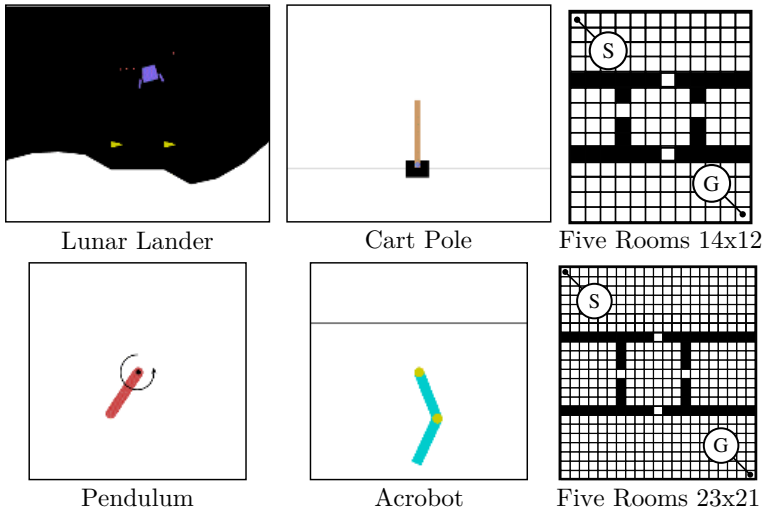

Figure 3.2: The benchmark environments used in this chapter to evaluate the learning correction. Lunar Lander for discrete and continuous action spaces, Cart Pole, Pendulum and Acrobot come from the OpenAI Gym library [Brockman et al., 2016]; Five Rooms is a custom grid-world environment inspired by Four Rooms [Precup et al., 1998]. *Lunar Lander:* The task is solved with rewards around 200. *Cart Pole:* An optimal policy receives rewards of 200. *Five Rooms* 14×12*:* An optimal policy obtains rewards of 75. *Pendulum:* The maximum reward is zero. *Acrobot:* Achieving the target results in a reward of 0. *Five Rooms* 23×21*:* The optimal reward is 57.

of computing the gradient solely based on $\pi_{\theta L}(s_t)$, it is computed based on the *combination* of the current policy $\pi_{\theta L}(s_t)$ and the advisory policy $\pi_A(s_t)$. An intuition of the impact of the learning correction is made more explicit in the modified actor update of Bootstrapped Dual Policy Iteration, which we present in the next section. This modified Policy Gradient loss allows the agent to learn even though its actions are influenced by the advisor, which would not be possible if the neural network was unaware of the existence of the advisor, due to Policy Gradient's strong on-policy nature.

Although the advantage of using the learning correction is clear with Policy Gradient, as illustrated on Figure 3.3 (left), it is less so when implementing it in PPO. We experimented the use of the learning correction on three different environments, all with discrete actions: Lunar Lander, Cart Pole, and a custom grid-world of 23×21 cells named Five Rooms. Out of the three environments, PPO seems to benefit from the learning correction only when facing Five Rooms. At time of writing, we

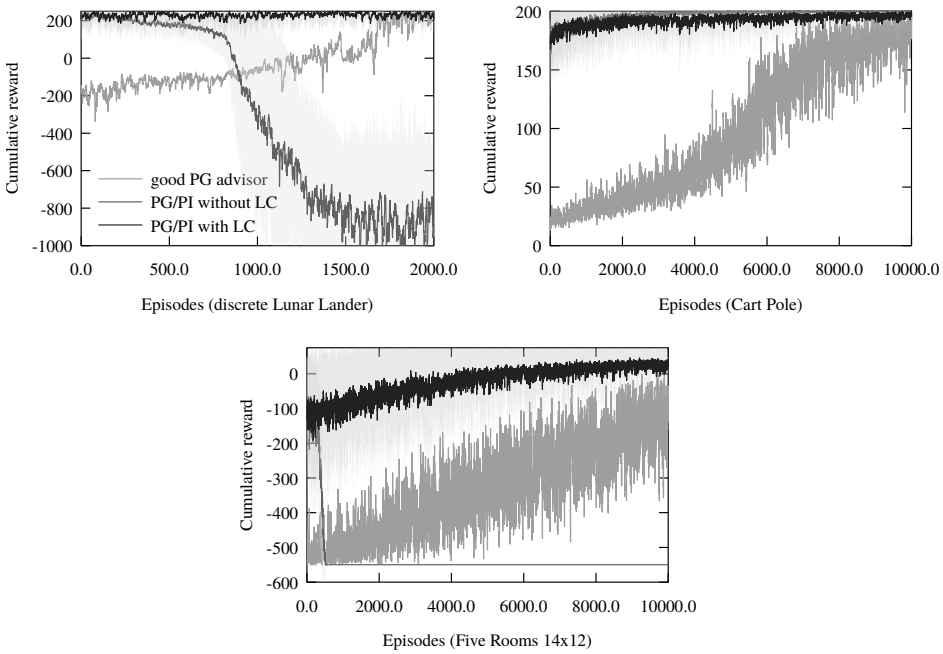

Figure 3.3: In Lunar Lander and Five Rooms, Policy Gradient needs the learning correction to be able to leverage the guidance provided by even a near-perfect advisor (Policy Intersection, "PI"), otherwise, its performances plummets. In Cart Pole, on the other hand, the use of the learning correction is not required by the agent to learn, nor does it jeopardize learning; both "PG/PI without LC" and "PG/PI with LC" curves overlap. We first train a regular Policy Gradient agent, without any external guidance, until it reaches a good policy. We freeze and store this agent to be used as an advisor. We then launch two fresh Policy Gradient agents, advised by the advisor : i) one using our learning correction to integrate the guidance from the advisor, and ii) one using vanilla Policy Intersection, without the learning correction. The learning correction allows agent i (top curve) to exploit the advice without issue, while the performance of agent ii (bottom curve) drastically drops.

do not have a well-constructed and convincing theory as to why the performance of PPO with or without the presence of a learning correction varies so drastically across environments. Nevertheless, a notable difference between Lunar Lander, Cart Pole and Five Rooms is that Five Rooms is a large, difficult to explore environment with sparse rewards. The agent starts in the top left corner of a 23 $\times$ 21 cells grid, and must find a goal cell in the bottom right corner; it receives a positive reward only when it has reached the goal, and -1.0 in all other states. In future experiments, we prefer to use a learning correction with PPO when dealing with particularly hard to explore environments with sparse rewards, but to not use it otherwise.

In the small experiment producing the results in Figure 3.3, the advisor and advisee are trained on the same task. However, most Transfer Learning settings seldom consider the transfer of a policy between two agents tackling the exact same task, unless they are solving the task in parallel [Lai et al., 2020], because there is simply not much point in doing so. Worthwhile Transfer Learning applications consider learning new, distinct tasks from previous ones, hence leveraging sub-optimal advisors which knowledge has become partially irrelevant. Similarly, learning from human intervention methods must take into account the inconsistencies and flaws embedded into advice from human teachers. As we explore these two RL subdomains in the next chapters, we empirically show how the Actor-Advisor allows the agent to learn to exploit advice when it is helpful and relevant, and to ignore it otherwise.

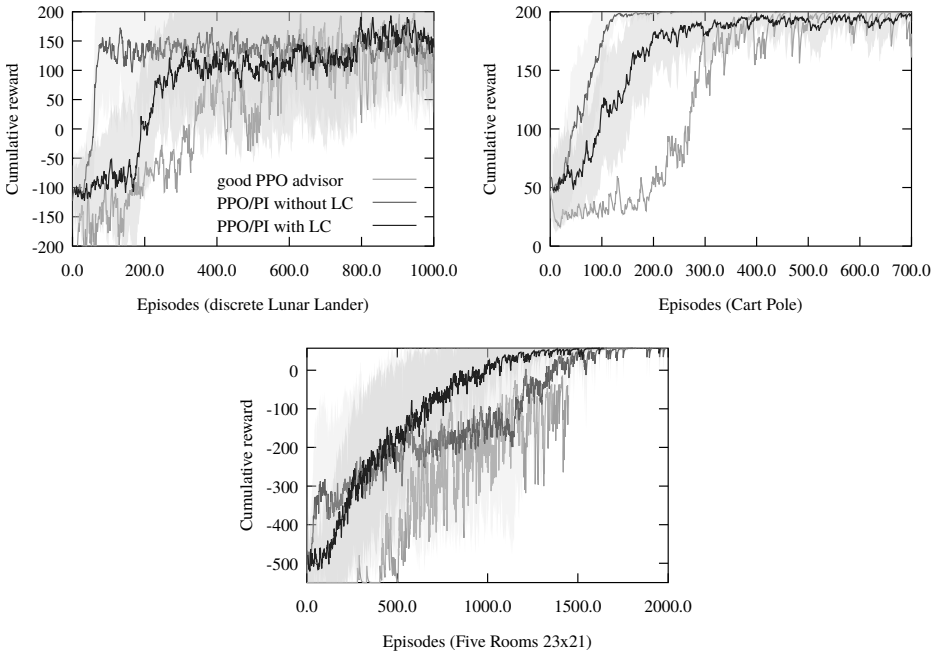

Figure 3.4: It is not clear whether the learning correction always helps the performance of PPO in the presence of an external advisory policy. We evaluate it on three different environments with discrete actions: Lunar Lander, Cart Pole, and a custom grid-world of 23×21 cells called Five Rooms. Although it leads to a decrease in sample-efficiency on both Lunar Lander and Cart Pole, the same learning correction leads to better results compared to not implementing it on Five Rooms. This drastic difference in results obtained might be due to the differences between the environments: Five Rooms consists in a large, difficult to explore environment with sparse rewards, in comparison to Lunar Lander and Cart Pole.

Policy Union Versus Policy Intersection

In Chapter 2, Section 2.5, we reviewed Policy Intersection, originally called Policy Shaping when first introduced by Griffith et al. [2013]. In the context of this thesis, we purposely refer to Equation 2.6 as an *intersection* between the actor's and the advisor's policies, since it performs an element-wise product between the two probability vectors. However, as the reader may guess, there exist other ways to combine probability vectors; one could sum both vectors instead of multiplying them:

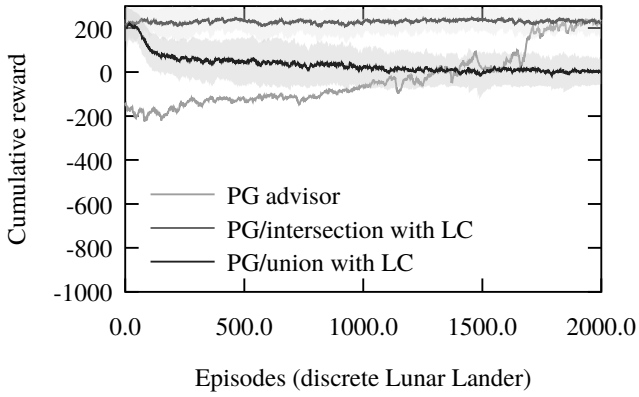

Figure 3.5: Both Policy Gradient agents using either Policy Intersection or Policy Union implement the learning correction, without which Policy Gradient's learning would diverge. The inferior performance of Policy Union is due to a too high exploration, instead of focusing on exploiting the advisor's relevant advice.

$$a_t \sim \pi_L(s_t) + \pi_A(s_t) = \frac{\overbrace{\pi_L(s_t) \; + \; \pi_A(s_t)}^{\text{element-wise sum}}}{\underbrace{\pi_L(s_t) \cdot \pi_A(s_t)}_{\sum_{a \in A} \pi_L(a|s_t) + \pi_A(a|s_t)}} \tag{3.2}$$

We refer to Equation 3.2 as Policy *Union*, as performing an element-wise sum of two probability vectors amounts to taking the union of their probabilities. In contrast to Policy Intersection, Policy Union broadens the agent's action choice instead of restricting it, and makes the resulting probabilities over actions become more uniformly distributed, in contrast to Policy Intersection. Even when the actor becomes deterministic towards the end of learning, the advisor forces the agent to keep on exploring actions it would normally take. This can be observed in Figure 3.5, where we compare a Policy Gradient actor being advised using Policy Intersection and one being advised using Policy Union. Both agents implement the learning correction. While Policy Intersection allows the agent to exploit the highly relevant advice it receives, Policy Union lets the agent perform useless exploration some of the time, preventing it from maximizing performance.

Another frequently used algorithm in this thesis, due to its inherently high sample-efficiency, is Bootstrapped Dual Policy Iteration. Below, we describe how we combine BDPI with the Actor-Advisor.

3.1.3 Bootstrapped Dual Policy Iteration Leveraging Advice

Although BDPI does not only use an actor, but also several critics, the only element that requires modifications to support the Actor-Advisor properly is the actor update rule, as well as its action selection strategy at acting time. Because the advice appears in the policy loss of the version of the Actor-Advisor implemented on top of Policy Gradient methods, we must consider how advice influences both *acting* and *learning* of the actor of BDPI.

BDPI with Advice at Acting Time

As with the Actor-Advisor with Policy Gradient, the BDPI actor remains purely state-dependant, and does not observe any form of advice. The actor therefore learns π_L, and has no extra input for advice. At acting time, the actions executed by the agent are therefore sampled from the mixture of the policy vector of the actor $\pi_L(s_t)$ with the policy vector of the advice $\pi_A(s_t)$, following the Policy Shaping formula shown in Equation 2.6.

Because BDPI is an off-policy algorithm, simply combining the output of the actor with the advice vector, and sampling actions from the result, maintains the convergence properties of BDPI. Nevertheless, we empirically show in Figure 3.6 that modifying the actor update rule to make it include advice in the learning mechanism increases the quality of the executed policy in the early stages of learning, and improves overall stability. At acting time, we therefore store the advice vector received by the agent at each time-step, which leads to the agent storing $(s_t, a_t, r_t, s_{t+1}, \pi_A(s_t))$ tuples in its experience buffer. We now discuss how the actor learning rule can leverage this advice to increase the quality of the actual behavior of the agent.

BDPI with Advice at Learning Time

While the BDPI critics play no role at acting time, they are updated, along with the actor, at learning time. Fortunately, the critics being off-policy, no special consideration is needed when learning Q^* from experiences generated by the mixture of the actor and advice. The actor, however, can benefit from an involvement of the advice in its learning mechanism, by implementing a learning correction similar to the one in the Policy Gradient loss.

Assuming that the advice is provided continuously (i.e., at each time-step), the objective is to maximize the expected cumulative reward obtained by *the agent*, that

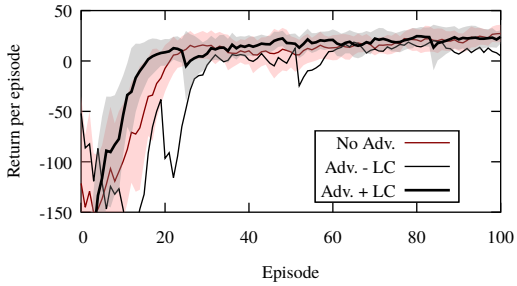

Figure 3.6: Applying a learning correction, that encourages the BDPI actor to diverge from the advice it receives, increases the quality of the policy in the early stages of learning. This figure shows how a BDPI agent performs without advice (No Adv.), with simple advice that encourages it to go forwards (Adv. + LC), and with that advice but no learning correction (Adv. - LC). Every curve show the average of 4 runs. The environment used is "Virtual Office" (see Section 6.3.2), a difficult to explore environment mimicking an office space, with fully continuous states and discrete actions.

executes a mixture of the BDPI actor *and* the advisor. We therefore want the actor to learn a policy that, *when combined with advice*, is optimal for the task. We formalize this objective in Equations 3.3 and 3.4, shown below, that we first introduced in Plisnier et al. [2019a]:

$$\pi(s, \pi_A(s)) \leftarrow \Gamma(Q(s)) \qquad \text{converges to the optimal policy} \tag{3.3}$$

Starting from Equation 3.3, we isolate π_L, the actor of BDPI, and derive an updated learning rule:

$$\begin{aligned}
\pi(s, \pi_A(s)) &\leftarrow \Gamma(Q(s)) \\
\frac{\pi_L(s)\pi_A(s)}{|\pi_L(s) \cdot \pi_A(s)|} &\leftarrow \Gamma(Q(s)) \\
\pi_L(s)\pi_A(s) &\leftarrow \underbrace{\Gamma(Q(s)) \times \overbrace{|\pi_L(s) \cdot \pi_A(s)|}^{\text{a scalar}}}_{\text{a vector}} \\
\pi_L(s) &\leftarrow \frac{\Gamma(Q(s)) \times |\pi_L(s) \cdot \pi_A(s)|}{\pi_A(s) + \varepsilon}
\end{aligned} \tag{3.4}$$

with the fraction an element-wise division between two vectors, and ε a small positive value that prevents a division by zero if the advice contains a zero probability for any of the actions.

Intuitively, in Equation 3.4, the actor learning rule moves the actor in the direction of the greedy function of a critic with a force or *pull* that influences how much the greedy function is followed, depending on how much it agrees with the advisor. The more the advice differs from the actor, the more the actor will follow the greedy function (and thus pull away from the advice). Such a pull allows the agent to compensate for bad advice, by learning an actor π_L that, when combined with the sub-optimal advice, still leads to a good policy. In the late learning stages, the Policy Gradient or BDPI actor becomes highly deterministic, and therefore defines (in the limit) the entirety of the behavior of the agent. In late learning stages, bad advice is therefore automatically overcome by the actor, which allows the optimal policy to be executed by the agent even with incorrect advice. Our learning correction improves the behavior of the agent in the *early stages* of learning, when the policy is not yet deterministic. By making it compensate for bad advice, we ensure a rapid improvement of the behavior of the agent, even when sub-optimal advice is used. We show in Figure 3.6 that the learning correction indeed increases the performance of the agent in the early stages of learning (after about 60 episodes), and provides a significant advantage over not using advice.

3.2 Continuous Action Spaces

Before presenting our extension compatible with continuous actions, we detail how the transition from a discrete action space to a continuous one alters the design of the Actor-Advisor. In the discrete actions case, the state-dependent policy vector $\pi(s_t)$ of the agent is a discrete probability distribution over actions, a vector of $|A|$ real values between 0 and 1, and that sum to 1. Such a discrete policy can be explicitly learned, with Policy Gradient for instance [Sutton et al., 2000], or computed on-the-fly based on learned Q values, using an exploration strategy.

In the continuous case on the other hand, enumerating the actions is impossible, as is producing an explicit probability density for each of them. Usually, the agent's policy is implemented as a Gaussian distribution, parameterized by a mean μ and a standard deviation σ, these two parameters being output by a neural network given a state s_t. At each time-step, a *single* action a_t is sampled from the Gaussian policy with mean μ and standard deviation σ. Continuous actions are challenging for Reinforcement Learning, and not every algorithm is compatible with them. For instance, almost all on-line RL algorithms for continuous actions need an explicit actor: there are very few critic-only algorithms for continuous actions [Engel et al., 2005; Antos et al., 2007].

3.2.1 The Actor-Advisor for Continuous Actions

We explore two ways to incorporate advice in the policy of a continuous actor. The first approach deals with combining the mean and the standard deviation of the actor μ_L and σ_L, with that of the advisor μ_A and σ_A; we experiment this approach with PPO. The second approach we propose merely assumes that actions can be sampled from the advisor, and that the actor can consider these actions and either "accept" or "reject" them, based on Gaussian distribution policy. We use Soft Actor-Critic (SAC) as our case-study for this second method, and report the results in Chapter 6, Section 6.2.

The Analytic Approach

In our first approach, we assume the advisor is parameterized by a mean μ_A and standard deviation σ_A. We average both means and sample actions from the Gaussian distribution parameterized with the new μ, following Equation 3.5:

$$\begin{aligned} \mu &= \frac{\mu_L + \mu_A}{2} \\ a_t &\sim \mathcal{N}(\mu, \sigma_L) \end{aligned} \tag{3.5}$$

Note that we keep the standard deviation σ_L from the actor unchanged. Preliminary experiments showed that averaging standard deviations similarly to the means yielded poor results. Equation 3.5 is our most satisfactory solution so far, although we plan on improving it in future work. Indeed, due to the sum performed, this method loses a property we find important: the actor cannot learn to ignore advice over time, in contrast to the original Policy Intersection formula (see Chapter 2, Section 2.5.2).

Incorporating the advice in the loss of PPO in the continuous actions space leads to mixed results, reported in Figure 3.7. Similarly to our observation made when applying PPO to several different environments with discrete actions in Section 3.1.2, the performance of PPO with or without the learning correction varies dramatically depending on the task. Pendulum and Lunar Lander with continuous actions require PPO to implement the learning correction for it to preserve good results, whereas the sample-efficiency of PPO on Acrobot is decreased when the learning correction is used. At time of writing, we do not hold a convincing theory explaining this behaviour; we merely report these results.

In the next section, we choose to use the Soft Actor-Critic algorithm instead of PPO, with another approach for computing the intersection between the actor's and advisor's policies when the actions are continuous. Since it has been introduced in Schulman et al. [2017] as an off-policy algorithm, in theory, SAC has no issue with

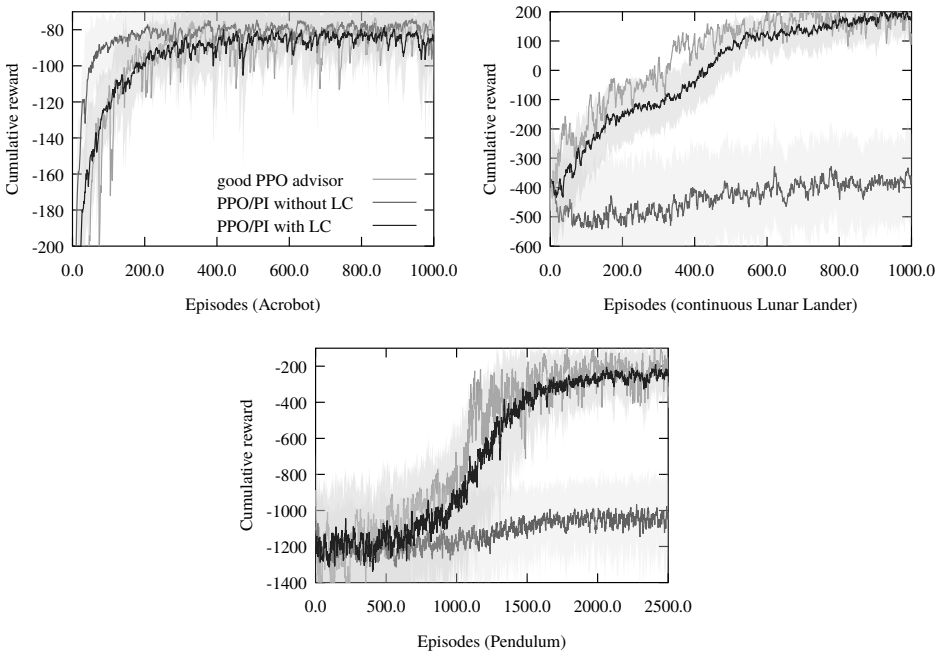

Figure 3.7: Again, the use of the learning correction with PPO leads to mixed results, depending on the environment. PPO is incapable of learning the continuous Lunar Lander and Pendulum tasks while using Policy Intersection without the learning correction (Policy Intersection, "PI"). However, the learning correction decreases sample-efficiency on the Acrobot environment, another task with continuous actions. The shaping of the PPO agent's policy is made possible by applying Equation 3.5.

having its action selection tampered with off-policy advice. As a result, we do not implement a learning correction for SAC.

The Sampling Approach

Equation 2.6 expresses the product of two state-dependent probability distributions: the actor's $\pi_L(s_t)$, and the advisor's $\pi_A(s_t)$. In the discrete actions case, $\pi_L(s_t)$ and $\pi_A(s_t)$ are both vectors of size $|A|$, where A is a finite set of actions available to the agent. Multiplying those two vectors amounts to taking the intersection between those two probability distributions. In other words, Policy Intersection samples actions that both the actor and the advisor "agree on", as these actions have a non-zero probability

Algorithm 1 Policy Intersection for Continuous Actions

Require: π_L is the currently learning actor, and π_A is the frozen actor used as advisor
  A^A = set of actions sampled from $\pi_A(s_t)$
  $P = [\,]$
  for every action $a_i^A \in A^A$ **do**
    Get $\pi_L(a_i^A|s_t)$
    Append $\pi_L(a_i^A|s_t)$ to P
  end for
  Create D = Categorical Distribution(P)
  Execute action $a \sim D$

in both $\pi_L(s_t)$ and $\pi_A(s_t)$ [4]. Unfortunately, when the action space is continuous, finding actions in the intersection between the actor's and the advisor's policies is less straightforward.

In Reinforcement Learning algorithms that can be applied to environments with continuous actions, such as SAC, individual actions can directly be sampled from the actor. Unfortunately, it is difficult to access a probability distribution over all possible actions, as the number of possible actions is infinite. However, the actor can provide, given a particular action, the probability density of that action. Assuming that the advisor can similarly be sampled, our approach to sampling the intersection between which actions are allowed by the actor and those allowed by the advisor is the following:

1. At action selection time, we sample a large amount of actions A^A from the advisor (where the superscript "A" stands for advisor).

2. We then submit each action $a_i^A \in A^A$ from the advisor to the actor π_L, which returns probability densities $\pi_L(a_i^A|s_t)$. These probability densities from the actor, for actions from the advisor, form a set P.

3. We create a new Categorical distribution D, parameterized with the probability densities P. This distribution can act as a policy that can be sampled for actions.

4. We execute in the environment an action sampled from D.

This method implements an "AND" between the actor and the advisor's policies. We basically consider that Policy Intersection computes the product of independent probabilities in the discrete-action case, note that it is equivalent to computing the

[4] Note that if no agreement can be made, i.e., the intersection between which action the actor and the advisor want to choose is empty, it can be arbitrarily decided that either the advisor or the actor has the last word, depending on the problem.

probability of the "AND" of two events, and implement that "AND" event in the continuous-action case, where the product is impossible to efficiently implement.

While our sampling technique allows to compute the intersection between an actor and an advisor, with no assumption about their distributions (we do not assume that they are Gaussian, for instance), we acknowledge that alternative approaches could have been devised, such as creating a new stochastic policy resulting of the point-wise multiplication of the advisor's probability density function with the actor's probability density function, sampled for a large number of actions. This method, however, requires an advisor that can produce density values for arbitrary actions, while ours merely requires an advisor that can sample actions.

The evaluation of this extension of the Actor-Advisor to continuous actions by sampling actions can be found in Chapter 6, Section 6.2, where we apply it to three challenging environments with continuous actions spaces.

3.3 Is the Learning Correction Always Useful?

In this chapter, we introduced the learning correction, firstly as a means to allow Policy Gradient-based methods to include Policy Shaping in their learning process. We show in Section 2.3 that if a few of the actions executed in the environment are randomly picked, and not selected according to the current policy of the Policy Gradient agent, then learning collapses. Again in this chapter, in Section 3.1.2, we show that the policy of a Policy Gradient agent tends to diverge, even when being advised by a very good advisor. Policy Gradient reacts badly to external forces choosing actions in its stead. However, Policy Shaping is, by definition, a family of techniques that tampers with the selection of actions, in order to steer the exploration of the agent. And, depending on the application, the use of a Policy Shaping method can greatly improve performance. Hence, the learning correction is first and foremost designed to make Policy Shaping possible for Policy Gradient. In addition, we observed that PPO can benefit from the learning correction in difficult to explore environments with sparse rewards.

When it comes to other reinforcement learning algorithms that are not based on Policy Gradient, such as Bootstrapped Dual Policy Iteration, the learning correction can be implemented, not because it is necessary for the learning algorithm to properly function in the presence of an external advisor, but because it might help the agent to fully benefit from the advisor. We see in Section 3.1.3, Figure 3.6, that using the learning correction with a BDPI actor can improve the overall performance of the agent in early stages of learning.

In the next chapters, when we refer to the Actor-Advisor, we imply that the learning correction is used by default. However, we regularly perform ablation studies to verify if the use of the learning correction still proves effective on different environments than

Lunar Lander, see for instance Figure 3.6 on Virtual Office, an environment described in Chapter 6, and 5.12 in Chapter 5 on a large grid-world environment. Although we certainly do not cover all existing reinforcement learning algorithms in this thesis, we at least empirically prove the benefit of using the learning correction for Policy Gradient, PPO and BDPI, in more than one environment.

4 | Learning from Human Intervention

This chapter is drawn from our publication, Plisnier, Steckelmacher, Brys, Roijers and Nowé, *Directed Policy Gradient for Safe Reinforcement Learning with Human Advice*, presented at the European Workshop on Reinforcement Leaning (EWRL), 2018.

Reinforcement Learning algorithms have the potential to unburden classical control engineers from having to develop sophisticated controllers for robotic applications. As suggested in Chapter 2, an RL algorithm is a compact, generic method that specifies *how to learn* a task, not *how to perform* it, leaving that chore to the RL agent. As a result, one RL algorithm can be used for a wide range of different tasks, instead of having to implement an individual controller per task. This feature could lead to the deployment of RL agents embedded on machines assisting humans in their daily-lives.

For the integration in human-populated spaces to be smooth, these agents should be able to quickly alter their behavior according to users' requests, to make them feel at ease and in control. Moreover, a non-negligible extra benefit of taking human input into account while learning is that it can potentially be used to speed up learning. It is this last point that motivates us to write this chapter. Learning from humans, and in human-populated environments, has enjoyed great interest from the Reinforcement Learning community over the past two decades [Jr. et al., 2001; Thomaz and Breazeal, 2006; Knox and Stone, 2009, 2010; Knox et al., 2012; Warnell et al., 2017; Griffith et al., 2013; Cederborg et al., 2015; Christiano et al., 2017; Mathewson and Pilarski, 2017; MacGlashan et al., 2017; Li et al., 2019; Najar and Chetouani, 2021, to only cite a

few]. Techniques have been developed for every form of signal a person can voluntary emit and enter in an AI system, such as providing feedback on the previously executed action, or advice on which action should be executed next, which behavior between two behaviors is preferred, or directly giving a demonstration of the desired behavior.

Arguably, some signals are easier to emit, and consist in more intuitive teaching tools than others [Li et al., 2019]. It is generally accepted that providing feedback is a fairly intuitively teaching technique for humans, in contrast to notably using demonstrations [Knox and Stone, 2009; Thomaz and Breazeal, 2006]; everyone can recognize success or failure, but only experts know how to best perform complicated tasks. Existing techniques often focus on learning from human-delivered feedback, i.e., rewards or critiques. This has been done either by shaping the environmental reward by mixing it with human reward [Thomaz and Breazeal, 2006; Knox and Stone, 2010] or totally ignoring the environmental reward and learning solely from human reward [Knox and Stone, 2009; Christiano et al., 2017; Jr. et al., 2001; Mathewson and Pilarski, 2017], or by shaping the agent's policy from human critiques [Griffith et al., 2013; MacGlashan et al., 2017]. Nevertheless, methods leveraging human feedback to shape the reward of the environment, or the actual agent's policy, can effectively improve the performance of the agent; however, it can only be used as a bias. If the agent has the intention to act undesirably, the human teacher only has the possibility to express their dissatisfaction *after* the agent's wrong-doing. Nevertheless, the teacher should not only be able to influence the agent's behavior and learning, but also to be able to immediately halt and correct the agent whenever it might engage in undesirable behavior, which is *by design* not possible using feedback-based techniques.

In this chapter, we apply the Actor-Advisor to the problem of learning from human-provided directives or advice. Specifically, we use the Actor-Advisor in this context as a Policy Shaping using human advice method, and evaluate it in an environment with discrete actions. This environment consists in a large, difficult to explore grid-world, in which Options [Sutton et al., 1999b] are used to allow our simulated human to give advice on a reasonable time-scale, as would be the case in a real-world application.

Additionally, for the experiment detailed in this chapter, the Actor-Advisor is implemented on top of a Policy Gradient agent, while methods guiding the exploration strategy have until now only been used together with algorithms based on Q-Learning [Griffith et al., 2013]. As detailed in Chapter 3, because the mixing in [Griffith et al., 2013] makes the actual behavior of the agent diverge from what it learns, thus off-policy, Policy Gradient cannot be applied as-is [Sutton et al., 2000]. Our contribution extends Policy Gradient to allow an advisory policy π_A to directly influence the actions selected by the agent, without impairing the convergence of learning. The Actor-Advisor allows the behavior of the agent to be immediately altered by an advisor, making it compatible with Safe Reinforcement Learning; allows the agent to leverage

advice to learn faster; is robust to mistakes in the advice; and does not need any insurance against sparse advice.

4.1 Existing Learning from Human Interaction Techniques

We now review existing techniques that are currently mainly used to learn from human intervention.

4.1.1 Reward Shaping

Feedback-based methods allow RL agents to learn from human intervention, one of the most famous ones being Reward Shaping [Ng et al., 1999]. Reward Shaping is a general method that allows the addition of a supplementary reward to the environmental reward signal that the agent already receives. The use of this method is motivated when the environmental reward is particularly sparse, or/and only one reward is provided at the very end of the task, which makes credit assignment difficult. In this case, shaping the reward received by the agent with a richer, more frequently emitted one can help speed the learning process. The shaping reward can be from many different sources; when it is applied to learning from human interventions, it is typically a measure of appreciation provided (in the form of a scalar) by the human in the loop. Learning from both human feedback and environmental reward can be achieved by simply summing them [Thomaz and Breazeal, 2006], or by using more advanced mixing strategies [Knox and Stone, 2010; Harutyunyan et al., 2015a].

In addition to allowing users to influence the behavior learned by the agent, human feedback can also help the agent learn faster [Jr. et al., 2001; Thomaz and Breazeal, 2006; Knox and Stone, 2009, 2010; Knox et al., 2012; Warnell et al., 2017; Griffith et al., 2013; Cederborg et al., 2015; Christiano et al., 2017; Mathewson and Pilarski, 2017; MacGlashan et al., 2017]. However, human feedback has been shown to be rich, but flawed and biased towards sub-optimal policies [Knox and Stone, 2010]. The rewards directly produced by the environment, and that theoretically encode the optimal policy, cannot therefore be ignored by the agent.

4.1.2 Human Imitation

The TAMER framework [Knox and Stone, 2009; Warnell et al., 2017] focuses on value functions, instead of policies or rewards. It considers that human teachers do not give a regular one-action reward, but an estimation of how fruitful the action is expected to be in the future, which is close to the definition of a value function. Hence, the

learning problem does not need to be solved by reinforcement any more, as a value function is directly provided to the agent. The agent's goal becomes to learn a model of the human value function, using supervised learning, and to myopically act by maximising this function. TAMER has been extended to continuous action spaces by using the human value-function in an actor-critic setting [Vien and Ertel, 2012]. The actor-critic setting also enables the use of advantage, i.e., how much better than expected an action is, instead of plain value functions, as it is achieved in COACH [MacGlashan et al., 2017]. Other work also considered human feedback as the sole reward source, and as a numerical value to be maximised [Jr. et al., 2001; Mathewson and Pilarski, 2017].

Nevertheless, a couple of issues of human imitation techniques have been identified. Firstly, they often assume that the human value-function is stationary in time, which has been observed not to be the case [Knox et al., 2012]. Secondly, the translation of human feedback (such as *good* or *marvellous*) to numbers is often arbitrary [Loftin et al., 2014; Griffith et al., 2013]. This led to work replacing the notion of value or reward with the one of *preference* [Christiano et al., 2017]. The human is shown trajectories produced by the agent, and which one is deemed the best is used to model a reward function.

4.1.3 Learning from Demonstrations

Human demonstrations consist of entire trajectories of states and actions, and are used to jumpstart learning in artificial agents with good initial policies [Peters and Schaal, 2008; Taylor et al., 2011]. In Taylor et al. [2011], a baseline policy is extracted from human demonstrations, then the agent autonomously learns using RL to outperform human performance on a simulated robot soccer domain. Demonstrations can also shape the agent's policy throughout its learning process [Nair et al., 2017]: an actor is trained off-policy with demonstration data, while a critic prevents the agent from blindly following the demonstrator's action when they are suboptimal.

The final approach we review in this chapter is Policy Shaping, which shares similarities with learning from demonstration techniques, as it allows a human to directly control the actions of the agent. However, Policy Shaping does not require the user to control the agent during a whole uninterrupted trajectory, from start to goal. Instead, the user is allowed to provide sparse advice, only when needed or when the operator is available, even if it is only for single time-steps at once.

4.1.4 Policy Shaping

Policy Shaping allows a human to advise the agent's actions, hence letting the human directly guide the agent's exploration strategy [Griffith et al., 2013; Najar and

Chetouani, 2021]. In Griffith et al. [2013]; Cederborg et al. [2015]; Sahni et al. [2016], the human feedback, expressed as binary positive or negative evaluations of the actions, is collected while the agent acts, and a critique policy is extracted from it. During action selection, this critique policy is then mixed with the agent's policy (following Equation 2.6), learned with standard variations of the Q-Learning algorithm. Policy Shaping has been successfully applied to 2D games [Griffith et al., 2013; Cederborg et al., 2015] and language games [Harrison et al., 2017].

Another application of Policy Shaping that does not rely on human feedback, which we prefer in our experiments and implement in the Actor-Advisor, is allowing the human teacher to directly submit actions to the agent, instead of first extracting a policy from collected human critiques. In our setting, at every time-step, the human provides a fresh advice to the agent with a probability L before any option has been selected. This advice can influence (or even determine, in the case of deterministic advice) the action selection. At training time, human directives are re-played and Policy Gradient automatically adjusts to them. Our method is, to our knowledge, the first allowing the use of Policy Shaping (or other guiding the exploration strategy method) in Policy Gradient. This is not only much simpler than human policies [Griffith et al., 2013], but also highly effective when the agent must urgently be prevented from acting undesirably, as we show in Section 4.3.2.

4.2 Challenges in Human Feedback

Except for learning from demonstrations, all the methods presented in Sections 4.1.4 to 4.1.1 allow for, or even rely on rewards or value functions. Rewards are either obtained from human feedback, preferences or critiques. Moreover, these rewards are used to slowly bias the policy of the agent towards human-sanctioned ones, and do not allow an immediate change of the agent's behavior if it is doing something wrong. On a more theoretical level, it has been observed that human teachers give feedback according to the agent's current policy [Kim et al., 2009; MacGlashan et al., 2017]. If the teacher is told that the learner is currently bad at a given task, they are more likely to deliver more positive feedback when the learner is succeeding at the task, than when they are told that the agent is already good at the task [MacGlashan et al., 2017]. This non-stationary feedback is a challenge for Reinforcement Learning algorithms designed for Markov Decision Processes (the reward function must depend only on the current state and action), and requires specific algorithmic precautions to be implemented [Harutyunyan et al., 2015b].

A bigger challenge is that, according to previous work analysing how people teach [Thomaz and Breazeal, 2006; Kaochar et al., 2011], humans actually seldom use feedback. Hence, offering only feedback as a teaching tool to users often leads to unex-

pected usage. For instance, teachers could communicate guidance through the feedback channel [Thomaz and Breazeal, 2006], because they want to tell the agent *what* to do, not whether it did well. Moreover, Reinforcement Learning algorithms slowly optimize their policy, which makes the impact of human feedback soft and progressive. As a result, teachers can feel like the agent is not listening to them.

Although these observations based on previous work seem to indicate that feedback-based methods should be avoided while teaching, there is a feature achievable via feedback-based methods, and that advice-based methods cannot reproduce. This feature is *teaching user preferences* to the agent. Despite the power of advice to immediately and visibly alter the action taken by the agent if the user commands it, an agent using policy shaping eventually learns the optimal policy, whether or not user directives align with this optimal policy. If following user preferences lead to a sub-optimal policy, the agent will learn to deflect from them whenever it can. Indeed, advice does not change the objective of the task; it merely improves exploration, allowing the agent to get to the goal quicker. Reward shaping methods, on the other hand, tamper with the objective function; they add extra rewards to the environmental reward signal, and thus can alter the overall goal of the task (unless the shaping reward is designed to be potential-based [Harutyunyan et al., 2015b]). As a result, rewards coming from the user can permanently teach the agent to respect their preferences while performing a given task, as they become part of its objective. On one hand, this presents the advantage of making the agent's behavior customizable by users; on the other hand, letting users tamper with the objective anticipated by the designer of the agent is not always desirable.

In summary, we believe that choosing a learning from human intervention method is a careful process that is strongly task-dependent. In our case, advice-based methods are preferred as we are primarily interested in increasing the agent's sample-efficiency at learning a complex task, which goal does not include respecting some user preferences. In this chapter, we compare our Actor-Advisor contribution, implemented on top of a Policy Gradient agent, to a simple Reward Shaping setting, in terms of amount of human interventions needed and final performance that can be achieved. We evaluate both methods on an environment with a discrete action space: a goal finding task in a large grid-world.

4.3 Helping an Agent Learn to Navigate in a Large Grid-World

The environment used to evaluate the Actor-Advisor learning from human intervention is Five Rooms, a 29×27 cells grid-world (see Figure 4.1a), inspired from the well-known Four Rooms environment used to evaluate options without human intervention [Sutton

et al., 1999b]. To simulate the human teacher, we handcraft a function providing the agent with advice or rewards (see Section 4.3.2).

Considering the sheer size of Five Rooms, making it hard for a Policy Gradient agent to explore, we let the agent use options to fasten navigation. Options have the additional and non-negligible benefit of being easier for a human to advise or give feedback on, in contrast to low-level actions which are often much less intuitive.

4.3.1 Options

To reduce the size of a prohibitively large-scale reinforcement learning problem, a complicated task can be divided into simpler sub-problems. The options framework [Sutton et al., 1999b] considers high-level actions, called options, which have their own execution policy over lower-level actions. An option o is defined as a tuple $\langle I_o, \pi_o, \beta_o \rangle$: an initiation set $I_o \subseteq S$ containing the states in which the option can start; a policy $\pi_o : S \times A \rightarrow [0, 1]$; and a termination function $\beta_o : S \rightarrow [0, 1]$ determining whether the option should end or not in a given state. The policy over options $\pi(o_t|s_t)$ is responsible for selecting a next option whenever one terminates.

Options present important advantages compared to flat Reinforcement Learning. On a technical perspective, it is sometimes possible for designers to hand-code options into an agent (or robot), since options can be smaller and simpler than the original complete task. In this case, the agent's goal becomes to learn the π policy over options, and not the π_o option policies, which can significantly simplify the learning process. Moreover, already-learned or designer-provided options can potentially be reused to tackle similar sub-problems. On a user-friendliness perspective, options may be easier to understand for humans, as they are high-level steps (such as opening a door) in comparison to low-level actions (such as actioning a single muscle motor). Because they are selected at a coarser time-scale, giving feedback or advice on options is also much easier than on low-level actions.

In this grid-world experiment, we use designer-provided options to illustrate the compatibility of the Actor-Advisor with options. Our agent learns the top-level policy π. Because our contribution can also be applied to flat Reinforcement Learning, and to simplify our notations, we denote π the policy being learned, and a_t the option selected at time t.

4.3.2 Evaluation on Five Rooms

Five Rooms is a custom [1] 29 cells high and 27 cells wide grid world, divided by walls. Each of the five rooms are accessible via one of the four one-cell-wide doors. This, in addition to its size, makes exploration difficult. In a conventional setting, the agent

[1] By "custom", we mean that we designed the Five Rooms environment.

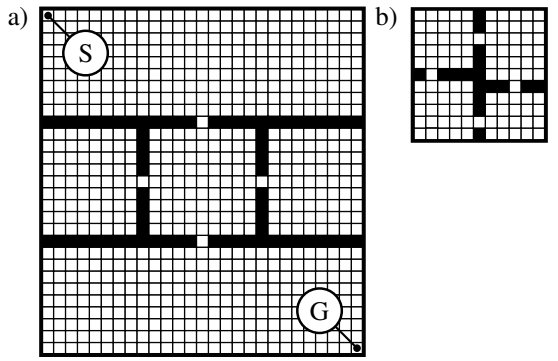

Figure 4.1: Comparison between a) our 29×27 Five Rooms environment on which we evaluate our method and b) the 11×11 Four rooms environment [Sutton et al., 1999b] which inspired Five Rooms. In environment a), black cells represent walls; the agent starts in the initial state S, and must reach the goal G.

can move one cell up, down, left or right, unless the target cell is a wall (then the agent does not move). The agent starts in the top-left corner of the grid, and must reach the bottom right corner, where it receives a reward of $+100$; it receives -0.1 in every other cell. The episode terminates either once the goal has been reached, or after 500 unfruitful time-steps. The optimal policy takes 54 time-steps to reach the goal, and obtains a cumulative reward of $100 - 54 \times (-0.1) = 94.6$.

Five Rooms is fairly large (in comparison to Four rooms in Figure 4.1b) and difficult to sufficiently explore, due to the sparse reward. As the original Four Rooms environment [Sutton et al., 1999b], and because of its sheer size, Five Rooms is well-suited to the use of options. Our use of a complicated environment, and options, is motivated by the nature of real-world robotic tasks in human-populated environments, that are big, complex, and are likely to already rely on options. Note that options are not necessary for the Actor-Advisor to work, but our experiments show, using this environment, that our method is compatible with Hierarchical Reinforcement Learning. We see this as an important benefit, as it is natural for humans to provide feedback on the level of options. We implemented 5 options: 4 door-options, each associated to one of the 4 doors, driving the agent directly to that door from any neighboring room, and then terminating. One option goes to the goal from any cell of the bottom room. Instead of choosing among the four up, down, left, right low-level actions, the agent's policy selects an option among these 5 ones. If the chosen option is defined for the agent's current position, the option leads directly to its respective destination. Otherwise, a random action is executed, after which the option immediately terminates

and returns control to the policy over options. As a result, the agent visits potentially every cell of the grid, instead of just hopping from door to door, keeping the problem challenging[2].

The Actor-Advisor Learning from Human Intervention

The Actor-Advisor learns from both the human teacher and the environment. Being aware of the environment allows the agent to learn the optimal policy defined by the environmental reward signal. By giving orders, the teacher helps the agent explore better, while having the possibility to deflect the agent's behavior before it engages in undesirable actions. In contrast to reward-shaping-based techniques, the Actor-Advisor does not rely on regular evaluations of how good the agent behaves, but on direct directions from the teacher. These directions immediately alter the behavior of the agent; we believe that this can procure a sense of power to the teacher, crucial in real-world applications [Thomaz and Breazeal, 2006]. Moreover, our experimental results in Section 4.3.2 demonstrate that the Actor-Advisor is able to leverage advice to learn faster, instead of merely obeying instructions and then learning as if nothing happened. If no advice is available, π_A is set to the uniform distribution, which cancels its effect and allows Policy Gradient to learn as usual. The network is trained using the standard Policy Gradient loss [Sutton et al., 2000], adapted for the use of advice, following our contribution in Section 3.1.2. We now demonstrate how much less human interventions are needed for the Actor-Advisor to be useful, compared to reward-based approaches.

Experiments

Because our main contribution, the Actor-Advisor, does not rely on rewards, values, classifiers or value functions, its properties are quite different from existing algorithms, that we review in Sections 4.1.1 and 4.1.2. Our experiments therefore thoroughly explore the properties of the Actor-Advisor, and are divided in three groups:

1. We compare the Actor-Advisor with policy shaping from a simulated human advisory policy, to simple Policy Gradient with reward shaping from a simulated human reward function, and to Policy Gradient without any human intervention (Section 4.3.2).

2. We quantify the impact on the Actor-Advisor of varying the hyper-parameters of Policy Shaping: the probability of giving advice L and the probability that the advice is correct $P(right)$ (Section 4.3.2).

[2]The task is much more challenging than if initiation sets 4.3.1 were used, as the agent has access to every option at any moment

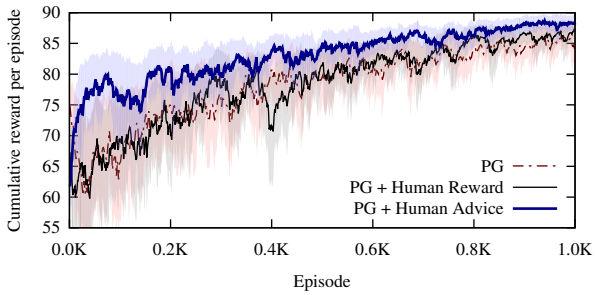

Figure 4.2: Comparison between Policy Gradient (PG) with Policy Shaping from human advice (PG + Human Advice) and Policy Gradient with Reward Shaping from human reward (PG + Human Reward). Both human advice and reward are given to the agent with a probability $L = 0.05$, corresponding to roughly 1000 human interventions for 1000 episodes. Human advice leads to the highest returns (curve on top), and improves the performance of the agent right from the start.

3. We consider a bounded amount of human advice, compatible with a real-world deployment of the Actor-Advisor, and show that the agent can still learn even if the amount of advice is heavily reduced (Section 4.3.2).

Simulated Human Advice

In our experiments, we use a function simulating a human teacher providing advice to the agent in a deterministic fashion. To design this function, we got inspiration from a small sample of actual people giving advice to a learner. Before the agent chooses an option, the human has the possibility of telling the agent which option to execute. This human advice is represented as a probability distribution over options, with a probability 1 for the option that the human wants the agent to execute, and 0 for every other option. The $\pi_A(s_t)$ advice is then mixed to the learned policy $\pi_L(s_t)$, as detailed in Section 4.3.2.

Because real humans may not always be present, or attentive, the human has a probability $0 \leq L \leq 1$ of providing advice to the agent, and a probability $0 \leq P(right) \leq 1$ of providing correct advice. A *wrong advice* consists in the teacher advising the agent to take the door leading to the middle left room, regardless of the agent's current location, which takes the agent away from the direct path to the goal.

The Actor-Advisor and Reward Shaping

In this section, we compare the Actor-Advisor to simple Policy Gradient with reward shaping, and Policy Gradient without any form of human intervention.

Our deterministic simulated human reward shaping works as follows, and is given to the agent only on some time-steps, with a probability of $L = 0.05$ (this leads to about 1000 human interventions over 1000 episodes). At time-step t, the agent chooses an option a_t. The human teacher has access to the agent's current position in the grid, and knows which option a^* is the correct one given that position. She allows (by giving 0) or punishes (by giving -5) the agent depending on whether $a_t = a^*$. This numerical human reward is then added to the environmental reward. Although it is not potential-based, this does not prevent the agent from learning the task; and making it potential-based [Harutyunyan et al., 2015b] would require modeling the human's reward function, which might alter it. Human advice is given to the Actor-Advisor with the same probability $L = 0.05$, which also results in about 1000 human interventions over 1000 episodes.

Results in Figure 4.2 show that, in the beginning, the performance of the agent is not improved by human reward; however performance seems to be slightly improved towards the end of the 1000 episodes (p-values are given in Table 4.1). On the other hand, the Actor-Advisor leads to much higher returns early on, and continues to dominate reward shaping and simple Policy Gradient until the end of the 1000 episodes. This experiment shows that, in contrast to reward-based methods, even a small amount of human advice significantly improves the agent's performance. The next experiments explores the behavior of the Actor-Advisor when various amounts of human advice is given.

Sensitivity to Hyper-Parameters

Different settings lead to varying kinds of human intervention given to the agent. In this experiment, we vary the value of L, the probability that the teacher gives advice, and $P(right)$, the probability that the advice is correct. We show that the most interesting results are obtained for values of L that are not too large. If L is too large, the agent is controlled by the human and learns to imitate her, but this is only useful if we consider the human to be the main controller, and the agent is only used to "fill in the gaps" when the human is unavailable. Hence, we keep the value of L small ($L = 0.2$ maximum) in order to fully emphasize the learning ability of the Actor-Advisor, in a setting where imitation learning is not enough.

Figure 4.3 compares various configuration of the Actor-Advisor, with a high or low probability of giving advice, and a high or low probability of giving correct advice. The main results are:

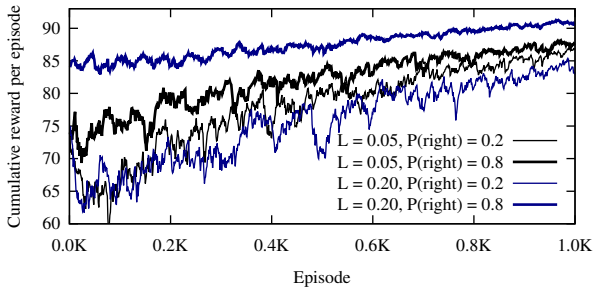

Figure 4.3: Comparison of the Actor-Advisor with small and large values of L (the probability of giving advice) and $P(rigth)$ (probability that the advice given is correct) As can be expected, a large L combined with a large $P(right)$ gives the best results (curve on the top). When $P(right) = 0.2$, the agent still manages to learn, regardless of the value of L, which demonstrates the robustness of the Actor-Advisor to incorrect advice.

1. When the advice delivered to the agent is correct, larger values of L lead to higher returns. Similarly and unsurprisingly, when incorrect advice is delivered, a small L allows the agent to overcome bad advice and still learn a good policy.

2. The agent always manages to learn a reasonable policy, even when most of the advice is incorrect ($P(wrong) = 0.8$), regardless of L.

The second point is the most interesting, as it demonstrates that the Actor-Advisor is robust to wrong advice. Adding advice to an agent is therefore free: scarce advice can only improve the performance of the agent, and wrong advice does not prevent the optimal policy from being eventually learned. The next experiment shows that the Actor-Advisor is even robust to non-stationary advice, contrary to most reward-based approaches (see Section 4.2).

Non-Stationary Advice

In many real-world settings, humans are not always available to help the agent learn. In this experiment, we interrupt advice after 700 human interventions. This assesses the robustness of the agent to advice that changes (abruptly) over time, and mimics human advice obtained over one or two hours of on-the-field training, before the human trainer leaves, possibly to set-up another agent in another company.

In Figure 4.4, the performance of the agent is very high while it is trained, as human advice can be directly followed, leading to high returns. This shows that

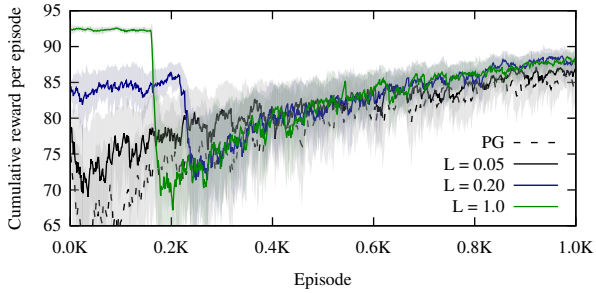

Figure 4.4: Advice is interrupted after 700 human interventions, with various probabilities of giving advice L, and a probability of giving correct advice $P(right) = 0.8$. An interesting phenomenon becomes stronger as L increases: the agent's performance drops when advice is removed (as the agent is on its own), but climbs back up afterwards, eventually outperforming Policy Gradient without human advice.

the trainer is able to completely override the agent's behavior, which is crucial for safe Reinforcement Learning. When advice stops, the agent's performance drops, then consistently recovers, and even surpasses Policy Gradient's performance without human intervention (as is suggested by the statistical significance test values in Table 4.2). This empirical result shows the robustness of our technique and suggests that Policy Shaping allows users to quit helping the learner whenever their interest fades: the agent will still be able to discover a good policy. More importantly, the final performance of the agent is higher when advice has been available at some point, which demonstrates that the Actor-Advisor learns from the advice, instead of merely following it. Interestingly, the more intensively the agent is helped (i.e., the higher the L), the better it recovers afterwards. In a real-world setting, intense and dedicated advice (but over a short period of time) therefore seems to be the way to go.

In contrast to reward shaping from human feedback, interrupting advice does not necessarily result in a performance loss in the long run. In Figure 4.5, we compare human advice, interrupted after 700 interventions, to reward shaping, interrupted after 10,000 punishments (so, much more than 10,000 time-steps during which the human had to watch the agent). Once the 10,000 human punishments have been consumed, the performance of Policy Gradient with human reward slightly decreases, then plateaus until the end. Using only 700 pieces of advice, the Actor-Advisor manages to learn a policy marginally better than the one obtained with reward shaping, even if reward shaping requires one order of magnitude more human dedication. Furthermore, the Actor-Advisor is simpler to implement (see Section 4.3.2) than reward

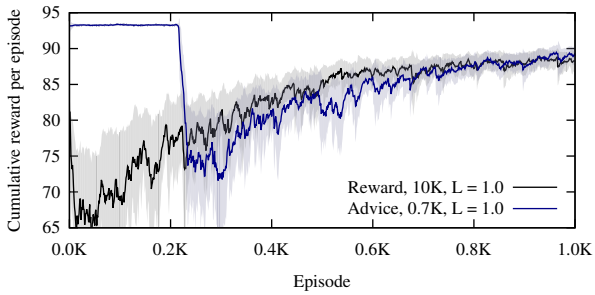

Figure 4.5: 700 pieces of human advice, with $L = 1$, versus 10,000 reward-based punishments, with $L = 1$. Even though a far higher amount of reward is provided, human advice eventually matches human reward.

shaping, which ideally requires a potential-based reward function to be designed [Ng et al., 1999; Harutyunyan et al., 2015b] to avoid biasing the learned policy.

Conclusion

In this chapter, we use the Actor-Advisor, an extension of Policy Gradient that allows an advisory policy π_A to directly influence the actions selected by the agent, in a learning from human intervention task. We illustrate the Actor-Advisor in a human-agent cooperation setting, where the advisory policy is defined by a human. We show that the Actor-Advisor allows good policies to be learned from scarce advice, is robust to errors in the advice, and leads to higher returns than no advice, or reward-based approaches. Contrary to other approaches, reviewed in Sections 4.1.1 to 4.1.3, the Actor-Advisor uses the environmental reward, and still allows the optimal policy to be learned. Finally, although we used the example of a human advisory policy, and compared our work to other human-based approaches, it is important to note that any advisory policy can be used to shape a learner's policy, such as expert demonstrations, policies to be distilled from other agents, backup policies for Safe Reinforcement Learning, or a mix of all the above. the Actor-Advisor is therefore a straightforward, effective, and widely applicable approach to policy shaping.

On a higher level, the Actor-Advisor is an original approach to learning from human intervention. It uses immediate advice instead of feedback or demonstrations, and incorporates that advice in the learning process by directly shaping a policy, instead of producing rewards or a value function. The Actor-Advisor is also easier to implement. For instance, a theoretically correct implementation of reward-shaping requires that the reward function is potential-based, which requires advanced algorithmic precau-

tions to be implemented [Harutyunyan et al., 2015b]. Such precautions are not needed by the Actor-Advisor. Moreover, incorporating human input through the reward function leads to slowly changing the agent's policy, in contrast to directly influencing the agent's policy. In this regard, our work is pushing forward two promising (but not yet well-investigated) paradigms, namely advice as a tool for humans to teach agents, and policy shaping as a method to learn from external directives.

Test	Episode Number	Human reward	Human advice
t-test	[180, 220]	0.343	0.001
	[960, 1000]	0.035	1.464e-06
Wilcoxon	[180, 220]	0.419	1.369e-03
	[960, 1000]	0.007	2.396e-09

Table 4.1: P-values comparing the returns obtained by Policy Gradient (PG) to PG + Human Reward and PG + Human Advice, in the early and late stages of learning. The returns obtained by PG + Human Advice are significantly higher than the ones obtained by PG + Human Reward, both in early and late stages of learning. Each test is performed on 41 (episodes) $\times$ 16 (runs) samples of cumulative reward values, taken in one of two given episode intervals.

L	p-value (Wilcoxon)
0.05	0.047
0.2	1.294e-07
1	2.035e-08

Table 4.2: P-values comparing the returns obtained by Policy Gradient (PG) to PG + Human Advice when advice is interrupted after 700 human interventions, on 41 (episodes) $\times$ 16 (runs) samples in the late stage of learning (episode interval [960, 1000]). The performance of PG + Human Advice is significantly higher than of Policy Gradient for the highest values of L. Interestingly, there is little significance difference between $L = 0.2$ and $L = 1$.

5 | Transfer Learning

This chapter is drawn from our publication, Plisnier, Steckelmacher, Roijers and Nowé, *Transfer Reinforcement Learning Across Environment Dynamics With Multiple Advisors*, Benelux Conference on Artificial Intelligence (BNAIC), 2019.

In the previous chapter, we looked at how advice from a simulated teacher can help improve the sample-efficiency of a Policy Gradient agent, via the use of the Actor-Advisor. As detailed in Chapter 3, the Actor-Advisor is based on Policy Intersection [Griffith et al., 2013], an algorithm originally designed to integrate human input in the learning process of an RL agent. Griffith et al. [2013], as well as our own contribution in Chapter 4, show that human input is valuable to accelerate learning by guiding the agent's exploration. Nevertheless, despite being primarily thought out as a learning from human intervention algorithm, Policy Intersection can be more generally considered as a transfer learning approach [García and Fernández, 2015], guiding an RL agent's exploration towards fruitful areas according to readily available knowledge, which can come from any source.

Arguably, humans constantly perform the same routine tasks over and over again, and even the new situations people may face are likely to resemble known ones. For instance, once you know how to open windows at home, opening the windows at your office or at your friend's house should not be a problem for you, despite minor design disparities. Solving new situations, yet similar to previous ones, are seldom a problem, since we (hopefully) do not have to relearn from scratch how to open a window every time we meet a new window. Vanilla Reinforcement Learning agents, on the other hand, do not automatically identify the links between old and new windows to be opened, and tend to tackle each new window with a blank memory, as if it were a

whole new adventure. Unfortunately, as the reader can easily guess, this does not lead to an efficient nor sustainable manner for RL agents to undertake real-world challenges in the long term. This is where Transfer Learning comes in.

Transfer Learning is an intuitive way to help an RL agent learn to perform tasks in a more sample-efficient manner. A Transfer Learning setting often involves a *source* task and a *target* task; first, the agent learns the source task, then the knowledge acquired in the source task is intelligently leveraged by the agent while tackling the target task, using a Transfer Learning algorithm [Taylor and Stone, 2009; Zhu et al., 2020]. Transferred knowledge can be the agent's learned policy or Q-values [Taylor et al., 2007; Fernández and Veloso, 2006; Brys, 2016]; learned skills or options [Andre and Russell, 2002; Ravindran and Barto, 2003; Konidaris and Barto, 2007], some general enough skill (like walking) to fit a large set of target tasks [Tang and Haarnoja, 2017]; parts of a modular neural network policy in which each module deals with a different aspect of the task [Devin et al., 2017; Mirowski et al., 2018]. Methods to effectively transfer knowledge include reward shaping [Brys et al., 2015], policy reuse methods, which shape the agent's exploration strategy [Fernández and Veloso, 2006; Griffith et al., 2013], initializing a policy [Taylor et al., 2007] using information from the source task, and initializing parts of the target network with the source network [Devin et al., 2017; Mirowski et al., 2018; Chaplot et al., 2016], to only cite a few. When it comes to the specific algorithms we evaluate in this chapter, i.e., Probabilistic Policy Reuse and Policy Intersection (which we review in Section 2.5), they can both be labelled as Transfer Learning via guiding the agent's exploration [García and Fernández, 2015]. PPR [Fernández and Veloso, 2006] has specifically been though out for Transfer Learning purposes; Policy Intersection [Griffith et al., 2013], on the other hand, was first introduced as a way to incorporate human interventions in the RL agent's learning process to improve its performance.

In this chapter, we present four applications of Transfer Learning leveraging the Actor-Advisor framework, each tackling a specific current limitation of existing Transfer Learning methods:

Transfer across robotic platforms with different sensors
: Transferring policies from a robotic platform equipped with a set of potentially expensive sensors, to another platform with similar effectors, but equipped with different, less efficient sensors. Once trained using this setting, the resulting system behaves as well as one equipped with a full set of expensive sensors, while running on a much less sophisticated platform. The appeal of this setting is the possibility to drastically reduce the cost of the robot, making the system more affordable to members of the general public.

Transfer from multiple advisors
: Transferring an "average policy" of multiple agents trained on a set of similar yet

distinct tasks, to help a fresh agent tackle a new task. This average policy acts as a voting mechanism for previously learned policies, potentially determining which parts are shared across tasks, and which parts differ. This average policy allows the fresh agent to make assumptions about which actions advised by the transferred policies can be applied to the new task, and which actions can be deemed as irrelevant, *prior to having any interaction with the new task*.

Transfer to kickstart learning in an air compressor management problem
Transferring from previously developed controllers on an environment with varying parameters, to learn a new controller faster on an additional version of the environment. It is not uncommon that, for multiple distinct variants of a given setting, multiple corresponding controllers must be developed to tackle them. However, learning each controller from scratch can be unfeasible, and unsustainable if the number of different variants is large.

Reinforcement learning web-service with transfer across users
Transferring policies across users of a smartphone application (for instance) offers each user the possibility to interact with an already well-formed agent, only requiring refinements specific to the user's preferences. Such transfer allows to model the behaviors most users tend to like, but keeps agents flexible enough for user individual personalisation.

Before presenting each of our contributions one by one, we non-exhaustively review and categorize some well-investigated Transfer Learning methods.

5.1 Existing Transfer Learning Techniques

Transferring knowledge in Reinforcement Learning potentially improves sample-efficiency, as it allows an agent to exploit relevant past knowledge while tackling a new task, instead of learning the new task from scratch [Taylor and Stone, 2009]. Usually, we consider that the valuable knowledge to be transferred in Reinforcement Learning is the actual output of a reinforcement learner: a Q function or a policy π [Brys, 2016, p.34]. Some work also consider reusing skills, or *options* [Sutton et al., 1999a], as a transfer of knowledge across tasks [Andre and Russell, 2002; Ravindran and Barto, 2003; Konidaris and Barto, 2007]. We focus on Transfer Learning methods directly reusing π_{source}, the policy learned in the source task. In this section, we sort previous work in categories related to the *way* π_{source} is transferred into the agent, and look at what is allowed to be different between the source task and the target task. The two predominant ways in which knowledge can be transferred are i) π_{source} serves as a guide during exploration, ii) π_{source} is used to train or initialize the agent, so that the agent actively imitates π_{source}.

5.1.1 Exploration

Guiding the exploration strategy of the agent, the topic of this thesis, transfers a policy in a fast and effective way. Altering the exploration strategy is a popular technique in the safe RL domain, and consists in biasing or determining the actions taken by the agent at action selection time [García and Fernández, 2015]. Such exploration requires the agent to be able to learn from *off-policy* experiences. The motivation behind guided exploration is the poor performance of a fresh agent at the beginning of learning, in addition to the presence of obstacles difficult to overcome in the environment. An exploration guided by a smarter external policy, such as π_{source}, could help improve the agent's early performance, as well as help it in the long run explore fruitful areas.

Some existing work applies guided exploration to transfer learning [Fernández and Veloso, 2006; Taylor and Stone, 2007; Madden and Howley, 2004], and illustrates how this technique allows the agent to outperform π_{source}'s performance. Regarding the components of the source and target tasks that are allowed to differ, Fernández and Veloso [2006] considers different goal placements (hence, different reward functions), Madden and Howley [2004] uses symbolically learned knowledge to tackle states that are seen by the agent for the first time, and Taylor and Stone [2007] assumes similar state variables and actions, but a different reward function. The translation functions required to map a state/action in one task to a state/action in another are assumed to be provided.

5.1.2 Learning

Although an improved exploration might result in an improved performance, and a jump-start occurs, an agent which actions are simply overridden by an external policy does not actively learn to imitate it. Other techniques have proposed to "teach" the agent to perform the target task (instead of merely guiding it), either by dynamic teaching, or by straightforward initialization. Imitation learning aims to allow a student agent to learn the policy of a demonstrator, out of data that it has generated [Hussein et al., 2017]. Similarly, policy distillation [Bucila et al., 2006] can be applied to RL to train a fresh agent with one or several expert policies, hence resulting in one, smaller, potentially multi-task RL agent [Rusu et al., 2015].

Imitation learning and policy distillation are somewhat related to transfer learning [Hussein et al., 2017, p.24], although imitation and distillation assume that the source and target tasks are the same, while transfer does not. The Actor-Mimic [Parisotto et al., 2015] uses several DQN policies (each expert in a different source task) to train a multi-task student network, by minimizing the cross-entropy loss between the student and experts' policies. To perform transfer, the resulting multi-task expert policy is used to initialize yet another DQN network, which learns the target task. The Actor-Mimic assumes that the source and target tasks share the same observation and action

spaces, with different reward and transition functions. In *Q-value reuse* [Taylor et al., 2007, Section 5.5], a Q-Learner uses Q_{source} to kickstarts its learning of the target task, while also learning a new action-value function Q_{target} to compensate Q_{source}'s irrelevant knowledge. In Taylor et al. [2007], the agent learns to play Keepaway games, and is introduced to a game with more players, resulting in more actions and state variables. Brys et al. [2015] transfers π_{source} to a Q-Learning agent through reward shaping; the differences in the action and state spaces between source and target tasks are solved using a provided translation function.

Our Actor-Advisor tries to get the best of both the exploration alteration and the teaching worlds. In a Transfer Learning context, it mixes π_{source} with the policy of the actor at action selection time (hence biasing the exploration strategy), using the policy mixing formula in Griffith et al. [2013]; the mixed policy is also integrated in the loss or update rule of the actor. This way, the learning process is influenced by π_{source}, while it also guides the agent's exploration.

5.2 Transfer Across Robotic Platforms with Different Sensors

For a robot to learn a good policy, it often requires expensive equipment (such as sophisticated sensors) and a prepared training environment conducive to learning. However, it is seldom possible to perfectly equip robots for economic reasons, nor to guarantee ideal learning conditions, when deployed in real-life environments. A solution would be to prepare the robot in a laboratory environment, in which all necessary material is available to learn a good policy. After training in the lab, the robot should be able to get by *without* the expensive equipment that used to be available to it, and yet still be guaranteed to perform well on the field. The transition between the lab (*source*) and the real-world environment (*target*) is related to transfer learning, where the state-space between the source and target tasks differ.

In Transfer Learning, many different components can vary between the source and target tasks. The state description, for instance, might not be the same in the two tasks; one might be richer and/or easier to learn from, than the other [Taylor and Stone, 2009, Section 3.2.1]. This dissimilarity naturally emerges when trying to share knowledge across robots equipped with different sensors. In this section, we are interested in transferring knowledge from a platform to another, both tackling the same task with the same action set, while sensing their environment differently. The transfer is made *from* the robot which state description is empirically easier to learn from *to* the robot which state description is harder to learn from. This could allow a cheap under-equipped robot to perform as well on the field as a more sophisticated one, for which the task is much easier to learn.

The task to be learned is a simulated drone navigation task in a suburban-like street, with houses and trees along it. We leverage two types of observations, which are typically collected by different types of sensors: depth maps, and color images. As suggested above, we would want to give our advisor observations of the best type, the observation type that helps an RL agent to learn the task the most. Indeed, in our problem, we assume that the most helpful type of observation is likely to be produced by the most expensive and effective sensors, which are only available in the lab environment. The resulting advisor policy would then be used by a second agent learning the same task in the same environment, but with observations of the type of lesser quality as input. However, as we are not sure whether observing depth maps or color images is the best to solve this particular task, we train two advisors, one on depth maps, the other on color images, and let the depth maps advisor advise agents seeing color images, and vice versa.

All agents in this setting, be them advisors or advisees, learn using Proximal Policy Optimization (PPO) [Schulman et al., 2017], reviewed in Chapter 2, Section 2.3. We compare our Actor-Advisor framework to reward shaping based on the advisor's policy, a method to perform transfer learning proposed by Brys et al. [2015]. We empirically observe that the Actor-Advisor reaches good rewards much sooner, shows a significant jumpstart compared to its reward-shaping-based challenger, and maintains a good policy when executed without the help of the advisor after training.

5.2.1 Reward Shaping for Policy Transfer

The original reward shaping for policy transfer, as introduced by Brys [2016], defines the shaping reward as the probability $R^{\pi_{advisor}} = \pi_{advisor}(s, a_{advisee})$ that the advisor would have taken action $a_{advisee}$ chosen by the advisee in state s. However, Brys [2016] argue that the advisee should not simply be fed with that policy-based shaping reward, since it is not based on a potential function as is. The risk behind a non-potential based reward shaping function is to prevent the advisee from learning the task it was supposed to learn in the first place [Ng et al., 1999]. Building on results by Harutyunyan et al. [2015b], Brys [2016] learns the Q-function $Q^{-R^{\pi_{advisor}}}$ related to the negation of $R^{\pi_{advisor}}$. $Q^{-R^{\pi_{advisor}}}$ is then used to base the shaping reward on, as Q-function are always potential functions.

In our setting, we do not go through the trouble of computing $Q^{-R^{\pi_{advisor}}}$, and directly feed the advisee with $R^{\pi_{advisor}} = \pi_{advisor}(s, a_{advisee})$. We justify our decision as follows: i) in our case, both the advisor and the advisee aim at solving the same task, hence this shaping should not perturb what the advisee eventually learns; ii) computing $Q^{-R^{\pi_{advisor}}}$ would have taken extra time that then would have to be accounted for at the disadvantage of the reward shaping method versus the Actor-Advisor, which does not require such lengthy preprocessing. Furthermore, our results

below empirically show that directly feeding the advisee with its advisor's policy as an additional reward not only preserves learning, but also yields a non-negligeable policy improvement towards the end of learning.

5.2.2 A Drone Flying Down the Street

In our setting, we use AirSim [1], a drone simulator, to let a drone learn to fly down a street in a suburban area, while avoiding to get stuck in trees or on roofs.

Environment

The specific environment in the AirSim world in which our experiment takes place is called Neighborhood. It simulates a drone flying around in a suburban area, and risking getting caught in trees and bumping against the roof of houses. In case of a collision, AirSim offers the possibility to retrieve information related to the object with which the drone collided. The goal of the agent is to fly forward while avoiding obstacle collision. An episode ends either in case of a collision, resulting in a -50 reward, or after 200 time-steps. The agent has four actions at its disposal: go forward, turn left, turn right, or go down. We disable the "go up" action to prevent the agent from simply learning to first fly high above trees and other obstacles. This way, the agent would be able to resume its task by flying around randomly, without ever hitting obstacles thanks to its high position in the sky. Instead, we want to force the drone to learn to fly carefully around objects. The agent receives a -2 reward at each time-step, unless it goes forward, in which case it receives a zero reward.

As mentioned above, there are two potential ways for the drone to perceive its surroundings in Neighborhood: via depth maps, or color images. Depending on the observation type used, states can either be matrices of $144 \times 255 \times 1$ (in the case of depth maps) or $144 \times 255 \times 3$ (in the case of color images). As the formal description of an MDP includes the state space, learning to navigate while observing depth maps consists in a different task than learning to navigate while observing color images. As a result, although the goal remains the same in both tasks, the transfer of policy from one to another accurately falls in the realm of conventional Transfer Learning.

Managing Multiple Sensors

To perform the actual transfer of a policy learned using one type of observation to another policy learned using another type of observation, we evaluate and compare two distinct methods: a reward-shaping based method feeding probabilities to the advisee

[1] AirSim is a simulator notably for drones and cars, built on the Unreal Engine, see: https://microsoft.github.io/AirSim/

as an additional reward [Brys, 2016], and our Actor-Advisor method. Both Transfer Learning methods rely on the training of a first agent, the advisor, which does not receive any external advice while learning. Once this advisor is obtained, the advisee learns while receiving input from the advisor, either in the form of additional reward in the case of reward shaping, or probability distributions over actions in the case of the Actor-Advisor. However, due to the fact that the advisor and advisee do not use the same sensor to observe the environment, we are faced with a technical challenge to allow the advisor to help the advisee during training.

Be it for reward shaping or the Actor-Advisor, to be able to advise the advisee, the advisor must be fed with observations of its own observation type (depth maps or color images), even though it is not updating its own policy anymore. This leads to the following assumption: during the training of the advisee, observations of both types (depth maps and color images) must be made available to the agent by the environment. We do not consider this assumption to be a problem, since that, in our setting, the advisee's training takes place in the laboratory environment, similarly to the advisor's training.

To solve this issue, we manage our advisor (i.e., feeding it its input, retrieving its output) in a environment wrapper, externally to the RL algorithm. This environment wrapper receives the state from the original environment, in which observations of both types are present, and dispatches observations in two different states, one for the actor and one for the advisor. When the reward shaping method is enabled, the environment wrapper adds an extra reward coming from the advisor (following the method proposed by Brys [2016]) to the original environmental reward. In case the Actor-Advisor is used, then the advice produced by the advisor, which is in the form of a vector of probabilities over actions in the case of the Actor-Advisor, is put in the advisee's state. When executing the advisee's policy after training without the help of the advisor, the advice value in the advisee's state is replaced by a vector of ones, rendering the advisor's influence null. Making the advice part of the agent's observation is made possible by the MultiInputActorCriticPolicy class in Stable Baselines 3, and a modification done to that class by Denis Steckelmacher, one of our colleagues.

Results

The goal of our drone is to navigate without bumping into obstacles; managing to move forward without any collision results in an episodic reward of zero. In our motivating story, one sensor is more effective than the other at making the agent learn the task, because it is more rich and informative, or particularly well suited for a neural network to learn from. However, in practice, a PPO agent shows no problem learning to navigate the drone, whether the agent is seeing only depth maps, or only

camera images. In the specific case we are facing, it just so happens that not one observation type suits the agent better than the other.

Since we did not know whether there would be a difference between using an observation type over another, and if so which observation type leads to the best advisor before launching our experiments, we tried both ways (see Figure 5.1):

i) rewards from depth maps (DM)/advised by DM: agent trained while seeing camera images, and receiving help (either through reward shaping or the Actor-Advisor) from an advisor seeing depth maps;

ii) rewards from camera images (CI)/advised by CI: agent trained while seeing depth maps, and receiving help from an advisor seeing camera images;

In both cases, the reward shaping method does not surpasses the baseline, until around 90 episodes, where it almost reaches a zero reward. The Actor-Advisor, on the other hand, manages to produce a jumpstart at the beginning, and keeps performance significantly above the baseline throughout the learning process. These results can be looked at more closely in Figure 5.2.

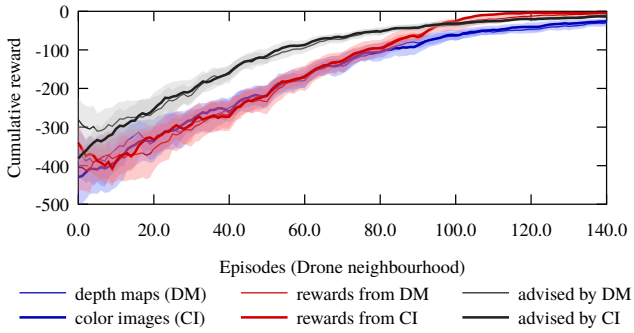

Figure 5.1: PPO seems to learn the task well, whether it is learning from depth maps (DM) or camera images (CI). Agents learning from depth maps while having their reward shaped by an advisor trained on camera images, and vice versa, does not lead to a significant improvement in performance, except at the very end of learning, where both reward shaping settings exceed the baseline. In contrast, when the transfer between advisor and advisee leverages the Actor-Advisor, we observe a non-negligeable jumpstart at the beginning of learning, followed by a steady positive difference, outperforming the baseline as well as the reward shaping method. The legends "rewards from DM", and "advised by DM" must be understood as follows: the agent learned with camera images as observations, and an advisor trained on depth maps (either using reward shaping or the Actor-Advisor respectively). Legends such as "rewards from CI", and "advised by CI" mean the other way around.

At the beginning of learning, the Actor-Advisor manages to gain a 20 episodes advance over the baseline and the reward shaping method. This difference is (somewhat) maintained throughout learning, until around 120 episodes in, when the performance of the reward shaping method surpasses that of the Actor-Advisor. To make sure that our advisees can get by without their advisor after 140 episodes of training, we freeze our advisees' policies and run them on the same environment, without updating their policies anymore (see Figure 5.3). They manage to maintain a reward close to zero throughout execution, without any external help from an advisor.

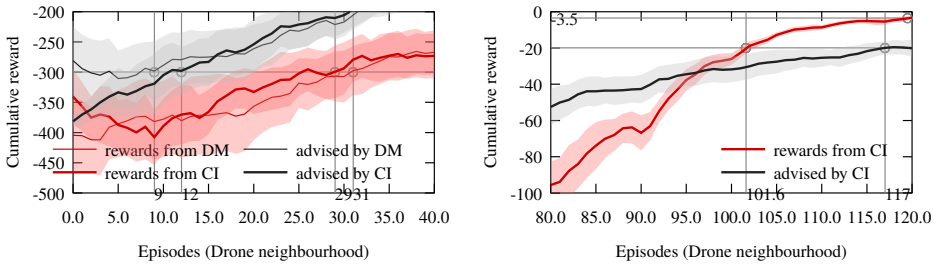

Figure 5.2: *Left:* Zoom in between 0 and 40 episodes in the learning process: the Actor-Advisor leads to a non-negligeable jumpstart at the beginning of learning; the Actor-Advisor method needs at most 12 episodes to reach a -300 reward, while the reward shaping method requires at least 29 episodes. *Right:* Zoom in between 80 and 120 episodes in the learning process: the reward shaping method eventually surpasses both the baseline and the Actor-Advisor towards the very end of learning; the reward shaping method reaches a -3.5 reward after a 120 episodes, while the Actor-Advisor plateaus at -20 until the end of our experiment.

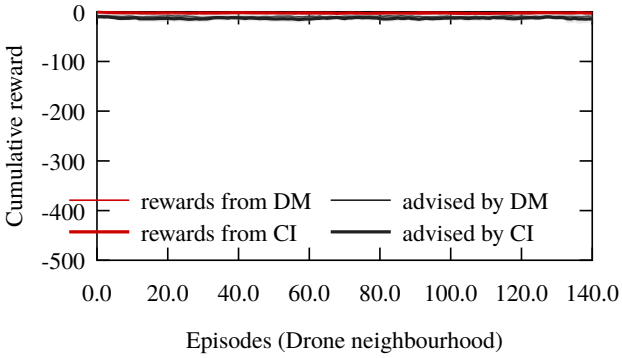

Figure 5.3: We execute the policies of advisees, either trained using the reward shaping method or the Actor-Advisor. During execution, the advisee's policy is not updated (it does not learn anymore), and the advisor is not queried; the advisee chooses actions without the advisor's help. All agents reach a close to zero reward; the absence of advisor does not impact performance during execution, as required.

Conclusion

To make assistive robot technology more accessible to a wide audience, one might want to cut back on the cost of the equipment required by the robotic platform, such as expensive sensors. However, an expensive sensor is often an effective, powerful one, that can help a Reinforcement Learning agent learn a policy much faster than without it. That is where Transfer Learning becomes handy: an agent is trained in a laboratory environment, where all necessary equipment is available, then launched on the field with much cheaper sensors. This way, the agent can learn to *get by without* the particular equipment that will no longer be present once deployed in the real-world. Nevertheless, to see if that plan is feasible, we must first ensure that transfer between tasks with extremely different state spaces, due to the change in sensors, can lead to positive transfer. In this section, we let an agent controlling a drone seeing depth maps train while being advised by another agent seeing color images, and vice versa. Interestingly, in this case, not one observation type seems better at forming an advisor than the other. The Actor-Advisor manages to extract valuable advice to be fed to the advisee in both situations, and yields an non-negligeable increase in performance during learning. To conclude, transfer across tasks with wildly different state spaces

can improve sample-efficiency and lead to an agent getting by without its advisor's help, as well as without its advisor's (potentially expensive) sensors.

5.3 Transfer from Multiple Advisors

An important challenge of Transfer Learning, for an RL agent launched in a target task, is to determine which parts are shared by both the target task and the source task, and which parts differ. This information is critical; indeed one cannot just blindly follow the policy learned in the source task while tackling the target task, because a behavior that worked well in the past might not apply anymore. Therefore, it is necessary for a Transfer Learning algorithm to be capable of following the past policy only when suitable, and to otherwise diverge from it. To mitigate this problem, current techniques either artificially set a probability of following the advisor, hence limiting the potential damage of irrelevant advice [Fernández and Veloso, 2006]; or let the agent learn when to follow advice [Taylor and Stone, 2007; Taylor et al., 2007; Parisotto et al., 2015]. If several expert policies are available, then isolating one of them and only exploiting that one policy often becomes the focus, instead of exploiting a combination of them [Taylor and Stone, 2007; Fernández and Veloso, 2006]. Hence, existing methods tend to add an extra learning problem, by either learning when to follow the advisor, or which advisor to follow.

In this section, we present a novel approach to Transfer Learning, building on the Actor-Advisor, in which all available source policies are exploited to help learn related new tasks. Our in-depth empirical evaluation demonstrates that our approach significantly improves sample-efficiency. We consider a setting where the agent needs to learn a large amount of similar yet distinct tasks. An example of this setting is a navigation task, in which the agent has to learn how to go to any of the 22 offices on a floor. Offices may have different shapes and furnishing, which makes them share a state representation, but *distinct transition dynamics*. We propose a method to reuse the knowledge acquired from learning several past tasks to tackle a new task, having the same state-space and action space but different dynamics and reward functions. We have at our disposal a number of expert source policies learned in the same environment, which we call *advisors*. Our method, integrated with the Actor-Advisor, combines all of the available expert source policies into one advisor. This advisor is confident in areas where the source tasks are similar, meaning that the advice probably applies in the target task as well, and is uncertain where the source tasks differ. The latter case allows the fresh agent to learn for itself what is best to do in situations unknown to its advisor. We implement our Transfer Learning method within Bootstrapped Dual Policy Iteration (BDPI) [Steckelmacher et al., 2019], as presented in Chapter 3, Section 3.1.3, an already extremely sample-efficient model-

free actor-critic Reinforcement Learning algorithm, and we empirically show that, combined with our contribution, BDPI can achieve an even higher sample-efficiency.

5.3.1 BDPI with Multiple Advisors

In this section, we leverage our novel extension of BDPI fully detailed in Section 3.1.3, in Chapter 3, that allows advice to be given to the agent. On top of that extension, we introduce a training method that consists of training an agent on N tasks, then using these N actors to advise the $N+1^{th}$ agent, then using the $N+1$ agents to advise the $N+2^{th}$, and so on.

An important challenge in Transfer Reinforcement Learning is discovering when to follow the advisor (trained on the source task), and when to ignore it. Ideally, the advisor should be followed only in the states for which its policy is optimal in the target task. Unfortunately, knowing whether the advisory policy is optimal in a state is impossible until the agent has fully learned the target task.

Previous work learns in which states to use the advisor, or which advisor to use in which state [Fernández and Veloso, 2006; Taylor and Stone, 2007], and makes sure to only sample one advisor at a time. We argue that it is initially impossible to evaluate when to follow advice, and when to ignore it, based solely on the advice received from a single advisor. Building on our hypothesis that evaluating a single advisor in a sample-efficient way is impossible, we instead propose to use *several* advisors, and combine them in the following way:

$$\pi_A(s) = \frac{1}{N} \sum_i \left(\pi_A^i(s) + 1 - \rho \right) \tag{5.1}$$

where π_A denotes the advisory policy (as always), N the number of advisors, and $\rho \leq 1$ the weight of the advisors' influence, that does nothing when set to 1, and artificially increases the entropy of the advisors when set to a value smaller than 1. The resulting vector $\pi_A(s)$ is then normalized to sum to 1, by dividing each element by the sum of all elements. Equation 5.1 is a simple average of the advice given by all the advisors, and can be seen as a uncertainty quantification through the use of an ensemble approach [Chua et al., 2018; Lakshminarayanan et al., 2017; Kurutach et al., 2018]. Intuitively, if N is large enough (typically 5 to 10), the advisors will tend to agree in states in which the tasks being learned share a common structure, and disagree in states where the tasks differ. Even if every single advisor is highly confident in these differing states, the average will produce a probability distribution of high entropy. Moreover, the advisors may agree that a few actions are bad in a given state, while not agreeing on which ones are good. To leave a room, for instance, moving towards the window

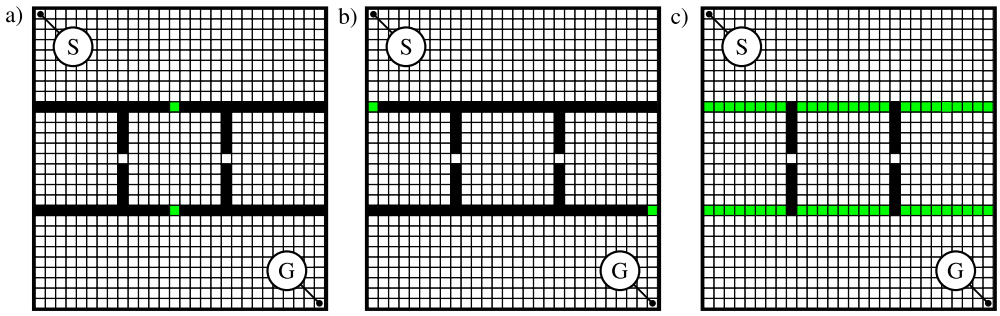

Figure 5.4: The 29×27 Five Rooms environment used in our experiments. 'S' denotes the starting cell, 'G' the goal cell. a) The original door configuration. The first and last doors (in green) are the ones that are allowed to move; b) an example of an alternative configuration of the doors; c) all the potential door locations. For each of the two doors, there are 26 such locations, which results in $26 \times 26 = 676$ different environment configurations.

is always bad, regardless of the room, while moving towards the door will be more strongly advised.

Building on Equation 5.1, we propose the following training method for multiple tasks that share a common structure but different dynamics: train N actors on N tasks from scratch, with N between 5 and 10 depending on the complexity of the tasks. Then, combine the N actors in a single advisor, using Equation 5.1, and use it to significantly improve the sample-efficiency and safety of the $N + 1^{th}$ actor. Then, add the $N + 1^{th}$ actor to the pool of actors used to produce advice, and repeat for $N + 2$, ... Our experimental results, that we now present, validate our approach and show that, as more advisors are available, the sample-efficiency of the agent increases on new tasks.

5.3.2 Generalizing Across Multiple Navigation Tasks

We now evaluate our Advised BDPI algorithm in a representative environment for which 676 tasks have to be learned. We demonstrate that learning a small amount of tasks from scratch (even as low as 4) allows the next tasks to be solved significantly faster. Moreover, the more tasks have been learned, the more sample-efficient the agent becomes on new tasks. Combined with the already high sample-efficiency of BDPI, this demonstrates that reinforcement-learning can now be used to train full multi-task systems.

Environment

We evaluate our method on our custom environment, Five Rooms (see Figure 5.4, detailed in Chapter 4, Section 4.3.2. In order to reach the goal, the agent imperatively has to go through at least two doors (the ones in green on Figure 5.4, a.). Reaching the goal results in a $+100$ reward, the agent otherwise receives -1 per time-step. The episode terminates either once the goal is reached, or after 500 unfruitful time-steps.

In our experiments, we vary the location of the doors on the two horizontal walls (see Figure 5.4). Each door can be located in any of 26 cells. Every combination allows the goal to be reached from the initial position, but some combinations lead to shorter or longer optimal paths (see, in Figure 5.5, the difference between the paths in examples a and g). Because we do not move the goal, and do not alter the reward function, but only vary the location of the doors across task, our environment isolates the impact of a varying transition function. Moving doors inside the environment is also a simple but relevant illustration of our example task: *learn to get out of many rooms, each room differing by its furniture.* Finally, moving a door is a localized change to the environment, but is still very challenging: directly applying a policy learned in a source configuration in a target configuration would lead the agent to get stuck against a wall in the target environment (where there is a door in the source environment). The source policy, strongly used to see a door where it is not there anymore, would be highly confident in its bad action and mislead the fresh agent. We now describe our experimental setup, and show that our Advised BDPI successfully tackles this challenge.

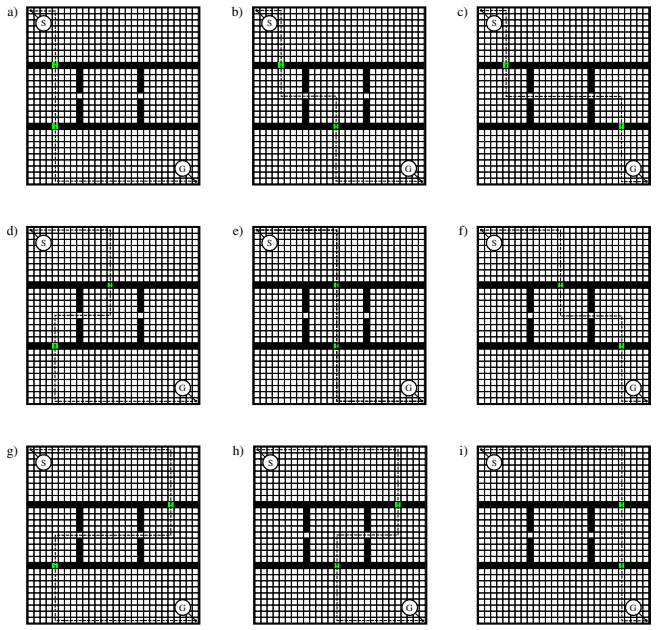

Figure 5.5: These 9 examples of configurations show how the difficulty of the task (i.e., reaching the goal from the initial cell) can strongly vary from configuration to configuration. The path from the initial cell to the goal cell is much longer and convoluted in configuration g) than in configuration a), for instance.

Experimental Settings

We evaluate our Advised BDPI on the tasks described above. BDPI has been configured closely to what is recommended in Steckelmacher et al. [2019]: 8 critics, all trained every time-step for 4 training epochs. The current state (the current cell in which the agent is) is one-hot encoded into a vector of 812 floats. The BDPI actor and critics are neural networks with a single hidden layer of 256 neurons, with the tanh activation function, and one output per action. Actor and critic networks are trained with the Adam optimizer, with a learning rate of 0.0001, for 10 gradient steps per training epoch. In order to produce our results, we trained many agent in this order:

1. 100 agents have been trained from scratch, each on a different random door configuration (from 676). This produces a pool of advisors.

2. The curves of Figure 5.6 have been produced by training 20 agents (per curve), each on a randomly-selected configuration of doors, and using 50 of the advisors produced at step 1 for advice.

3. The curves of Figure 5.9 have been produced the same way as in step 2, but by using either more (100) or less (4 or 1) advisors for advice. Our experiments with $N = 100, 50, 4$ or 1 advisors measure the sample-efficiency gains that are obtained when training the $N + 1^{\text{th}}$ agent using N advisors.

4. The curves of Figure 5.12 have been produced the same way as in step 2 and 3, but means to illustrate the effect of implementing the learning correction (see Section 5.3.1). They are both advised by 100 advisors, and averaged over 10 runs each.

5. For steps 2, 3 and 4, we vary the ρ parameter of Equation 5.1, to evaluate the impact of artificially increasing the entropy of the advisors. We generally show that artificially increasing the entropy of the advisors is not needed to obtain good results, which demonstrates the benefits of averaging advisors, and removes one tunable parameter from our algorithm.

Because each configuration of doors has an optimal policy that achieves a different return (as illustrated in Figure 5.5), we evaluate each agent on a large amount of runs, to average out the effect of the door positions. This allows us to produce curves with high confidence.

Results

Figure 5.6 shows that using advice generally improves performance, especially at the beginning of learning. In addition, it can be noticed that a high ρ (i.e., $\rho = 1$) helps in the beginning of learning, but tends to prevent the agent from achieving the best performance towards the end. A lower ρ (i.e., $\rho = 0.7$), on the other hand, provides more freedom to the agent to reach that high performance at the end, but at the cost of lower performance in early learning stages.

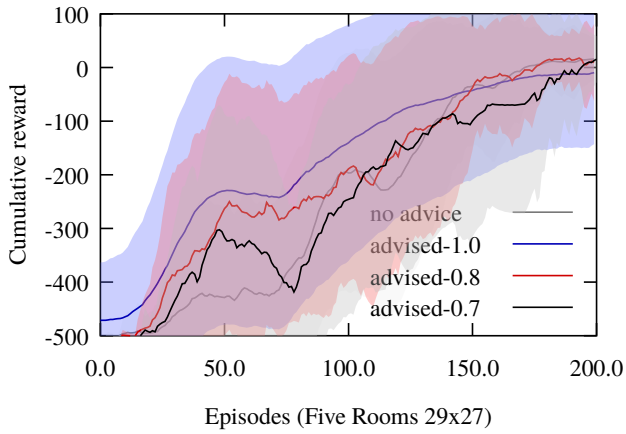

Figure 5.6: Comparison between learning the task while using advice from 50 advisors (with $\rho = 1.0$, 0.8 or 0.7) and learning the task without advice. These curves are averaged over multiple runs: 25 runs for the "advised" curves, and 100 runs for the "no advice" curve. A high ρ greatly helps the agent at the beginning of learning but slightly decreases performance in the long run, while a lower ρ allows the agent to reach a better policy at the end of learning, but provides a weaker jumpstart.

We also evaluate the impact of either having a large amount of advisors (100, in our case) or only a few (4 or 1), while varying the advisors' weight ρ (0.8 or 1, see Figure 5.9). When $\rho = 0.8$ (see Figure 5.7), i.e., the influence of the advisors is somewhat moderate, having a few or a large amount does not seem to matter in the long run. However, when the influence of the advisors is maximum (i.e., $\rho = 1.0$, in Figure 5.8), then increasing the amount of advisors increases sample-efficiency. Averaging over a large number of advisors naturally exhibits uncertainty in areas where the target task and the source tasks might differ, allowing the agent to be less dependent on the ρ parameter. Even having only 4 advisors instead of 1 dramatically improves performance.

We compared leveraging several advisors to leveraging only one carefully selected advisor. This advisor is selected based on its doors configuration; the location of the doors in its source task is the closest to the location of the doors in the target task. Hence, as suggested by Fernández and Veloso [2006], out of all advisors from the pool, this advisor should be the most suited to provide quality advice to the fresh agent. However, even though using the best possible advisor leads to impressive results when $\rho = 0.8$, averaging over 4 randomly chosen advisors can still provide better results in

the long run, in both cases where $\rho = 0.8$ and $\rho = 1.0$. Moreover, having only one advisor, albeit the best, still makes the fresh agent rely on a low value of ρ.

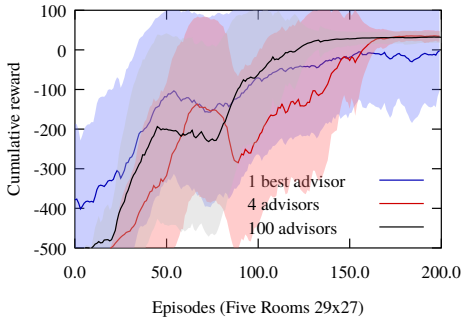

Figure 5.7: $\rho = 0.8$

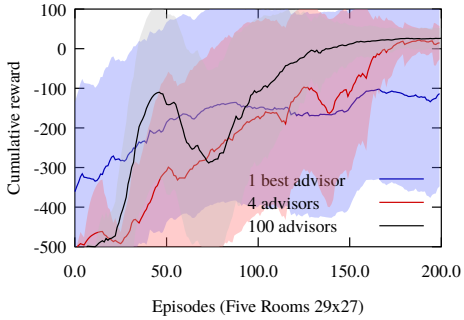

Figure 5.8: $\rho = 1.0$

Figure 5.9: Comparison between learning the task while advised by either 100 advisors, 4 advisors or the best advisor for the target task. The "best advisor" is the advisor which source task has the most similar doors configuration to that of the target task. Even though one carefully chosen advisor gives better advice than a randomly chosen one, the performance it can achieve in the long run is still below that of 4 randomly chosen advisors, whether $\rho = 0.8$ or $\rho = 1.0$. Additionally, being advised by only advisor, albeit the most suited one for the target task, still leads to an agent highly dependent on ρ.

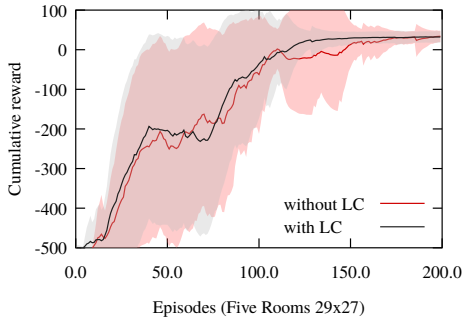

Figure 5.10: $\rho = 0.8$

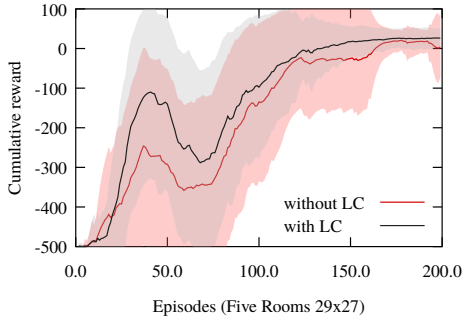

Figure 5.11: $\rho = 1.0$

Figure 5.12: In both plots, the two curves are averages over 20 runs of agents advised by 100 advisors. We compare between using advice with the learning correction and without the learning correction. The learning correction ensures a more stable performance regardless of whether $\rho = 0.8$ or 1.0 than when it is not implemented.

Finally, we assess the influence of the learning correction (see Section 5.3.1) on learning while being advised (see Figure 5.12). When we do not artificially increase the entropy of the advisors (i.e., when $\rho = 1$), learning the task while being advised is harder without the learning correction. The importance of the learning correction is lesser when $\rho < 1$, though. In contrast, the performance of the agents with the learning correction is similar in both settings of ρ (0.8 or 1), which demonstrates that our learning correction positively impacts the robustness of the agent.

We have a hunch as to why there is a visible dip in the performance in almost all plots, in between 50 and 100 episodes. At the beginning of learning, the policy of

the actor is random, while the advisor is not, and shows actions to the actor that are generating good rewards. At this point, the critics of BDPI believe that all actions are uniformly good, since the actor has only seen good actions, thanks to the advisor's guidance. Due to the inner randomness of neural networks, the policy of the actor then arbitrarily concentrates its probability on one action, which is not necessarily good, producing the dip in the curve. It takes some exploration time for the actor to realize the suboptimality of that action, and to learn to choose better ones. Our hunch is shared by Wexler et al. [2022], in which the authors call this phenomenon Warm Start Reinforcement Learning Degradation.

Conclusion

In this section, we present a transfer learning method exploiting multiple advisors to tackle new tasks. The source tasks and the target tasks take place in the same state-space, but present crucial differences in their environment dynamics. Our motivating story is an agent having to exit several different offices. Even though it is likely that all offices of a building floor share a common layout, these offices might also be furnished differently, which makes navigation a unique experience in each of them for a reinforcement learner. We contribute a transfer learning method consisting in averaging the advice coming from multiple advisors, and providing this averaged advice to the fresh agent tackling a target task. This approach naturally balances advice versus *tabula-rasa* learning, depending on where the tasks are similar or not. We perform a thorough empirical evaluation of our method by: i) assessing the increase in performance gained thanks to the use of advice compared to none; ii) evaluating the benefit of having multiple advisors to average over instead of only one; iii) assessing the importance of implementing our learning correction to ensure stable learning. We saw that our contribution allows BDPI, an already highly sample-efficient algorithm, to be even more sample-efficient in a multi-task setting. This opens multi-task reinforcement-learning to areas, such as robotics, where many tasks have to be learned quickly.

5.4 Transferring Policies to Kickstart Learning in an Air Compressor Management Problem

In this section, we explore the use of transferring several previous policies to learn a new one faster, in an environment with continuous actions mimicking an industrial application. Three piston compressors are expelling air in a tank (see Figure 5.13); air is going out through a valve which progressively opens or closes following a demand curve. The compressors must provide a satisfactory pressure in between 3 and 5 bars,

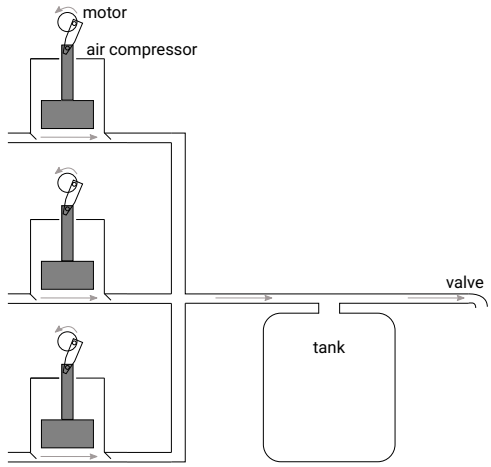

Figure 5.13: Description of the setup. Three piston air compressors are filling a tank of varying size. The valve progressively opens or closes to simulate a demand.

while conserving energy. A Reinforcement Learning agent, in this case Soft Actor-Critic [Haarnoja et al., 2018] (SAC), learns to control the three motors, each activating one of the piston compressor, while observing the current speed of the motors and the current pressure; the demand curve is unknown to the agent. Variants of the air compressor management problem are generated by varying the volume of the tank, and the compressing power of the three compressors. Each previous policy is trained on a different variant of the setting.

We apply our Continuous Policy Intersection (see Chapter 3, Section 3.2) to effectively distill multiple transferred controllers in the learning process of a SAC agent on a new variant of the air compressor management problem. We empirically show that Policy Intersection outperforms simply loading a transferred controller in the new task, be it with one or multiple transferred controllers, and that leveraging multiple controllers performs significantly better than leveraging only one transferred controller.

5.4.1 The Air Compressor Management Problem

A piston compressor is a type of industrial compressor, which increases the pressure of air enclosed in a cylinder. Air enters the cylinder through a valve, then is expelled under pressure by a piston operated by an eternal motor, through a second valve. In our particular setting, three compressors, each operated by its own motor, supply air to a tank. The air flows through a valve which is progressively opened or closed to

simulate a demand. The demand curve changes at each episode. Episodes last for 250 time-steps; at each time-step, the agent chooses the rotary speed of each of the three motors operating the compressors. The action space is continuous, and ranges from 400 revolutions per minute (RPM) to 600 RPM, with the possibility to completely stop the motor if the agent chooses to. When restarted, the motor goes from 0 directly to 400 RPM. At each time-step, the agent receives an immediate reward consisting in the negative amount of energy (in joules) that was consumed to power the motors during the time-step. The agent must keep the air under a pressure in between 3 and 5 bars within the system; going above 5 bars does not bring any added value; going under 3 bars triggers a backup policy. The backup policy is a naive controller that sets all three motors to their maximum speed for one time-step, which results in a very negative reward for the agent whenever the backup policy takes over. The agent observes the current speed of the motors, as well as the current pressure in bars, however, it does not know the demand curve. Although the reward is dense and highly informative, this setting consists in a challenging environment with continuous actions and partially observation states.

We generate several variants of this setting by randomly altering the size of the tank, as well as the compressors. The parameters determining the amount of air coming out of a compressor given x RPM are defined at the beginning of an experiment, using the following formula: $\rho \times (1.0 + \lambda \times \zeta \sim \mathcal{R} \in [0, 1))$, where ρ is a compressor parameter, $\lambda \in [0, 1]$ is the percentage of alterations applied to ρ, and ζ is a random float yielded by a generator simulating a uniform distribution. The tank volume is defined as $50 + 10$, in liters. The random float generator $\mathcal{R}$ takes an integer called *seed* s as parameter; we produce multiple variants of the environments to transfer across by varying the value of s (from 1 to 17).

This simulated setup is modelled as an MDP as follows.

Action space

The action space is continuous and consists of 3 real values, ranging from -1 to 1. It is considered best practice [stable-baselines] to have the action and state spaces be centered around 0, and of a range of as close to 1 as possible. Each of the 3 real values allows to set the target RPM of its corresponding compressor, linearly interpolated between 0 (off, for an action value of -1) to 100% (for an action value of 1).

The action space is discontinuous in two places: for every compressor, the actual effect of the action depends on the target RPM value it defines:

- Below 20 RPM: the compressor is turned off completely
- Between 20 RPM and the minimum RPM of the compressor: the target RPM is adjusted to the minimum RPM of the compressor

- Above the minimum RPM of the compressor: the action is left untouched

The effect is that the agent can produce low actions to indicate that a compressor should work at its lowest RPM, and an even lower action to indicate that the compressor should be turned off. This allows the agent to control the on/off status of the compressors, in addition to their target speed, with a single real value.

Observation space

The observation space consists of several real values that measure past tank pressures and RPMs. For every time-step, 4 real values are logged: the current pressure (in bars) in the tank, and the current RPMs of the 3 compressors. When producing observations, the environment looks N time-steps in the past to produce N real values corresponding to the tank pressures at these past N time-steps (so, this is a history of past tank pressures). The environment also looks M time-steps in the past to produce $3M$ real values corresponding to RPMs of the compressors during these past M time-steps. N and M can be distinct, and in our experiment, we use $N = 5$ past tank pressures and $M = 3$ past RPMs.

By observing past tank pressures and RPMs, the agent gets a feel of:

- How fast is the tank depleting, which allows it to approximate the derivative of the tank pressure (that only looks at the past), thereby adjusting the future RPMs.
- What is the demand of air in the recent past (by combining how the tank pressure changes and what the RPMs were), which may help the agent guess what the demand will be in the future if there are time-specific patterns in the demand.

This information is still not enough for optimal control, as it does not contain information about the future demand, but observing a history of past sensor readings has been shown to be one of the best approaches to learn in Partially Observable MDPs [citation], and is easy to implement.

Reward function

The reward function is the change in cost that occurs after a given time-step executes (so, after 60 seconds of simulated time after an action has been applied to the physical or simulated system). More specifically, the reward given after a time-step consists of two components:

1. Minus the amount of kilojoules consumed during the time-step. In the simulated setup, the power consumption of the compressors is approximated in a lookup

table (in watts). In simulation, we assume that a compressor instantly reaches its target RPM and its corresponding power consumption, so the amount of kilojoules is $60s \times P \times 0.001$, with P the power (in watts) obtained from the lookup table.

2. A turn on penalty when a compressor turns on. In addition to the point above (the power consumption during the whole time-step), a compressor that goes from off to on at the start of a time-step is considered to incur a cost equal to its maximum (max RPM) power consumption during 60 seconds. For instance, if the third compressor powers on, a reward of $-60s \times 2000W \times 0.001 = -120kJ$ is given to the agent.

5.4.2 Transferring Versus Loading (χ)

We set λ, the percentage of alteration applied to the parameters of the compressors, to 0.5. We train 16 different advisory policies, each trained on the environment with a different seed $s \in [1, ...16]$. We consider two settings in which Policy Intersection for continuous actions can be used: i) (*single*) a single advisor, with a different seed than the advisee, is exploited, ii) (*multiple*) all available advisors are exploited, except the one with the same seed as the advisee. To leverage multiple advisors, at each time-step, one advisor is randomly picked from the set of advisors. We compare transferring advisory policies using Policy Intersection to simply loading an advisor of a different seed and letting it resume learning.

The phenomenon observed in Section 5.3.2 appears again in both Figure 5.14 and Figure 5.15: a significant dip can be noticed in the learning curve, occurring only a few episodes after the beginning of learning. Moreover, this dip only occurs when using Policy Intersection; it is not present when loading an advisor. This is not surprising as, similarly to BDPI, SAC is an actor-critic, off-policy algorithm.

Policy Intersection, leveraging either a single or multiple advisors, outperforms loading an advisor and letting it resume learning. In addition, we observe in Figure 5.14 that exploiting multiple advisors instead of a single one significantly improve performance, especially when recovering from the dip in performance. This phenomenon is also noticeable in Figure 5.15, in which we compare leveraging 1 advisor, 4 advisors, and 16 advisors. However, there is a performance ceiling, reached by all settings in the long run, that does not seem possible to exceed by increasing the amount of advisors.

Conclusion

We applied Soft Actor-Critic to learn to control three motors, each activating a piston compressor in an air compressor management problem. In this problem, air is expelled by the piston compressors in a tank, while air is going out through a valve which

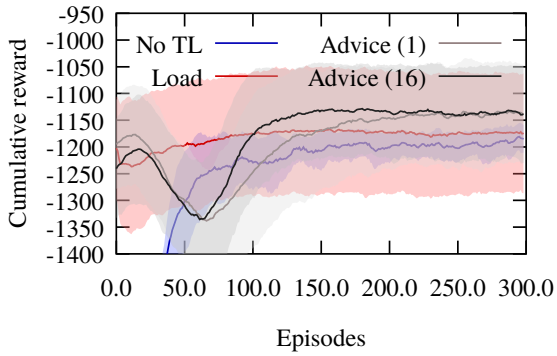

Figure 5.14: Comparison between transferring from 16 advisors, 1 advisor, loading one advisor at a time, or not using any transfer at all. Using Policy Intersection, either with a single or multiple advisors, outperforms loading an advisor and letting it resume learning. However, there is a significant dip in the learning curve shortly after learning begins when Policy Intersection is used.

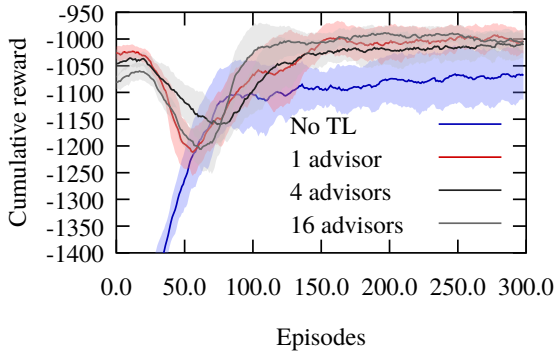

Figure 5.15: There is a significant difference in performance between using 4 and 16 advisors, especially when recovering from the dip in the curve. This difference lessens in the long run.

progressively opens or closes following a demand curve. The goal of the compressors is to provide a satisfactory pressure in between 3 and 5 bars, while conserving energy. Moreover, we considered the problem of learning a new controller based on previously

learned ones on different variants of the air compressor management problem. To tackle this Transfer Reinforcement Learning problem, we used Policy Intersection, a Policy Shaping method, allowing one or multiple already acquired policies to distill their knowledge in a new policy, through guiding the exploration strategy of the agent. We empirically showed that this approach outperforms simply loading a transferred controller, and that exploiting multiple transferred controllers instead of a single one can significantly improve performance.

5.5 Reinforcement Learning Web-Service with Transfer Across Users

At time of writing, Reinforcement Learning is getting an increasing amount of attention from companies, eager to apply the methods on their in-house problems, but not necessarily willing to invest the time and effort to choose and tune the RL algorithm best suited to their needs. Although RL is certainly not the solution to all problems, there exist some business and industrial applications that could benefit from an RL-based approach. Most current implementations of Reinforcement Learning agents consider that one agent interacts with one environment, and both the agent and the environment run on the same machines. Previous work, such as RL-Glue [Tanner and White, 2009], went a step in the direction of allowing the agent and environment to be different processes on a computer, but a wider separation of the agent and environment is much less common. Nevertheless, we believe that a good solution for companies would be to externalize the RL algorithm content and processing, so to not have to administer the machine on which the algorithm runs, and keep up with the literature to regularly update the algorithm itself.

To make the use of RL methods more accessible to a non-expert audience and to meet the above-mentioned requirements, we introduce Shepherd, a web application, implemented in Python with Django3. Shepherd allows anyone with an internet connection to remotely query a Reinforcement Learning agent for actions, and allows multiple clients (each with their own instance of an environment) to interact at the same time with a single agent. It consists in a single agent, multiple executions setting, comparable to what A3C proposes for compute-efficient Reinforcement Learning [Mnih et al., 2016]. The novelty of Shepherd is that it does not rely on the A3C algorithm, but instead is compatible with any Reinforcement Learning algorithm.

In addition to regularly updating the set of available RL algorithms on the server side, we implemented the Shepherd framework so that agents can transfer knowledge with each other following our Actor-Advisor architecture. This can be interesting in cases where multiple users interact with a given application; in such setting, there are global policy elements that can be learned and be applied to all users, and each user

can personalize their own local policy with their preferences. Whenever a new user visits the application, a new execution is instantiated and loads a saved version of another execution if available. This kickstarts the learning of this new execution, and allows the user to benefit from the knowledge already acquired by executions trained by previous users.

We evaluate the individual performance of PPO agents sharing knowledge with each other, while all simultaneously learning Lunar Lander. Finally, we describe a novel environment in the form of a web application, called Bored in the city, inspired by Tickle [De Troyer et al., 2019]. Tickle is a mobile application suggesting small challenges to school students, aiming at re-engaging youngsters in learning. Bored in the city follows the same idea, and suggests places, e.g., restaurants and shops, to visitors of Brussels. All places are centered around the de Brouckère, Sainte Catherine and Gare Centrale metro stations, and are close enough to each other so that users can walk from one place to another, without having to take the metro or buses between places. Although we did not undertake a proper user experiment to evaluate Shepherd on it, Bored in the city consists in an ideal application for Shepherd. Each visitor would interact with the Shepherd agent individually, while the agent learns what place to suggest, based on its experience with all the visitors it has met so far.

We now describe the architecture allowing for one RL agent to collect data from multiple users, and learn a policy that can behave in front of multiple users.

5.5.1 The Shepherd Architecture

By the term "user", we mean a person that possesses an environment applicable to the use of an RL algorithm, a User denotes the object representing a user in the database, and a "client" is the program running on the user's machine and communicating with a server.

Shepherd is a web application using the Django Python web development framework [Forcier et al., 2008] [2]. It acts as a bridge between web clients, that connect to it over the network (using JSON commands sent over HTTP), and state-of-the-art Reinforcement Learning agents available in the Stable Baselines 3 [Raffin et al., 2019]. Shepherd presents itself to the RL algorithms as a fully standard OpenAIGym environment [Brockman et al., 2016]. At the core of Shepherd, the Actor-Advisor framework is used to allow RL agents (all under one Shepherd agent), each trained by a different user of Shepherd, to advise and help improve each other. This is why Shepherd is comparable to a single agent, multiple executions setting, without the need to fundamentally modify the RL algorithms from Stable Baselines 3.

In a typical use case of Shepherd, the problem to be solved by an RL agent is located on the side of the user; a company may want a solution for efficiently managing

[2] https://www.djangoproject.com

the heating of its buildings, for instance. Sensory data about the current temperature in the buildings and the setting of the thermostats is sent to the Shepherd server by the client program, running on a machine administered by the company. In return, an action describing the new setting of the thermostats is received, and is transmitted to the controller of the thermostats. In other words, the actual environment, in which the actions of the agent are executed, takes place on the user's grounds. Moreover, to follow the client-server communication convention, actions are only sent "on demand": the environment prompts the agent for actions, despite it generally being the other way around in standard RL applications. This way, users can request for an action whenever they want, instead of being bombarded with unsolicited actions. The company chooses the frequency at which it wants the setting of its thermostats to be updated, and it can quit sending sensory data to the Shepherd server at any time.

We describe below the components of Shepherd, namely: i) the database allowing to keep track of users, and of their RL agents; and ii) the communication workflow between client and server, letting the environment running on the client side to sporadically send observations to an RL agent on the server side, and getting actions in return. In Section 5.5.1, we detail how several instances of a Shepherd agent can advise each other to help each other learn faster, as when used in an application like Bored in the city.

For a user environment to be able to benefit from our service, the only requirements are that observations can be extracted from the environment, and that the environment can take actions as input. Instead of learning how to properly apply RL algorithms themselves, or hiring new expert staff, non-experts eager to try RL on their applications can ask us to hyperparameter-tune and run the RL algorithms on our server. The only contribution left from the user is to implement a client establishing the connection with our server, sending observations and receiving actions. However, this part is relatively trivial to implement, and we provide an example in Section 5.5.1.

Database

We now describe the different tables of our database (see Figure 5.16), as well as the general workflow of the registration of new users and RL agents in our system.

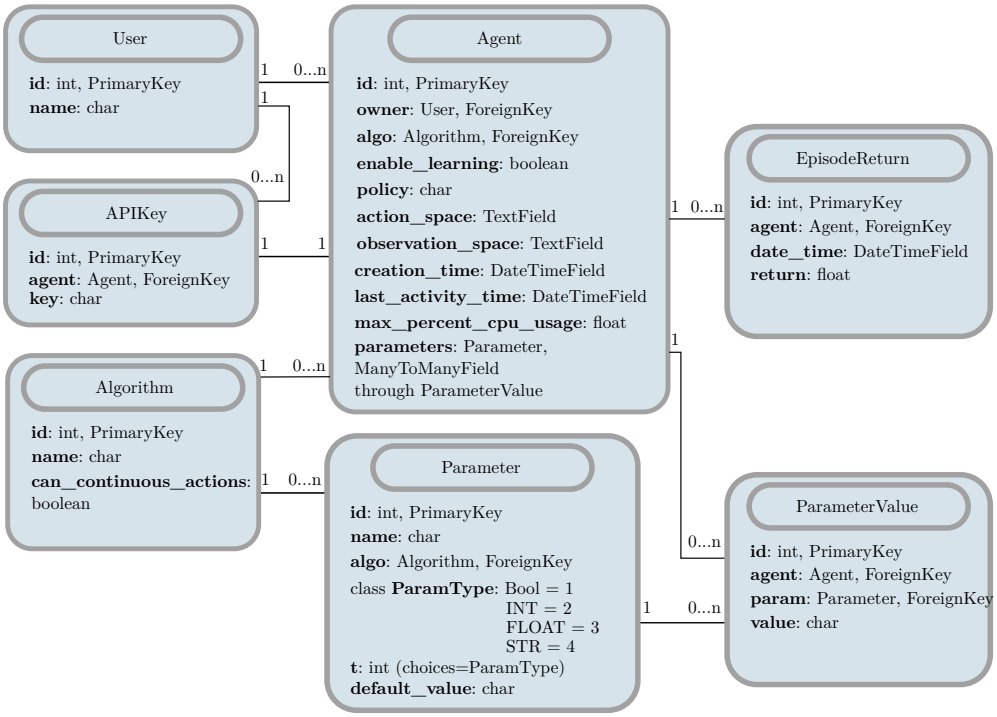

Figure 5.16: The database structure of the Shepherd framework. A User can have several Shepherd Agents, which belong to that sole User. Each agent is running a specific RL algorithm on a environment provided by the user; the action and observation spaces of that environment are stored as attributes of the Agent. Associated to one given Shepherd Agent are ParameterValues; it can be set by the user, otherwise the predefined default value from the corresponding Parameter is used.

User

New users are added to the database by an internal staff member through the admin page. The onfly information required is a name to identify the user; we use API keys to log users in instead of a user password. A User can possess more than one Agent (one per environment that they have, or one per different algorithm on one environment to compare algorithms, for instance).

Agent

When a user wants to launch an RL algorithm on their environment, a new Shep-

herd Agent object is created and identified by both an APIKey and the owner of the agent. The specification of the action and observation spaces of the user's environment are required for the initialization of a Shepherd Agent. New Shepherd agents are created through the admin page, as well as the parameter values to configure the algorithm run by an agent. The difference between a Shepherd agent and a standard RL agent is that several standard RL agents can execute one Shepherd agent: a Shepherd agent stores a configuration, specifying which algorithm must be executed, the parameters, as well as the latest generated model and results obtained. There are a few "special actions" that the user can do through the admin page; they can download data from the agent, such as the episode rewards and the agent's model, and reboot its learning, which is necessary when the environment has been modified and that previous results become irrelevant. If the enable learning field is set to False, then the current agent's model executes without updating its weights anymore; this mode is typically used to evaluate a policy after some time spent training it. Given one algorithm, especially when using the Stable Baselines 3, a type of policy network [3] can be specified through the policy field, depending on the observation provided by the environment. The creation time and last activity time fields exist for user's information about their agent; the max percent cpu usage indicates how much of one cpu, or how many cpus can be allocated to a given Shepherd agent to execute. Since one Parameter, such as the learning rate, is used by multiple Agents, but each can have a different value of that parameter, and that an Agent generally uses several parameters, the Agent table includes a custom many to many field linking Agents with Parameters, through the ParameterValue table.

Algorithm

An Algorithm has a fairly brief definition in our database; it has a name, often the abbreviated version of the name of an RL algorithm, such as PPO or A3C and a Boolean flag indicating whether it is compatible with environments with continuous actions or not. An Algorithm also has several Parameters associated to it. Algorithm objects are preexisting the creation of Users and Agents; we pre-populate the database with RL algorithms from Stable Baselines 3, but adding new algorithms that are not from Stable Baselines 3 is trivial. Hence, we also made BDPI [Steckelmacher et al., 2019] and its extension using the Actor-Advisor available on Shepherd.

Parameter value

When a new Shepherd agent is created, an RL algorithm must be chosen to be run by this agent. If they want to, the user can configure the algorithm through

[3] See https://stable-baselines3.readthedocs.io/en/master/guide/custom_policy.html

setting the value of some parameters, via the addition of ParameterValue objects in the database[4]. This comes in handy when the user wants to try parameter values different from predefined default ones. The value chosen is stored in value_int, value_float, value_bool (exclusive) or value_str, depending on the type of the corresponding Parameter object; all three other value attributes are set to null. A ParameterValue is associated to one Shepherd Agent, and to one Parameter, described below.

Parameter

Each RL algorithm in the database uses a multitude of parameters (e.g., learning rate, batch size, γ, etc), all associated to that one algorithm. The type of the value this parameter can take is defined by an Integer $\in [1, 2, 3, 4]$; if type is equal to 1, the value of the parameter must be an Integer, if type $= 2$, the value must be a floating point, etc[5]. Only one of the attributes value_int, value_float, value_bool and value_str contains a value different than null, depending on the type attribute. Similarly to Algorithms, Parameters are pre-existing ParameterValues; we populate the database with parameters used by most RL algorithms, and set their default values, before the addition of Users and their Shepherd Agents.

Episode return

Multiple EpisodeReturns are associated to one Shepherd Agent. Each EpisodeReturn stores a float (the sum of all rewards collected during one episode), and a date time field. EpisodeReturns are used to plot the Agent's learning curve, displayed on the admin page.

API key

To avoid the need for a user name and password, we use generated API keys by the UUID module [Kuchling and Zadka, 2012]. The particularity of our usage of API keys is that one APIKey is also associated to one Shepherd Agent. When logging in to Shepherd, the client sends an API key to the server, which retrieves the corresponding APIKey object, linked to the user's corresponding Agent. In the case where a user has multiple Shepherd agents, they would also have multiple API keys, one per agent. The advantage of this setting is that if one API key is leaked, only one of the user's Shepherd Agents is leaked.

[4] Users can also ask experts on our side to suggest values for the parameters of the RL algorithm they want to try out, depending on the particular environment they have.

[5] We have arbitrarily chosen these integers to type relations; they do not hold a particular meaning.

Client-Server Communication

Once the database is populated with at least one User, owning at least one Shepherd Agent, a client can start sending observations to our server, and receive actions in return. Users must first log in by sending their API key, as mentioned above. Once the Shepherd agent linked to the particular API key is retrieved, a session key is created, and a process from the Shepherd agent's process pool is allocated to that session. The process pool of a Shepherd agent is a class keeping track of all currently running processes (or instances) of a Shepherd agent: it allocates available processes or creates new ones when users request a session; it deletes processes from the pool that have been inactive for a given amount of time; etc. More importantly, since the process pool keeps all running instances of a Shepherd agent "under one roof", it facilitates the exchange of advice between instances, making knowledge transfer between instances possible throughout their learning (see next section).

Before the process for the newly created session starts learning, it looks for the latest save of the processes previously learning under the corresponding Shepherd agent, to pick up where the last one ended. If one such save is found (i.e., a zip file in which neural networks weights have been saved), the process loads it, otherwise, learning is started from scratch. Each running process regularly saves its network weights in a directory shared by all processes of a Shepherd agent.

In a conventional reinforcement learning setting, the agent is usually the driving force of the interactions between the agent and the environment. In a typical RL workflow, the agent generates an action, then prompts the environment for an observation, in exchange of that action. In our Shepherd framework, on the other hand, we want the environment on the client side to prompt the agent on the server for an action, in return of an observation. Because of that role reversal, which is most unnatural for a standard reinforcement learning algorithm, we implement a mock gym environment, called ShepherdEnv-v0. Each process keeps an instance of that mock environment as an attribute and communicates with it as if it were the environment on the client side.

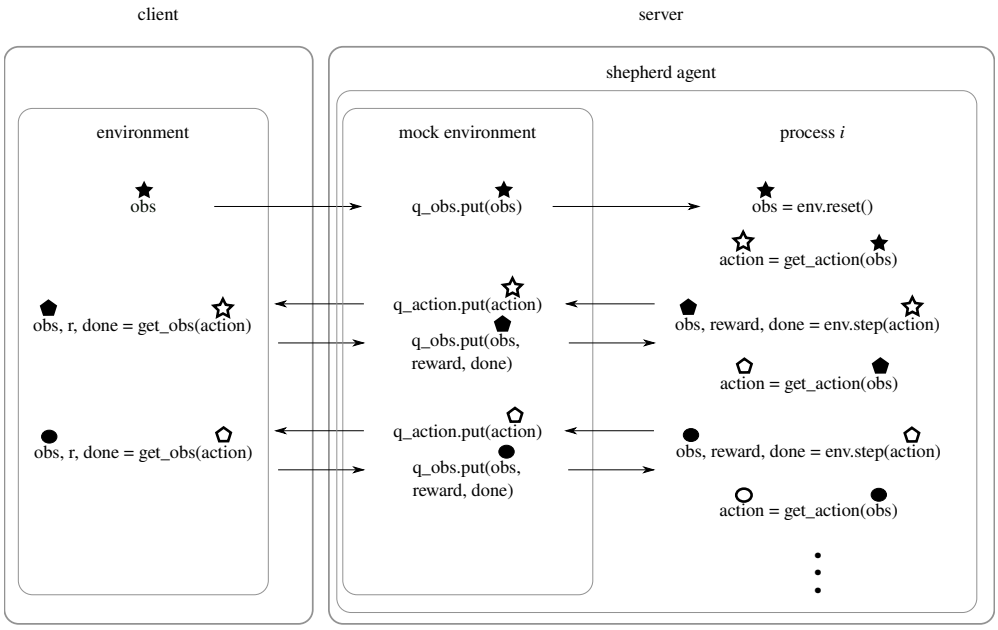

Figure 5.17: Time diagram of the first few exchanges between a client and the server. The real environment on the client side produces the observations (abbreviated as "obs"), the rewards ("r"), and the done flags. The little shapes above the observations and actions indicate how observations travel from the client to the process running on the server, and actions are relayed back to the environment on the client side. Queues q_obs and q_action are used to buffer observations from the client and actions from the process on the server side until they can be delivered to their recipients. The reinforcement learning process on the server side expects to be able to call the environment with a step() function, taking an action as input, and returning an observation. However, in a conventional client-server relationship, it is the client that prompts the server, not the other way around. Hence our need for a mock environment on the server side, so that our Shepherd framework remains compatible with current RL algorithms.

The time diagram in Figure 5.17 illustrates a few exchanges between client and server, once the user is logged in, starting at the first observation sent by the client to the server. We keep the description of the actual environment on the client side extremely vague; all we care about is that it produces observations, rewards and done flags. First, the RL algorithm resets the environment, which returns the first

observation. Thankfully, the client is likely to start the exchange with an observation, expecting an action in return. The client gets its action when the algorithm calls the step() function on the mock environment. The learning process is stalled until the client sends a new observation, etc. When done is True, indicating that the episode is over and that a new one can start, the server responds with None instead of an action, and the server waits for a new first observation. The mock environment uses queues to store observations coming from the client, and actions coming from the RL process, until these information are claimed by their effective recipients. This way, it meets the expectations of the standard RL algorithms running on the server side, by acting like a regular gym environment. Note however that the mock environment does not simulate the actual environment on the client side; it merely serves as an interface between the real environment and the RL process.

Transfer Between Processes

Our Shepherd framework allows for two main use cases: i) a simple case where one learning execution tackles a given task; the user can train its agent as long as they wish; its learning can be interrupted at any time, and resumed later by loading its saved model, and ii) a more complex case where multiple learning executions learn the same task in parallel.

An example of case ii could be multiple users of a web application, such as Bored in the city (see Section 5.5.3 below), each logging in with the same API key, training a different learning execution of the same Shepherd agent. Although all learning executions seem to tackle essentially the same task, each user might respond slightly differently to their learner's actions. This leads to several different policies to be learned: one global policy that is in common to all users of the application, and one local policy per user. This motivates the use of some knowledge transfer mechanism between learning executions under a given Shepherd agent, so that newer executions can benefit from older, better trained ones, and to potentially improve the individual sample efficiency of all learning executions. Note that case ii assumes that all learning executions use the same observation space and action space, and that each learning execution acts in its own instance of the environment; if learning executions interact with each other on the field, this makes the problem multi-agent.

As we have hinted at in the previous section, the process pool of a Shepherd agent keeps track of all learning executions of a given Shepherd agent, and facilitates the exchange of advice between learning executions. At each time-step, once an observation is received by the server from the client, a process other than the current session process is picked at random and queried for advice (see Figure 5.18). This advice is then embedded in the observation, relayed to the RL algorithm executed by the session process. Currently, the RL algorithms supported by Shepherd all come from

the Stable Baselines 3 Raffin et al. [2019]; in the experiment in the next section, we use their PPO implementation, extended with the Actor-Advisor. The integration of advice in the Stable Baselines 3 implementations is rendered relatively minimal and straightforward thanks to the recent addition of MultiInputPolicy [6], a type of policy that accepts observations in the form of a dictionary. Using dictionaries of observations, instead of merely one observation, makes learning from several different types of sensors possible, for instance, such as using both camera images and distance sensors to solve a navigation task. In our case, we fill the observation dictionary with both the actual environmental observation, as well as an advice vector coming from another learning execution. We then ask the neural network to ignore this advice vector when feeding itself with the observation, and relay it to the action selection and update rule functions, as detailed in Chapter 3.

By picking one process from the pool at random at each time-step, all processes under one Shepherd agent eventually advise each other to solve a given task. In the case of a web application with multiple users responding differently to recommendations, or of an industrial task with slight changes between instances of the environment, allowing learning executions of one Shepherd agent to share knowledge with each other can help speed up individual learning, and provide already well-trained behavior to later added processes.

[6] See https://stable-baselines3.readthedocs.io/en/master/guide/custom_policy.html#multiple-inputs-and-dictionary-observations

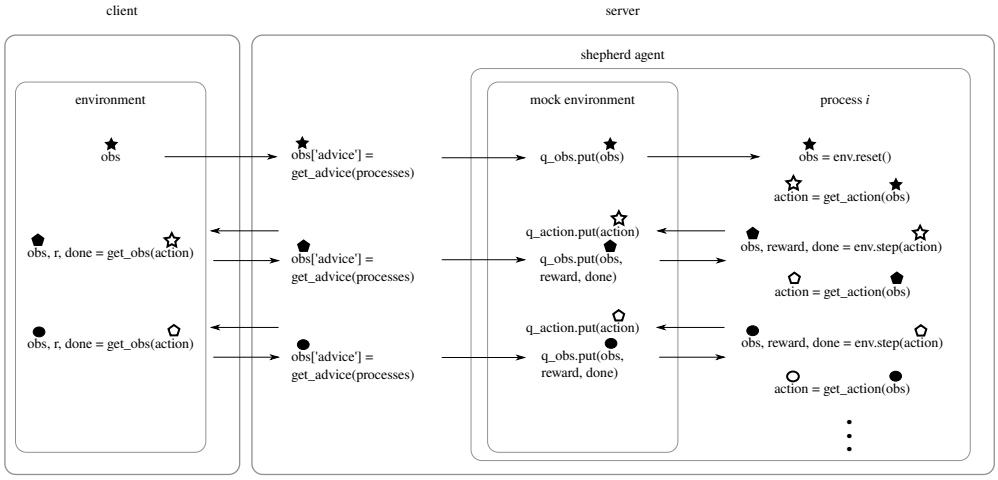

Figure 5.18: Same time diagram as above, but including the use of advice. Each time an observation is received from the client, the process pool of the corresponding Shepherd agent adds an advice vector in the observation before passing it to the mock environment. Advice is produced by picking one of the processes in the pool at random and querying it for advice. The advice vector is then leveraged internally by the RL algorithm executed by process i.

In the next section, we evaluate the ability of our Shepherd architecture to allow for effective transfer between several learning executions under one Shepherd agent, on the Lunar Lander environment. We empirically show that having several processes learning in parallel, each on their separate instance of the environment, while exchanging advice with each other not only does not impair individual learning, but can even non-negligibly improve individual sample-efficiency. We keep a throughout evaluation of Shepherd on a more complex setting involving human users, such as Bored in the city (described in Section 5.5.3), for future work.

5.5.2 Evaluation on Lunar Lander

To asses whether our Shepherd framework truly delivers what it promises in terms of effective transfer, we compare the rewards obtained by several PPO agents learning simultaneously under one Shepherd agent on discrete Lunar Lander, versus that of a single PPO agent. Agents learning in parallel belong to the same pool of processes. As a result, each time a learning execution receives an observation from the client, that

observation is first handled by the process pool class which augments the observation with an "advice" field. This advice is produced by one other process from the process pool picked at random. Since a different process is picked to produce advice at each time-step, the learning execution is eventually advised by all other processes from the pool it belongs to, given enough time-steps.

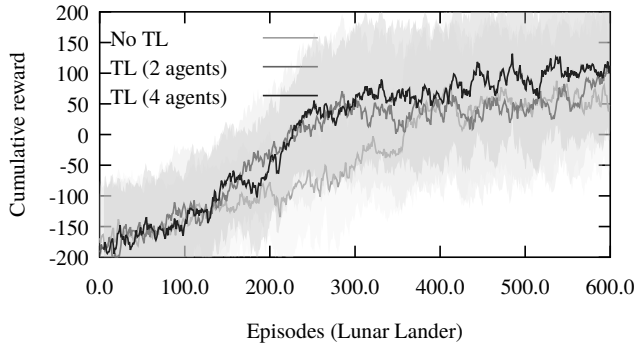

Figure 5.19: Performance of learning executions under a Shepherd agent learning discrete Lunar Lander. We test three different settings: a single execution, two parallel executions, and four parallel executions. When only one execution is running, no transfer occurs, in contrast to when there are multiple executions. Each curve is the average performance of 8 runs for each setting, and shows the individual performance of a learning execution when several are learning in parallel. For the discrete Lunar Lander environment, adding merely one extra learning execution adds a significant boost in individual sample-efficiency, and the curve resulting from four parallel executions remains on top during the whole experiment.

In our experiment, for which results are shown in Figure 5.19, all executions start from scratch and execute actions in their own instance of the environment, but are advised by the other executions, and advise them as well. When a single PPO execution is launched under a Shepherd agent, no transfer occurs, since there is no other execution to share knowledge with. Interestingly, although all executions have the same "age" and level of knowledge of their environment, they seem to benefit enough from each other's advice for a significant individual sample-efficiency increase to occur in between 200 and 400 episodes. Moreover, the individual performance of one of four parallel executions remains on top during the whole experiment. Such improvement in sample-efficiency for the discrete Lunar Lander environment, with no difference between environment instances, is a pleasant surprise; however, the real challenge

for Shepherd is to perform positive transfer in settings where the environments are different.

We conclude this chapter with a presentation of Bored in the city, an environment involving human users in the form of a web application, allowing multiple users to train one Shepherd agent (through each training their own learning execution), and new incoming users to benefit from an already well-trained agent. Bored in the city is the proof of concept of an environment on which a tool such as Shepherd could be leveraged to improve user experience.

5.5.3 Bored in the city: a Web Application to Visit Brussels

Bored in the city is a serverless web application in Javascript, that lists over one hundred curated places, which we personally selected[7]. Amongst these places, there mainly are Belgian and Asian restaurants, various shops, touristic spots, cafés and bars in Brussels. We manually annotated each place with a type, from which there are seven: restaurant, Asian restaurant, shop, thriftshop, cafe, bar, site-seeing. These seven types are certainly not an exhaustive list of all types of the places that can be found in Brussels; we chose these types as they would best represent our selection. Users can visit the website without the need to create an account, and start requesting suggestions coming from a Shepherd agent. The goal of the Shepherd agent is learning to give suggestions that lead the users to the kind of places they like. The website includes the following features:

- the user can request a place to be suggested to them by clicking on "Next place". When a place is suggested, its name and full address are displayed, and a marker is shown on a map, using the Google Maps API;
- a route from the user's current location to the place is displayed on the map when "Show directions" is clicked [8];
- the user can add a place to their favorites by clicking on "I love it, add to favorites";
- the user can revisit their favorites by clicking on "Show favorites". Once they wish to go back to being requested new places, they can click on "Hide favorites".

[7] As a result, the places listed in Bored in the city may be biased by our personal preferences, which tend to be Asian restaurants and second-hand clothing stores.

[8] The application asks for the user's geolocation data, but does not refuse to work in case geolocation is unavailable, but then assumes that the user's current location is the last place they favorited.

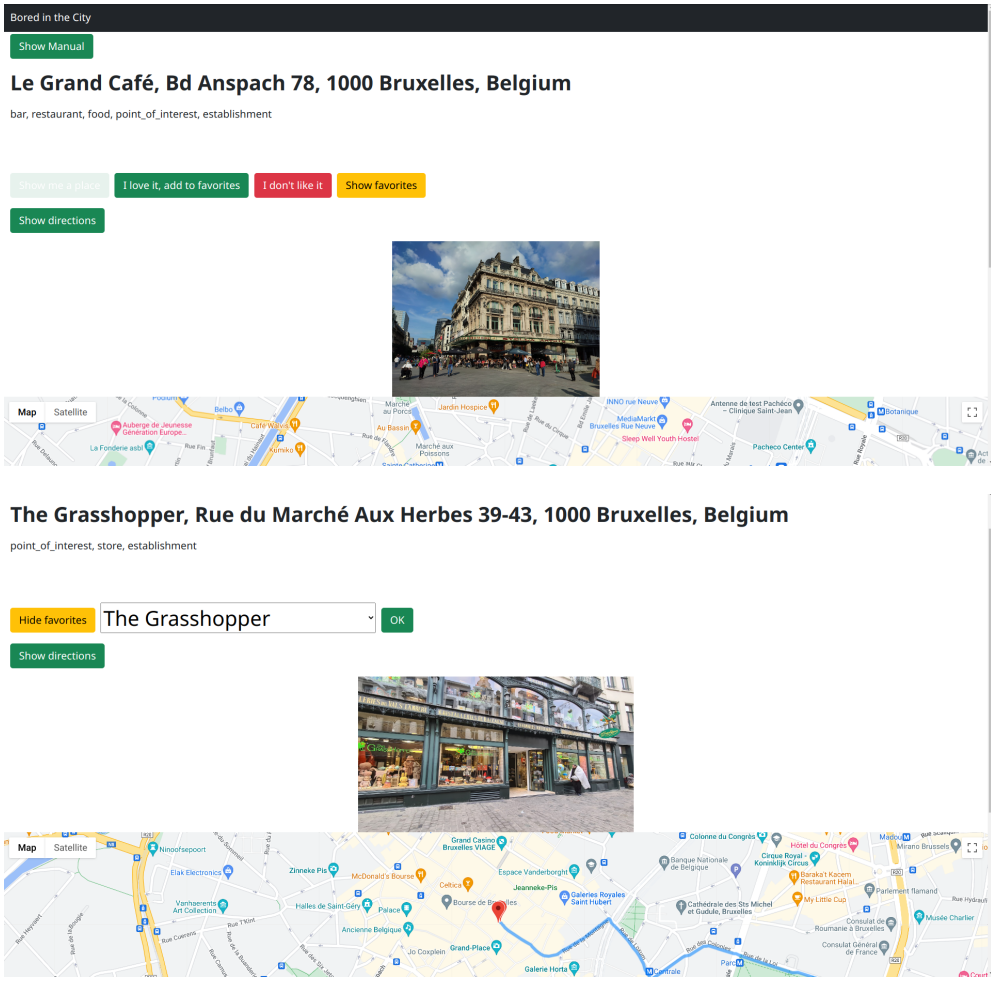

Figure 5.20: A potential suitable environment for Shepherd consists in a web application suggesting places to visitors of Brussels, such as shops, restaurants and site-seeing spots. *Top:* The main page displays the name and address of the suggested place, a picture, and map on which the suggested place is marked. *Bottom:* The application is leveraging the Google Maps API to retrieve information about the place, show a map, and show directions from the user's current location to the location of the place. Places are suggested to the user by a Shepherd RL agent.

Coupled with Shepherd, Bored in the city is a reinforcement learning-powered tour guide of Brussels; it is intended for people who are visiting Brussels, specifically the area between Sainte-Catherine and de Brouckère. In our original use case, as they would walk around, users would request place suggestions throughout the day, and perhaps physically go to some of the suggested places. Users reward the agent after they have visited the place and if they liked it. Episodes are meant to represent full days or nights, and last seven hours.

However, we had to rethink our original use case, as allowing users to reward the agent *only after they have physically been to a suggested place* [9] leads to potentially hours-long episodes with very sparse rewards. As a result, we modified our target use case: instead of "liking" a place only after having visited the place, users can add it to their favorites (to visit them later, perhaps). Instead of hours long, episodes last only one time-step and end right after the user has either favorited or dismissed the suggested place, which transforms our original sequential decision problem into a contextual bandit. A typical usage of Bored in the city becomes requesting places suggestions, then either adding the place to a favorite places list, or dismissing the place, then requesting another place, and so on. Users can still use the application on their phone while on the field, as originally intended, since we did not remove the "Show directions" feature, which remains handy to walk from a location to another. However, in contrast to the original setting, users are likely to reward the agent a lot more often since they don't have to physically visit the place first, and can do it from their couch. This lessens the commitment asked from volunteers. Nevertheless, we believe that our original use case idea would be feasible and interesting in a museum guide setting, for instance, in opposition to visiting almost a whole city.

From the perspective of the RL agent, the agent observes the user's current location, their previous location (both in latitude and longitude coordinates), the type of their previously suggested place (one-hot encoded), what type of place they favorited so far (for each type, the normalized number of favorited places belonging to that type), the time of the day (whether it is the morning, afternoon, evening or night, one-hot encoded), and the day of the week (one-hot encoded). By observing geolocation data, the agent can potentially learn whether the user is more likely to favorite a place if it is close to their current location. The agent can also learn user's preferred type of place, and whether the time of the day, and day of the week influence the rewards. For instance, restaurants may be more likely to be favorited around lunch time or dinner, cafés in the morning and early afternoon, and bars in the evening; people tend to party on Fridays and Saturdays evening. There are as many actions as there are places listed in Bored in the city, that is, slightly over one hundred. We

[9] We did not implement any mechanism to verify whether the user is physically at a suggested place, and prevent them from rewarding the agent if they are not at the place. It is simply implied in the phrasing of the button's label that the user would first visit a place, then reward.

have pondered the possibility to cluster places in seven groups, one per type of place, and to restrict the number of actions to seven, but we eventually chose against it as it leads to a too important information loss in our opinion.

For sufficiently informative results to be obtained, we estimate that a user experiment on the Bored in the city environment would require at least 20 participants, spending around a week visiting places in Brussels. Unfortunately, we did not manage to organize such experiment during our PhD study, hence results on this problem are not reported in this thesis, but we hope that the opportunity to carry it out will present itself in the near future. When considering RL environments involving human users, the feasibility issue of evaluating contributions becomes much more prevalent than when no human intervention is required. This is particularly regrettable as systems involving humans are likely to become the norm, hence the need to regularly test contributions in user experiments. Specifically, when designing Bored in the city, we felt compelled to make an environment not only interesting in terms of reinforcement learning problem for the agent, but also in terms of user-friendlyness for users, in order to keep volunteers engaged. However, the more sophisticated Bored in the city became, the more user involvement it required, and the more difficult to organize the user experiment revealed itself to be.

6 | Self-Transfer

This chapter is drawn from our publication, Plisnier, Steckelmacher and Nowé, *Self-Transfer Learning*, presented at the Adaptive and Learning Agents (ALA) workshop, in 2021.

A Transfer Learning setting often involves a *source* task and a *target* task; first, the agent learns the source task, then the knowledge acquired in the source task is intelligently leveraged by the agent while tackling the target task, using a Transfer Learning method [Taylor and Stone, 2009; Zhu et al., 2020]. The aim of TL is generally to make an RL agent learn new tasks faster by allowing it to reuse previously learned knowledge efficiently.

In this chapter, we introduce Self-Transfer Learning, a TL setting in which the source task and the target task are the same. We let an RL agent learn a task for a short period of time, then freeze it and use it as an advisor for a fresh agent, that learns the same task from scratch, until reaching a good policy. The training time allocated to learning the frozen policy is a fraction of the time allocated to the fresh policy. We evaluate Self-Transfer both on environments with discrete and continuous actions, on which we test our novel implementation of Policy Intersection (PI) for continuous action spaces (see description in Chapter 3, Section 3.2). Although transferring a policy from a task to itself might seem redundant, we present preliminary results in Sections 6.2 and 6.3.2 empirically showing that this approach brings a non-negligible gain in sample-efficiency. We suspect that Self-Transfer positively impacts exploration, both in policy space (exploring more states) and in parameter space (the fresh agent's policy is initialized used a fresh random seed).

In addition to showing the benefits to be obtained from our implementation of Self-Transfer, our empirical results in Sections 6.2 also show that our particular implementation of the Policy Intersection idea outperforms two other algorithms that can be used in a Self-Transfer setting: Probabilistic Policy Reuse (PPR) [Fernández and Veloso, 2006], and Dual Policy Distillation (DPD) [Lai et al., 2020], that we review in Sections 2.5.1 and 6.1.2 respectively. For the discrete actions case, we compare the Actor-Advisor to PPR and directly loading the advisor, using Bootstrapped Dual Policy Iteration [Steckelmacher, 2020]. The way PPR can be implemented on top of BDPI is detailed in Section 6.3.1.

6.1 The Self-Transfer Setting

In addition to extending Policy Intersection to the continuous actions case (see Section 3.2), our contribution evaluated in this chapter is the Self-Transfer framework, in which two policies sequentially learned in the same environment are used in a Transfer Learning setting, to increase sample-efficiency and final policy quality.

Our Self-Transfer procedure works as follows: an RL agent is launched in an environment as usual. After a given amount of episodes I (at the designer's discretion), the agent's learning is interrupted, and its actor π_L is deep-copied into π_A. Then, π_L and, for actor-critic algorithms such as Soft Actor-Critic [Haarnoja et al., 2018] [1] and Bootstrapped Dual Policy Iteration, the critics, are re-initialized to fresh random weights. Learning then resumes, except that the fresh agent π_L is now advised by π_A, using the Actor-Advisor in the discrete actions case, and continuous Policy Intersection in the continuous actions case. Pseudocode for the Self-Transfer setting is shown in Algorithm 2.

6.1.1 Conventional Transfer Versus Self-Transfer

Due to the relatively odd nature of the particular setting in which we use Probabilistic Policy Reuse and Policy Intersection [2], i.e., the transfer of knowledge acquired in a given task to the same task, it is a little difficult to find directly comparable existing work. Existing Transfer Learning work usually focuses on transferring knowledge across tasks with different goals or differing environmental dynamics [Fernández and Veloso, 2006; Zhang et al., 2018; Taylor et al., 2007; Plisnier et al., 2019b, only a few examples], while we transfer from a task in a given environment to the same

[1] The specific SAC implementation we use can be found at: https://github.com/Rafael1s/Deep-Reinforcement-Learning-Algorithms/tree/master/Ant-PyBulletEnv-Soft-Actor-Critic

[2] Although PPR [Fernández and Veloso, 2006] and PI [Griffith et al., 2013] find their roots in traditional Transfer Learning, and learning from human interventions respectively, these methods are general enough to be applied to a variety of other problems.

Algorithm 2 Self-Transfer

Require: I is the amount of episodes reserved to training the advisor π_A
  Initialize critics and actor π_L
  for every episode $e = 1..\infty$ **do**
    if $e = I$ **then**
      $\pi_A \leftarrow$ a copy of π_L
      Reset π_L and critic networks
    end if
    for every time-step t until the end of episode **do**
      if π_A exists **then**
        Sample a_t with the Actor Advisor given π_L, π_A, s_t
      else
        Sample a_t from $\pi_L(s_t)$
      end if
      Periodically learn using the equations of the RL algorithm
    end for
  end for

task, in the same environment; only the initialization of the agent's networks changes. Methods using more than one actor to improve exploration of a given environment are somewhat related to Self-Transfer, such as A3C [Mnih et al., 2016], Multi-Agent RL settings in which agents actively share knowledge with each other [Omidshafiei et al., 2019; da Silva et al., 2017; Hadfield-Menell et al., 2016], and Dual Policy Distillation (DPD) [Lai et al., 2020] (see Section 6.1.2).

In contrast to DPD, but similarly to Self-Transfer, Self-Imitation Learning [Oh et al., 2018] does not require a second agent to improve exploration; the agent learns to reproduce its own past good decisions to deepen exploration. In Oh et al. [2018], a preference for previously chosen actions that lead to a greater return than the current value estimate for a given state is integrated in an actor-critic loss. Although this concept is very close to ours, Self-Transfer leverages knowledge from past experiences produced by an agent with a different initialization than the agent currently learning, while Self-Imitation never resets the agent. We suspect that agents with different initialization can lead to distinct, complementary experiences of the same environment, and thus achieve a more thorough exploration when joining knowledge.

6.1.2 Dual Policy Distillation

In addition to Probabilistic Policy Reuse (PPR, see Section 2.5.1), one of the closest existing technique to our contribution, that is using more than one actor, is Dual

Policy Distillation [Lai et al., 2020]. DPD launches two agents (both using the same RL algorithm, either DPG or PPO agents in the original paper) at the same time in *two instances of the same environment*, but with different initializations of their neural network weights. As the agents learn to solve the same task in parallel, they optimize a distillation objective that incites the policies of the two agents to remain close to each other. A theoretical justification of why the two policies complement each other knowledge-wise is provided in Lai et al. [2020]: as with our Self-Transfer setting, having more than one policy intervening in the action selection procedure (or learning procedure) limits convergence to poor local optima. However, methods leveraging more than one agent/actor have two potential downsides: i) their deployment to real-life settings is likely to be more difficult if they require the two actors to execute actions *concurrently*, in two instances of the environment, and ii) such methods assume that all the policies are RL agents, hence one cannot be easily replaced by a fixed transferred policy. Algorithms such as PPR and PI do not make such assumptions about the advisor policy, allowing it to be from any source.

6.2 Self-Transfer in Continuous Action-Space Environments

We present a preliminary evaluation of our contribution by applying Self-Transfer on three Pybullet [Coumans and Bai, 2019] continuous control environments: Ant, a four-legged insect-like creature; Half-Cheetah, a two-legged creature; and Hopper, a single disembodied leg hopping away. In these three environments with continuous actions, the goal is generally to learn to move forward as fast as possible, without falling. We evaluate our Self-Transfer setting based on two Policy Shaping algorithms: our Continuous Policy Intersection, and Probabilistic Policy Reuse (see Section 2.5). For PPR, we tested two values for ψ, the probability in each timestep to sample the advisor's policy: 0.1 and 0.2. Setting ψ to 0.1 leads to the best results, therefore we only show the results generated by that configuration in Figures 6.1, right. For Continuous Policy Intersection, the size of the vector A^A of actions sampled from the advisor, then submitted to the actor, is set to 4096.

Results are reported in Figure 6.1. Each line is produced by averaging the results of 8 random seeds, with the 95% confidence interval shown as shaded regions. Both when the advisor is trained for 100 episodes and 200 episodes, Policy Intersection outperforms Probabilistic Policy Reuse. In the Ant environment, PPR with a 200 episodes trained advisor shows a good jumpstart until approximately 1500 episodes; our Continuous PI with a 100 episodes trained advisor exceeds PPR's performance after that. Moreover, Self-Transfer with Policy Intersection brings a non-negligible performance gain compared to not using Self-Transfer, even while helped by an advisor

trained during only 100 episodes. This performance gain is present even when the advisor does not have enough training time to learn how to produce rewards above zero, as shown in the bottom plot (for the Hopper environment), and an advisor trained for 100 episodes seems more beneficial than one trained for double that time in that particular environment.

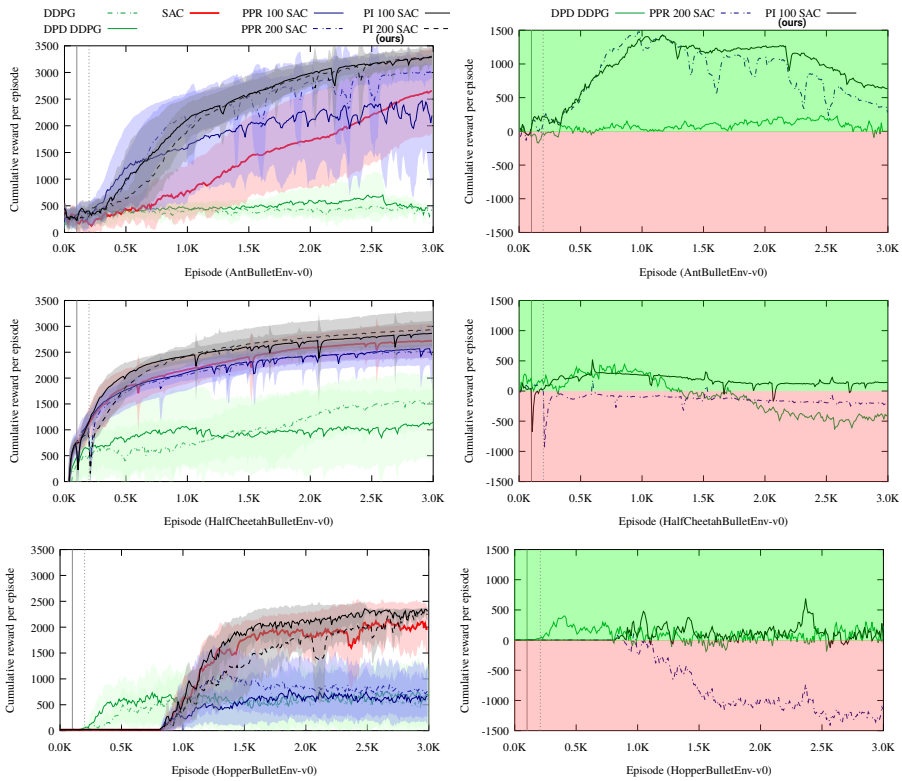

Figure 6.1: *Left*: Comparison between Soft Actor-Critic with no Self-Transfer, and using Self-Transfer with either Probabilistic Policy Reuse (PPR) or Policy Intersection (PI). These curves are averaged over 8 runs per algorithm. The two vertical lines indicate when the current actor is saved as a frozen advisor, and that the current agent is replaced with a fresh agent. In all three PyBullet environments, Self-Transfer with our Continuous Policy Intersection algorithm significantly outperforms its baseline (SAC) and Self-Transfer with Probabilistic Policy Reuse. Results obtained by Dual Policy Distillation on the three PyBullet environments are also shown.

We also report results obtained by Dual Policy Distillation in Figure 6.1, although these are not directly comparable to PI and PPR since they were produced by the implementation provided with the original paper [3].

Figure 6.1: However, DPD's results cannot directly be compared to that of PPR and PI since they are implemented on top of SAC, while DPD is using Deep Deterministic Policy Gradient (DDPG), which seems to learn these tasks less well than SAC. *Right*: Comparison between the performance gain brought by DPD to its baseline DDPG, the gain brought by PPR to SAC and by our continuous PI to SAC in each PyBullet environment. These curves are averaged over 8 runs per algorithm. The performance gain is computed by subtracting the reward obtained by the baseline (in this case, either SAC or DDPG) from the reward obtained by the baseline augmented with the transfer or distillation algorithm for each timestep of the experiment. Hence, we can see the impact of each algorithm over time and how much they concretely improve their baseline. Moreover, this allows for a fair comparison between algorithms implemented on top of different baselines. Out of all three algorithms, our continuous PI is the only one that either consistently significantly improves its baseline, or that does not decrease its baseline performance.

When it comes to the positive difference in performance brought by each individual method to their baseline, our continuous Policy Intersection outperforms both Self-Transfer using PPR, and DPD. Figure 6.1, right, shows for each algorithm the difference between its performance and the performance of its baseline. For instance, the "DPD DDPG" curve is obtained by, for each timestep, subtracting the reward obtained by the original Deep Deterministic Policy Gradient (DDPG) [Lillicrap et al., 2015] algorithm from that obtained when DPD is enabled. Similarly, the curves for PPR and PI are computed by subtracting the reward obtained by SAC alone from that when Self-Transfer is used. We kept the configurations for PPR and PI that led to their best results, namely 200 advisor training episodes for PPR and 100 advisor training episodes for PI. In contrast to learning curves as in Figure 6.1, computing this gain achieved by a given extension of an RL algorithm allows extensions implemented on different baselines to be fairly compared. However, we do not think that this completely excludes the need to compare extensions once they are all implemented on top of the same baseline.

The gain brought by Self-Transfer in general, be it using PPR or our continuous PI is especially visible on the Ant environment, while DPD brings a much smaller gain

[3]This implementation uses Deep Deterministic Policy Gradient (DDPG) [Lillicrap et al., 2015] instead of Soft Actor-Critic; we did not re-implementing DPD on top of SAC as this consists in a non-trivial task considering the complexity of the DPD algorithm.

slightly above zero. However, it seems much harder for all three algorithms to increase the rewards of their baseline on Half-Cheetah and Hopper. In particular, DPD starts off with a positive difference during the first half of the experiment on Half-Cheetah, then worsen DDPG's performance during the other half. PPR very negatively impacts performance in the Hopper environment, while PI and DPD manage to generally remain close to zero if not slightly above. Our continuous PI is the only algorithm leading to a non-negligible improvement of its baseline in both Ant and Half-Cheetah, and manages to keep its impact either null or positive in Hopper.

6.3 Self-Transfer in Discrete Action-Space Environments

We evaluate the use of Self-Transfer in the following scenario: an agent learns to navigate in a large, difficult to explore environment, and is given three slightly pre-trained advisor policies. Each advisor policy has been trained for 40 episodes; the advisee is to be trained for 1000 episodes. One could have simply trained one advisor for 120 episodes, however, as shown below, advisors can vary in quality, in terms of how improved the advisee's policy is thanks to the advisor's help. By training three advisors for shorter amounts of time, rather than training one advisor for longer, the hope is to have a higher chance of getting at least one good advisor.

Note that we do not devise a method to select the best advisor. However, in our limited experience, we noticed that an agent reaching high rewards often makes a good advisor. In addition to trying the solution of using the best advisor from the advisors pool, we also try combining all three advisors, following our contribution in Chapter 5, Section 5.3.1. The environment we evaluate our setting on is Virtual Office, a large, difficult to explore environment with continuous states and discrete actions. In addition, we compare two methods to perform Self-Transfer: Probabilistic Policy Reuse (PPR), and the Actor-Advisor. Both methods are implemented on top of the BDPI reinforcement learning algorithm; we describe this extension of BDPI with PPR in the next section; see Section 3.1.3 in Chapter 3 for that with the Actor-Advisor.

6.3.1 BDPI with Probabilistic Policy Reuse

Our implementation of PPR within our BDPI actor works as follows: at acting time, the action to be executed is sampled from π_{source} with probability ψ, and from the policy learned by the actor $\pi_L(s_t)$ with probability $1 - \psi$. The actor still only learns $\pi_L(s_t)$ and does not have an extra input for the action sampled by π_{source}.

Because BDPI with the Actor-Advisor implements a learning correction, that is beneficial to it (see Section 3.1.3, Chapter 3), we also devise a learning correction

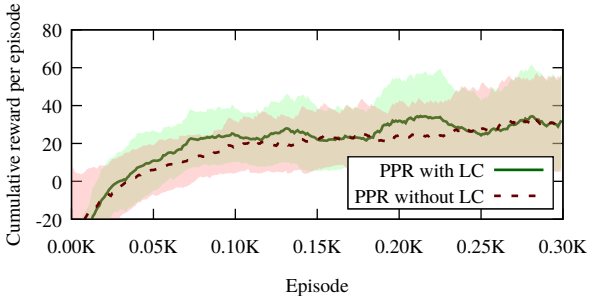

Figure 6.2: Cumulative reward per episode in the Virtual Office using Probabilistic Policy Reuse. The source policy reaches the goal from a single initial position, the target policy reaches the goal from any initial position. The learning correction does not have a measurable impact on PPR ($p = 0.839$ between episodes 250 and 300, each line is the average of 4 runs).

for Probabilistic Policy Reuse. Similarly to the modified actor learning rule in the Actor-Advisor (see Equation 3.4), the learning correction designed to compensate for the use of π-reuse in BDPI must allow the combination of the actor's policy π_L with the advice to converge to an optimal policy:

$$\begin{aligned} \pi(s, \pi_A(s)) &\leftarrow \Gamma(Q(s)) \\ \phi_{reuse} \times \pi_A(s) + (1 - \phi_{reuse}) \times \pi_L(s) &\leftarrow \Gamma(Q(s)) \\ (1 - \phi_{reuse}) \times \pi_L(s) &\leftarrow \Gamma(Q(s)) - \phi_{reuse} \times \pi_A(s) \\ \pi_L(s) &\leftarrow \frac{\Gamma(Q(s)) - \phi_{reuse} \times \pi_A(s)}{(1 - \phi_{reuse})} \end{aligned} \tag{6.1}$$

with $\pi_A(s))$ a single source policy, or a combination of several source policies using Equation 5.1; and $\pi_L(s)$ the agent's currently learned policy. However, in contrast to the Actor-Advisor, using the learning correction in Equation 6.1 together with the PPR algorithm at acting time does not lead to a significant difference in performance ($p = 0.839$ when comparing with and without the learning correction between episodes 250 to 300, in the Virtual Office environment described below). Following the principle of the lowest complexity, we therefore choose not to use any learning correction with PPR in our experiment.

6.3.2 The Virtual Office Environment

Virtual Office is meant to represent an open space cluttered with tables and chairs. We have modeled Virtual Office based on an actual office space situated in the building we work in. In this environment, the agent starts each episode in the same initial spot with a random initial orientation, and must navigate to a goal spot, while avoiding to bump into walls and obstacles. The agent can be in any continuous location in the space, and has 3 actions at its disposal: move forward 0.01 units, rotate about its axis by a 0.1 radians increment to the left, and rotate about its axis by a 0.1 radians increment to the right. The agent does not observe its absolute location in the room, but sees its immediate surroundings using the 60 distance sensors, covering a field of view of 120 degrees, placed on its front. These sensors measure the distance between the agent and the closest obstacle (in the $[0, 1]$ range). The episode ends after 300 time-steps.

If the action executed by the agent would lead to it entering an obstacle, the action is cancelled and a reward of -1 is given. Otherwise, the reward of the agent depends on its distance to the goal region, in light gray in Figure 6.3, and is computed as 100 times the change in distance that the action caused. Because the speed of the agent is 0.01 (0.01 units traversed per time-step), the agent therefore receives a reward of +1 every time-step it moves directly towards the goal region, -1 if it goes in the opposite direction, and values closer to zero when it takes a more tangential path.

Self-Transfer in Virtual Office

We trained three RL agents in Virtual Office during 40 episodes each, then froze them to be used as advisors. Note that it is likely that, out of these three advisors, one may have learned a slightly better policy than the others, and one may have learned a lesser policy. We see two potential ways to leverage these advisors: i) use a single advisor, and the best one, or ii) use a mixture of all three advisors. In case the information of which advisor is the best one is not available, we show in Figure 6.5 below that the Actor-Advisor actually manages to leverage sub-optimal advisors relatively well.

In Figure 6.5, we explore the case in which only one advisor can be used. Simply loading a good advisor and let it resume learning leads to impressive results, and outperforms both the Actor-Advisor and PPR when both are helped by a single advisor. However, when the worst advisor is used, the loading and PPR approaches are unable to override the worst advisor, while only the Actor-Advisor is able to recover. This robustness is crucial in the real world, when producing even a handful of advisors may already be too costly. The Actor-Advisor ensures that *any advisor*, even a sub-optimal one, will still allow the agent in the target task to learn a good policy.

In contrast to the loading approach, PPR and the Actor-Advisor can benefit from our contribution allowing several advisors to be combined when multiple source policies

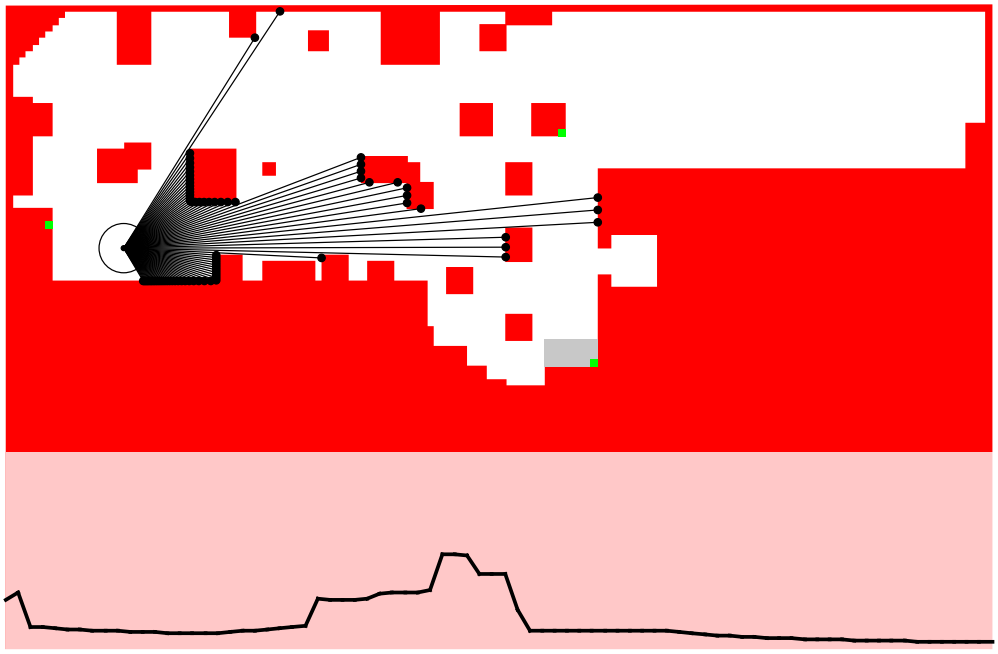

Figure 6.3: *Top:* Our Virtual Office environment representing an open-space cluttered with furniture. The black circle is the agent, and the lines coming from it represent the readings of its 60 frontal distance sensors. The goal region to be reached is in light gray. *Bottom:* Distance readings as observed by the agent (sensor index on X, sensor value on Y, between 0 and 1).

are available, and the designer cannot afford or does not wish to select one over the others. Figure 6.6 shows how the Actor-Advisor, while using combined advisors, can catch up with loading the best advisor in the long run. This means that the Actor-Advisor allows, in the highly-challenging setting where identifying a good advisor is impossible, to learn a policy comparable to what loading the best-possible advisor allows (which would require an oracle in the real world).

Finally, in Figure 6.7, we show that our method of combining advisors, detailed in Section 5.3.1, increases the performance of both PPR and the Actor-Advisor over using only one advisor (we compare the combined advisor to all 3 available advisors in the figure).

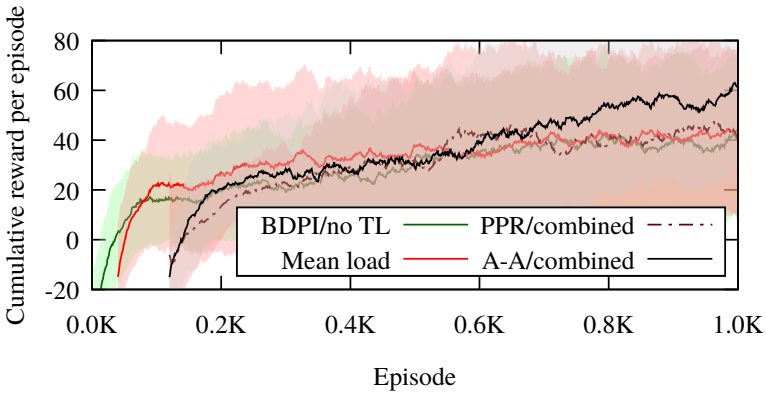

Figure 6.4: Comparison of BDPI without transfer (BDPI/no TL) with the average performance of loading a single advisor (Mean load), Probabilistic Policy Reuse using combined advisors (PPR/combined), and the Actor-Advisor using combined advisors (A-A/combined), on Virtual Office. Each curve takes into account the number of episodes needed to train the advisor(s). Our Actor-Advisor, able to use combined advisors, outperforms all the other approaches in final policy quality. Simply loading an advisor and let it resume learning seems to perform better in the early stages, but the variance of that approach makes it too risky to deploy in the real world.

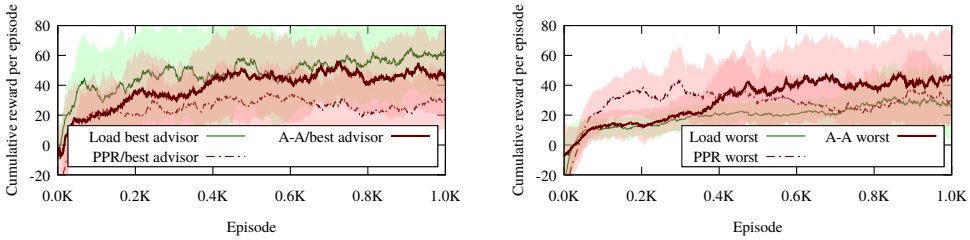

Figure 6.5: Comparison between loading an advisor, PPR and the Actor-Advisor when only one advisor can be used, in the best (*left*) and worst case scenarios (*right*). Results obtained in the Virtual Office when learning to reach the goal from random initial positions. Each curve is the average of at least 4 runs. The Actor-Advisor is the safest method: in the best-case scenario, it is only marginally below simply loading the best advisor on the target task. In the worst-case scenario, the Actor-Advisor largely outperforms the other approaches, and *still allows a high-quality policy to be learned.*

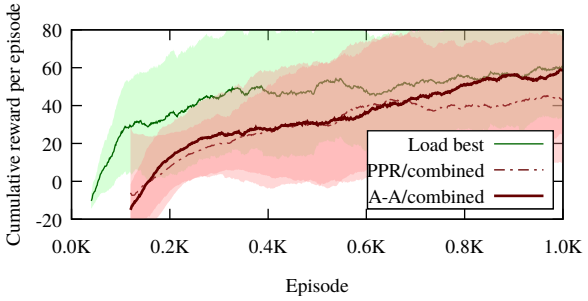

Figure 6.6: Comparison between loading the best-possible advisor (requires an oracle) to the Actor-Advisor and PPR using a combination of 3 advisors (possible in the real-world). The Actor-Advisor is able to match the best-case scenario of the loading approach, while PPR does not.

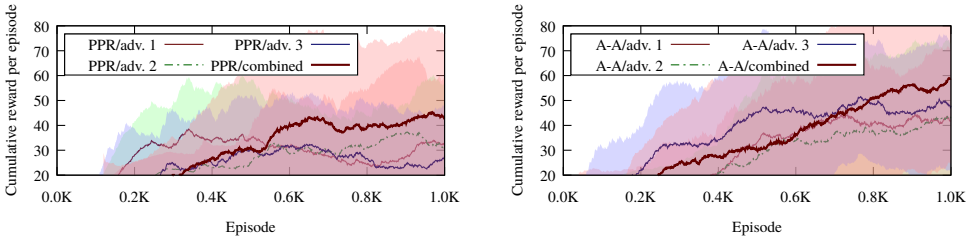

Figure 6.7: Comparison between using our proposed combined advisors, or any individual advisor, both with PPR (*left*) and the Actor-Advisor (*right*). Combined advisors increase the performance of both algorithms, with the Actor-Advisor measurably outperforming PPR with any advisor.

Conclusion on Self-Transfer

In this chapter, we present Self-Transfer Learning, a scheme leveraging conventional Transfer Learning algorithms to allow an RL agent to improve its performance at solving a given task, in both the continuous and the discrete actions settings. Self-Transfer does not require a second instance of the agent learning in parallel, nor a second, separate instance of the environment. The only cost of performing Self-Transfer is an extra training time; we empirically showed in our preliminary experiments that dedicating less than 15 % of the total training time needed to learn the task is enough to gain a significant performance improvement.

As suggested by [Lai et al., 2020], two agents with a different initialization launched in the same environment can gather a distinct experience of that environment, and can improve each other's exploration by exchanging knowledge. Hence, it is likely that the performance gain achieved by our contribution is a result of an improved exploration, as we randomly reset the agent's networks weights after the advisor has been trained. A potentially interesting experiment to verify that hunch could be to have the advisor and the advised agent have the exact same initialization, and compare that to distinct individual initialization. In addition, as future work, we will compare Self-Transfer to Dual Policy Distillation and Self-Imitation on the same reinforcement learning algorithm on environments with continuous actions, and investigate the potential of learning to imitate an agent with a different initialization. Finally, we will compare our current implementation of Policy Intersection for continuous action spaces to the use of the *minimum* operator from fuzzy set theory on the actor's and advisor's possibility distributions.

7 | Conclusion

An important challenge of Reinforcement Learning is sample-efficiency, or how many interactions the agent needs with the environment to learn a good policy. There exist many ways to render learning faster; one can for instance focus on improving the exploration carried out by the agent. Intuitively, making the exploration of an RL agent more effective consists in quickly leading the agent towards areas of the environment that are likely to be fruitful, and prevent it to waste time in areas likely to not be profitable. This can be achieved by directly shaping the policy of the agent, i.e., letting an external advisory policy influence or even determine the action executed during action selecting time. In this thesis, we present our main contribution, the Actor-Advisor, based on Policy Intersection, a Policy Shaping method.

Moreover, as we started applying the Actor-Advisor to numerous problems and settings, we observed that not only a gain in sample-efficiency could be obtained, but also that the Actor-Advisor reveals itself to be convenient for Transfer Learning tasks. Transfer Learning and guiding exploration are related, as one can use the policy learned in a previous task as an external advisory policy for a fresh agent learning a new task. In addition, guiding exploration and Transfer Learning share a similar goal, which is to help the fresh agent learn faster. The only difference lies in the assumption made about the external advisory policy: TL considers that the advisory policy has been produced by another RL agent, while a hand-coded heuristic, or a person can also be used to guide an agent's exploration.

Chapter 3 is dedicated to the extensions of the Actor-Advisor making it applicable to several different RL algorithms and problems. One of the first challenges we encountered was the inherent incompatibility of Policy Gradient methods, a large family of Reinforcement Learning algorithms, with vanilla Policy Intersection, due to their

strong on-policyness. Our first contribution, detailed in Section 3.1.2, is the incorporation of off-policy advice in the Policy Gradient loss to allow the actor to see its actions be influenced by off-policy advice without convergence issue. We empirically show that the learning correction makes Policy Shaping possible for Policy Gradient, in several challenging environments. Moreover, we observe that PPO can benefit from the learning correction as well in difficult to explore environments with sparse rewards, in both the discrete and continuous action cases. We also detail in Section 3.1.3 how the actor update rule of Bootstrapped Dual Policy Iteration, a very different actor than a Policy Gradient one, can be modified to harmoniously incorporate advice. Using the learning correction with a BDPI actor improves the overall performance of the agent in early stages of learning. Finally, we extend vanilla Policy Intersection to environments with continuous actions, thus largely expanding its application scope.

Chapter 4 is focused on evaluating the Actor-Advisor on a learning from human intervention task, in which we compare teaching via rewards and teaching via advice. In contrast to Reward Shaping, the often preferred method to influence the behavior of an agent, Policy Shaping immediately and visibly influences the actions executed by the agent, but does not tamper with the actual objective pursued. On the other hand, Reward Shaping alters the agent's behavior slower and requires a prohibitive number of interventions, but potentially allows for the human user to teach the agent about their preferences. We show that the Actor-Advisor allows good policies to be learned from scarce advice, is robust to errors in the advice, and leads to higher returns than no advice, or reward-based approaches.

In Chapter 5, we apply the Actor-Advisor to four Transfer Learning settings: transfer between two simulated drones which observe the environment through different sensors in Section 5.2; transfer from several advisors to one fresh agent in Section 5.3; transfer from several previous controllers to learn a new one in an air compressor management problem 5.4, and letting several agents learn one task in parallel while simultaneously sharing knowledge with each other in Section 5.5. Our positive results in Section 5.2 show that it is possible to prepare the policy of an agent in a laboratory environment with expensive sensors, and to later deploy it on the field with a cheaper equipment. Our fundamental contribution in Section 5.3 consists in an approach to identify areas where the previous tasks and the new task are similar, and areas in which they are likely to differ, which are crucial information when dealing with Transfer Learning problems. Our positive empirical results in Section 5.4 further motivate us to use RL and Transfer in realistic, close to real-life industrial problems. In Section 5.5, we present a novel web service named Shepherd which allows anyone with an internet connection to train an RL agent, without having to keep their machine up-to-date with the latest software versions required to run RL algorithms. In future work, we wish to explore the possibility for several human users of a web application to train the same policy via Shepherd. This policy would learn to suggest places to visit

in Brussels following some global objectives, such as at what time should a restaurant be suggested to a user, and how far consecutive places should be from one another, but also to suggest new places based on the user's favorite places.

Finally, our thesis ends with the introduction in Chapter 6 of our last contribution, Self-Transfer. Self-Transfer is based on Transfer Learning, and could be considered to exist in that realm, although its setting does not follow the conventional Transfer Learning one. In Self-Transfer, a first agent is trained for a small amount of episodes on a given task, then frozen and saved to serve as an advisor. Then, a fresh agent is launched on the same task as the advisor, and trained while being advised by the advisor until reaching a good policy. If, for instance, the advisor is trained for 100 episodes, and the advisee is trained for 2900 episodes, we observed in Section 6.2 that the performance of the advisee exceeds that of an agent trained without Self-Transfer for 3000 episodes. As a result, Self-Transfer can be considered as a handy learning trick to improve sample-efficiency, which we made applicable to environments with discrete actions and continuous actions.

7.1 Future Research Avenues

We now review a few additional research directions. Some of them are new application opportunities for the Actor-Advisor, such as continuous transfer learning for life-long learning agents, agents with different RL algorithms learning in parallel while advising each other, and learning from human biometrics. Others take a broader look at the field of RL.

Throughout our PhD study, we often felt the need to develop our own Reinforcement Learning environments, due to an insufficient amount of existing benchmarks for RL algorithms. Developing our own environments made us realize the lack of documentation, standard practices and investigation done in the domain of RL environment development. In the near future, we wish to promote RL environment design as a RL research sub-domain of its own, generative of valuable contributions, and not merely as a tool to evaluate researchers' new RL algorithms.

We also mention research avenues that are much further from our comfort zone, and out of the scope of this thesis, namely Reinforcement Learning used for Generative Art. More and more artists are already opting for Supervised Learning methods to create art pieces. To leverage RL to generate art, one defines the creative scope and framework of the agent through the design of the RL environment. As a result, using RL in the context of generating art requires more involvement from the artist than when using Supervised Learning techniques, since designing an RL environment consists in a more challenging task than passing a set of images to a classifier. Note that this last topic points back to our environment design research direction.

7.1.1 Life-Long Transfer

In the pursuit of life-long learning [Ring, 1998; Khetarpal et al., 2018], future RL algorithms will probably need to include some automatic transfer mechanism to make learning more efficient and already acquired skills easily reusable. We believe that the Actor-Advisor has the potential to make that incorporation of systematic transfer from older policies to newer policies easy to implement. An automatic transfer scheme could be devised as follows: every x amount of episodes, the current policy is frozen and saved in an advisor pool, and a fresh policy is initialized. This fresh policy resumes learning for the next x episodes, while being advised by either a combination of advisors from the pool of advisors, one randomly picked advisor from the pool, or an advisor selected based on its relevance according to the current situation of the agent. Such scheme could work well in ever-changing environments in which the agent cannot afford to get too confident and must regularly keep on exploring, while being able to exploit relevant already acquired knowledge.

7.1.2 Policy Distillation Between Different RL algorithms

Similarly to the framework proposed by Lai et al. [2020], the Actor-Advisor can be used to let two agents learn a task in parallel while advising each other at the same time. The Actor-Advisor does not make any assumption on the underlying RL algorithm used by each agent, as long as they can produce state-dependent policy vectors (or a probability given an action in the continuous actions case). As a result, one agent could run a widely different RL algorithm than the other. This could help the merging of originally incompatible RL techniques that can complement and compensate each other's limitations.

7.1.3 Extracting Advice From Biometrics

In Chapter 4, we briefly mentioned the need for AI assistive systems to make users feel at ease around them. To achieve this, some biometric sensors could be used, such as a pulse sensor, to let the RL agent observe how stressed the user is at each time-step. Although this information is fairly straightforward to include as an additional state information, or even as a reward, it is less clear however how this stream of data can be interpreted as an advisory policy. Nevertheless, we believe desirable for a user to have the possibility to give directions to its assistive robot without having to utter a word. By measuring the user's stress levels, the agent could be commanded to stop whatever it is currently doing because its actions are stress inducing, for instance. As we have discussed in Chapter 4, the advantage of using advice over rewards or feedback is that the behavior of the agent is directly and visibly impacted, instead of slowly changed over time.

In the next section, we move on to a completely different research avenue that we did not mention in this thesis, and that has nothing to do with the Actor-Advisor, but that we wish to explore in the future.

7.1.4 RL Environments

Not only does most of the research in Reinforcement Learning restricts itself to idealized, simulated tasks running on computers, rather than real-life, robotic tasks (except for a few notable exceptions [Levine et al., 2016]) but also little research is done on the design of environments used as RL benchmarks in general. Therefore, when it comes to choosing on which environment one would like to evaluate their RL algorithm, the options can feel limited. There exist only a few collections of environments, among which the best-known are OpenAI Gym [Brockman et al., 2016], the proprietary physics engine Mujoco [Todorov et al., 2012], the 3D navigation simulator AirSim [Shah et al., 2018], and the continuous control benchmark RL-Lab [Duan et al., 2016]. One could argue that keeping the amount of available environments small, hence forcing everyone to evaluate their method on the same benchmarks is desirable to better compare algorithms against each other; that is the point of a common benchmark, after all. Nevertheless, this can also lead to an overfitting of RL algorithms to a specific type of problem, if the benchmarks available do not represent a large enough range of different applications. If the research community as a whole prefers evaluating their contributions to video game-like environments, for instance, we might end up with RL methods very proficient at playing video games, and little else.

It is likely that, as RL slowly comes out of the labs and in industry-oriented fields, more researchers will be forced to design their own environment, accurately representing a very specific problem to be solved. However, as mentioned above, environment design is still a secondary research interest, and the difficulty of a given environment is even less investigated.

Environment Design

Environment design consists in formulating a problem, potentially existing in real-life, as a program interacting with the RL agent. In this thesis, we designed a few environments ourselves to evaluate our contributions on: Five Rooms, the large grid-world used in Chapter 4 and 5, section 5.3, Bored in the City, the web application used in Section 5.5, and Virtual Office, the large office space with continuous states and discrete actions in Chapter 6. The main choices to be made when designing an environment are the following:

1) What information should the agent be able to observe in order to learn the task, and how should the information be encoded?

2) Which actions should be available to the agent?

3) How should the agent be rewarded so that the it learns a desirable behavior?

In our experience, points 1 and 3 are often the trickiest, as they can make or break the ability of an RL agent to learn the task. For a reinforcement learner to have a chance at learning the task, the problem formulated must have the Markov property [Sutton and Barto, 2018, p. 57]. At time-step t, the state s_t is the only source of information the agent has on its environment; it must contain all relevant information for the agent to be able to learn the task. If some relevant information about the current state of the environment is not present in s_t, the environment becomes a Partially Observable Markov Decision Process, in contrast to a fully observable one; solving POMDPs is a challenging research field of its own. Once one has determined what information should be included in the state, there remains the question of how should this information be encoded. The same information can sometimes be expressed in many, more or less compact ways. A verbose state representation can be easier to learn from, but result in larger state spaces that take more time to explore; a highly compact state representation reduces the size of the state space, but the neural network representing the policy or Q function may have a more difficult time making sense of it.

The reward function defines the goal of the agent; tampering with the reward function means changing the objective of the task. Extreme care must be put in the design of the reward function to prevent the agent from reward *hacking*, i.e., exploiting a loophole allowing it to gain rewards without having to actually carry out the task [Amodei et al., 2016]. The safest approach is to make sure that the reward is accurately leading the agent towards the desired goal, and that any step taking the agent further away from that goal gets punished; such reward function is said to be potential-based [Ng et al., 1999]. As we briefly discussed in Chapter 5, Section 5.2, adding an extra shaping reward that is also based on a potential function is possible, thanks to the work presented in Harutyunyan et al. [2015b]. The problem with that method is that it requires an additional agent to learn a Q function on which the shaping reward can then be based, which is not always an option; many designer tend to design both their environmental reward function, and their shaping reward if needed, based on gut feeling.

To summarize, although there exist many contributions in terms of new RL algorithms (and this thesis is no exception), not much formal work has yet been presented on the design of RL environments themselves. Nevertheless, we feel that environments will soon transition from mere benchmarks to representations of actual specific real-life problems, and mastering the design of such environment will become an important skill for RL experts. To our knowledge, for now, people use their intuition and rules of thumb, which can only be acquired through practice, to create new environments.

Such tips and tricks are seldom reported in papers. We wish to promote the existence of designing RL environment as a full-fledged RL research sub-field of its own.

Environment Difficulty

Related to environment design, little is formally and exhaustively investigated about what makes an RL environment difficult to learn for an agent, and how can the currently most used environments be graded in terms of difficulty. It is generally accepted by the community that a sparse reward signal is harder to learner from than a rich, frequent reward signal, as it requires the agent to thoroughly explore the environment while receiving little reward in return. Some environment, which consist in games such as chess, may reward the agent only at the very end of the game. This typically gives rise to a *credit assignment* problem, i.e., the agent has a hard time figuring out which actions were decisive for it to get the reward, and which were not.

Moreover, to go back to point 1) mentioned above, and as we briefly discussed in Chapter 5, the content of the state, as well as its representation are important factors when it comes to the feasibility of learning the task. In Chapter 5, Section 5.2, we hint at the possibility that some sensors might help the agent learn better than others, such as using distance sensors over a webcam. This may be because RL algorithms become more and more limited by what their underlying neural network can learn. Learning from raw images is generally difficult for a neural network without any prior tweaking of the state representation, for instance.

In the future, we hope to see a more systematic discussion in papers of why a given environment chosen to evaluate a new algorithm is considered to be challenging and in what specific ways; this can notably help the reader better understand the strengths and weaknesses of the proposed RL algorithm. We also wish to investigate the design of a difficulty metric for environments.

RL and Art

This last direction is probably the one we are the least familiar with, which is to delve into the field of Generative Art [Galanter, 2016]. Specially, concrete work has already been carried out to show how Reinforcement Learning can be used to create art pieces [Luo, 2020]. When Machine Learning is used in an artistic endeavor, artists tend to prefer Supervised Learning methods, perhaps because the use of Supervised Learning is more intuitive to create graphic art. However, when using SL, the input of the artist in the creative process is limited, and often merely consists in feeding input images to the neural network; the network outputs the final art piece.

In RL, the contribution required from the artist is potentially much more important: they would have to design an RL environment in which the agent learns the steps to make an art piece. For example, the agent could have to show a sequence of

pictures in an order that would provoke an emotion, such as surprise, in the human visitor. Through the environment, the artist provides the pictures, describes that actions consists in showing a given picture, or a combination of several; they can add sounds to be played by the agent to accompany the images. As the current state, the agent could observe what images and sounds it is currently displaying, and the facial expression of the visitor using a webcam. Another example of RL used to create art is music generation with a human in the loop: the agent can progressively add notes to a track, while a human rewards the agent according to how good the current track sounds to them. In contrast to using Supervised Learning techniques, creating complete RL environments to serve as creative playgrounds to RL agents generally consists in a greater effort and requires a good grasp of the field; this further motivates us to promote environment design as a field of applications and research of its own.

Bibliography

Abu-Mostafa, Y. S., M. Magdon-Ismail, and H.-T. Lin
2012. *Learning from data*, volume 4. AMLBook New York.

Achiam, J. and S. Sastry
2017. Surprise-based intrinsic motivation for deep reinforcement learning. *arXiv preprint arXiv:1703.01732*.

Alshiekh, M., R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu
2018. Safe reinforcement learning via shielding. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Amodei, D., C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané
2016. Concrete problems in AI safety. *Arxiv pre-print*.

Andre, D. and S. J. Russell
2002. State abstraction for programmable reinforcement learning agents. In *AAAI/IAAI*, Pp. 119–125.

Antos, A., C. Szepesvári, and R. Munos
2007. Fitted q-iteration in continuous action-space mdps. *Advances in neural information processing systems*, 20.

Bellman, R.
1957. A Markovian decision process. *Journal Of Mathematics And Mechanics*, 6:679–684.

Brockman, G., V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba
2016. Openai gym. *arXiv preprint arXiv:1606.01540*.

Brys, T.
2016. *Reinforcement Learning with Heuristic Information*. PhD thesis, PhD thesis, Vrije Universitet Brussel.

Brys, T., A. Harutyunyan, H. B. Suay, S. Chernova, M. E. Taylor, and A. Nowé
2015. Reinforcement learning from demonstration through shaping. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, Pp. 3352–3358. AAAI Press.

Bucila, C., R. Caruana, and A. Niculescu-Mizil
2006. Model compression: Making big, slow models practical. In *International Conference on Knowledge Discovery and Data Mining*.

Burda, Y., H. Edwards, D. Pathak, A. Storkey, T. Darrell, and A. A. Efros
2018. Large-scale study of curiosity-driven learning. *arXiv preprint arXiv:1808.04355*.

Cederborg, T., I. Grover, C. L. Isbell, and A. L. Thomaz
2015. Policy shaping with human teachers. In *IJCAI*.

Chaplot, D. S., G. Lample, K. M. Sathyendra, and R. Salakhutdinov
2016. Transfer deep reinforcement learning in 3d environments: An empirical study. In *NIPS Deep Reinforcemente Leaning Workshop*.

Christiano, P. F., J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei
2017. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds., Pp. 4299–4307. Curran Associates, Inc.

Chua, K., R. Calandra, R. McAllister, and S. Levine
2018. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. *Advances in neural information processing systems*, 31.

Coumans, E. and Y. Bai
2016–2019. Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org.

da Silva, F. L., R. Glatt, and A. H. R. Costa
2017. Simultaneously learning and advising in multiagent reinforcement learning. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '17, P. 1100–1108, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

De Troyer, O., J. Maushagen, R. Lindberg, J. Muls, B. Signer, and K. Lombaerts
2019. A playful mobile digital environment to tackle school burnout using micro learning, persuasion & gamification. In *2019 IEEE 19th International Conference on Advanced Learning Technologies (ICALT)*, volume 2161, Pp. 81–83. IEEE.

Devin, C., A. Gupta, T. Darrell, P. Abbeel, and S. Levine
2017. Learning modular neural network policies for multi-task and multi-robot transfer. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, Pp. 2169–2176. IEEE.

Duan, Y., X. Chen, R. Houthooft, J. Schulman, and P. Abbeel
2016. Benchmarking deep reinforcement learning for continuous control. In *International conference on machine learning*, Pp. 1329–1338. PMLR.

Engel, Y., S. Mannor, and R. Meir
2005. Reinforcement learning with gaussian processes. In *Proceedings of the 22nd international conference on Machine learning*, Pp. 201–208.

Fernández, F. and M. M. Veloso
2006. Probabilistic policy reuse in a reinforcement learning agent. In *International Conference on Autonomous Agents and Multiagent Systems*.

Forcier, J., P. Bissex, and W. J. Chun
2008. *Python web development with Django*. Addison-Wesley Professional.

Frank, M., J. Leitner, M. Stollenga, A. Förster, and J. Schmidhuber
2014. Curiosity driven reinforcement learning for motion planning on humanoids. *Frontiers in neurorobotics*, 7:25.

Fujimoto, S., H. van Hoof, and D. Meger
2018. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*.

Galanter, P.
2016. Generative art theory. *A Companion to Digital Art*, 1:631.

García, J. and F. Fernández
2015. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*.

García, J. and F. Fernández
2019. Probabilistic policy reuse for safe reinforcement learning. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 13(3):1–24.

Garivier, A. and E. Moulines
2011. On upper-confidence bound policies for switching bandit problems. In *International Conference on Algorithmic Learning Theory*, Pp. 174–188. Springer.

Gaweda, A. E., M. K. Muezzinoglu, G. R. Aronoff, A. A. Jacobs, J. M. Zurada, and M. E. Brier
2005. Individualization of pharmacological anemia management using reinforcement learning. *Neural Networks*, 18(5-6):826–834.

Griffith, S., K. Subramanian, J. Scholz, C. L. Isbell, and A. L. Thomaz
2013. Policy shaping: Integrating human feedback with reinforcement learning. In *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, eds., Pp. 2625–2633. Curran Associates, Inc.

Haarnoja, T., A. Zhou, P. Abbeel, and S. Levine
2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv*, abs/1801.01290.

Hadfield-Menell, D., S. J. Russell, P. Abbeel, and A. Dragan
2016. Cooperative inverse reinforcement learning. In *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, eds., volume 29, Pp. 3909–3917. Curran Associates, Inc.

Harrison, B., U. Ehsan, and M. O. Riedl
2017. Guiding reinforcement learning exploration using natural language. *CoRR*, abs/1707.08616.

Harutyunyan, A., T. Brys, P. Vrancx, and A. Nowé
2014. Off-policy shaping ensembles in reinforcement learning. *arXiv preprint arXiv:1405.5358*.

Harutyunyan, A., T. Brys, P. Vrancx, and A. Nowé
2015a. Shaping mario with human advice. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '15, Pp. 1913–1914, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Harutyunyan, A., S. Devlin, P. Vrancx, and A. Nowé
2015b. Expressing arbitrary reward functions as potential-based advice. In *Association for the Advancement of Artificial Intelligence*.

Hessel, M., J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver
2018. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*.

Hussein, A., M. M. Gaber, E. Elyan, and C. Jayne
2017. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):1–35.

Jr., C. L. I., C. R. Shelton, M. J. Kearns, S. P. Singh, and P. Stone
2001. Cobot: A social reinforcement learning agent. In *Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, December 3-8, 2001, Vancouver, British Columbia, Canada]*, Pp. 1393–1400.

Kaochar, T., R. T. Peralta, C. T. Morrison, I. R. Fasel, T. J. Walsh, and P. R. Cohen
2011. Towards understanding how humans teach robots. In *User Modeling, Adaption and Personalization*, J. A. Konstan, R. Conejo, J. L. Marzo, and N. Oliver, eds., Pp. 347–352, Berlin, Heidelberg. Springer Berlin Heidelberg.

Khetarpal, K., S. Sodhani, S. Chandar, and D. Precup
2018. Environments for lifelong reinforcement learning. *arXiv preprint arXiv:1811.10732*.

Kim, E. S., D. Leyzberg, K. M. Tsui, and B. Scassellati
2009. How people talk when teaching a robot. In *Proceedings of the 4th ACM/IEEE International Conference on Human Robot Interaction*, HRI '09, Pp. 23–30, New York, NY, USA. ACM.

Knox, W. B., B. D. Glass, B. C. Love, W. T. Maddox, and P. Stone
2012. How humans teach agents - A new experimental perspective. *I. J. Social Robotics*, 4(4):409–421.

Knox, W. B. and P. Stone
2009. Interactively shaping agents via human reinforcement: The tamer framework. In *Proceedings of the fifth international conference on Knowledge capture*, Pp. 9–16. ACM.

Knox, W. B. and P. Stone
2010. Combining manual feedback with subsequent MDP reward signals for reinforcement learning. In *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*.

Konidaris, G. and A. G. Barto
2007. Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, volume 7, Pp. 895–900.

Kuchling, A. and M. Zadka
2012. What's new in python 2.5. *Python Software Foundation. Retrieved*, 11.

Kuhnle, A., N. Röhrig, and G. Lanza
2019. Autonomous order dispatching in the semiconductor industry using reinforcement learning. *Procedia CIRP*, 79:391–396. 12th CIRP Conference on Intelligent Computation in Manufacturing Engineering, 18-20 July 2018, Gulf of Naples, Italy.

Kurutach, T., I. Clavera, Y. Duan, A. Tamar, and P. Abbeel
2018. Model-ensemble trust-region policy optimization. *arXiv preprint arXiv:1802.10592*.

Lai, K.-H., D. Zha, Y. Li, and X. Hu
2020. Dual policy distillation. *arXiv preprint arXiv:2006.04061*.

Lakshminarayanan, B., A. Pritzel, and C. Blundell
2017. Simple and scalable predictive uncertainty estimation using deep ensembles. *Advances in neural information processing systems*, 30.

Lazic, N., C. Boutilier, T. Lu, E. Wong, B. Roy, M. Ryu, and G. Imwalle
2018. Data center cooling using model-predictive control. *Advances in Neural Information Processing Systems*, 31.

Levine, S., C. Finn, T. Darrell, and P. Abbeel
2016. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*.

Li, G., R. Gomez, K. Nakamura, and B. He
2019. Human-centered reinforcement learning: A survey. *IEEE Transactions on Human-Machine Systems*, 49(4):337–349.

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

Loftin, R. T., B. Peng, J. MacGlashan, M. L. Littman, M. E. Taylor, J. Huang, and D. L. Roberts
2014. Learning something from nothing: Leveraging implicit human feedback strategies. In *The 23rd IEEE International Symposium on Robot and Human Interac-*

tive Communication, IEEE RO-MAN 2014, Edinburgh, UK, August 25-29, 2014, Pp. 607–612.

Luo, J.
2020. *Reinforcement learning for generative art.* University of California, Santa Barbara.

MacGlashan, J., M. K. Ho, R. Loftin, B. Peng, G. Wang, D. L. Roberts, M. E. Taylor, and M. L. Littman
2017. Interactive learning from policy-dependent human feedback. In *Proceedings of the 34th International Conference on Machine Learning*, D. Precup and Y. W. Teh, eds., volume 70 of *Proceedings of Machine Learning Research*, Pp. 2285–2294, International Convention Centre, Sydney, Australia. PMLR.

Madden, M. G. and T. Howley
2004. Transfer of experience between reinforcement learning environments with progressive difficulty. *Artificial Intelligence Review*, 21(3-4):375–398.

Mathewson, K. W. and P. M. Pilarski
2017. Actor-critic reinforcement learning with simultaneous human control and feedback. *CoRR*, abs/1703.01274.

Mirowski, P., M. Grimes, M. Malinowski, K. M. Hermann, K. Anderson, D. Teplyashin, K. Simonyan, A. Zisserman, R. Hadsell, et al.
2018. Learning to navigate in cities without a map. In *Advances in Neural Information Processing Systems*, Pp. 2419–2430.

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

Moerland, T. M., J. Broekens, and C. M. Jonker
2020. Model-based reinforcement learning: A survey. *arXiv preprint arXiv:2006.16712*.

Nair, A., B. McGrew, M. Andrychowicz, W. Zaremba, and P. Abbeel
2017. Overcoming exploration in reinforcement learning with demonstrations. *CoRR*, abs/1709.10089.

Najar, A. and M. Chetouani
2021. Reinforcement learning with human advice: a survey. *Frontiers in Robotics and AI*, 8.

Ng, A. Y., D. Harada, and S. J. Russell
1999. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning (ICML 1999), Bled, Slovenia, June 27 - 30, 1999*, Pp. 278–287.

Oh, J., Y. Guo, S. Singh, and H. Lee
2018. Self-imitation learning. *arXiv preprint arXiv:1806.05635*.

Omidshafiei, S., D.-K. Kim, M. Liu, G. Tesauro, M. Riemer, C. Amato, M. Campbell, and J. P. How
2019. Learning to teach in cooperative multiagent reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, Pp. 6128–6136.

Oudeyer, P.-Y. and F. Kaplan
2009. What is intrinsic motivation? a typology of computational approaches. *Frontiers in neurorobotics*, P. 6.

Parisotto, E., J. L. Ba, and R. Salakhutdinov
2015. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*.

Pathak, D., P. Agrawal, A. A. Efros, and T. Darrell
2017. Curiosity-driven exploration by self-supervised prediction. In *International conference on machine learning*, Pp. 2778–2787. PMLR.

Peters, J. and J. A. Bagnell
2010. Policy gradient methods. *Scholarpedia*, 5(11):3698.

Peters, J. and S. Schaal
2008. Reinforcement learning of motor skills with policy gradients. *Neural Networks*, 21(4):682–697.

Pirotta, M., M. Restelli, A. Pecorino, and D. Calandriello
2013. Safe policy iteration. In *International Conference on Machine Learning*, Pp. 307–315.

Plisnier, H., D. Steckelmacher, D. M. Roijers, and A. Nowé
2019a. The actor-advisor: Policy gradient with off-policy advice. *arXiv*, abs/1902.02556.

Plisnier, H., D. Steckelmacher, D. M. Roijers, and A. Nowé
2019b. Transfer reinforcement learning across environment dynamics with multiple advisors. In *BNAIC/BENELEARN*.

Precup, D., R. S. Sutton, and S. P. Singh
1998. Theoretical results on reinforcement learning with temporally abstract options. In *10th European Conference on Machine Learning (ECML)*, Pp. 382–393.

Raffin, A., A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann
2019. Stable baselines3.

Ravindran, B. and A. G. Barto
2003. Relativized options: Choosing the right transformation. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, Pp. 608–615.

Ring, M. B.
1998. Child: A first step towards continual learning. In *Learning to learn*, Pp. 261–292. Springer.

Russo, D. J., B. Van Roy, A. Kazerouni, I. Osband, Z. Wen, et al.
2018. A tutorial on thompson sampling. *Foundations and Trends® in Machine Learning*, 11(1):1–96.

Rusu, A. A., S. G. Colmenarejo, C. Gulcehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell
2015. Policy distillation. *arXiv preprint arXiv:1511.06295*.

Sahni, H., B. Harrison, K. Subramanian, T. Cederborg, C. Isbell, and A. Thomaz
2016. Policy shaping in domains with multiple optimal policies: (extended abstract). In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, AAMAS '16, Pp. 1455–1456, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Schulman, J., P. Moritz, S. Levine, M. Jordan, and P. Abbeel
2015. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

Schulman, J., F. Wolski, P. Dhariwal, A. Radford, and O. Klimov
2017. Proximal policy optimization algorithms. *Arxiv pre-print*.

Shah, S., D. Dey, C. Lovett, and A. Kapoor
2018. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*, Pp. 621–635. Springer.

Steckelmacher, D.
2020. *Model-Free Reinforcement Learning for Real-World Robots*. PhD thesis, University of Applied Sciences Utrecht.

Steckelmacher, D., H. Plisnier, D. M. Roijers, and A. Nowé
2019. Sample-Efficient Model-Free Reinforcement Learning with Off-Policy Critics. *arXiv e-prints*, P. arXiv:1903.04193.

Steckelmacher, D., D. M. Roijers, A. Harutyunyan, P. Vrancx, H. Plisnier, and A. Nowé
2018. Reinforcement learning in POMDPs with memoryless options and option-observation initiation sets. In *Proceedings of the AAAI 2018 Conference on Artificial Intelligence*.

Still, S. and D. Precup
2012. An information-theoretic approach to curiosity-driven reinforcement learning. *Theory in Biosciences*, 131(3):139–148.

Sutton, R., D. McAllester, S. Singh, and Y. Mansour
2000. Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Neural Information Processing Systems (NIPS)*, P. 7.

Sutton, R. S.
1990. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine learning proceedings 1990*, Pp. 216–224. Elsevier.

Sutton, R. S. and A. G. Barto
2018. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge.

Sutton, R. S., D. A. McAllester, S. P. Singh, Y. Mansour, et al.
1999a. Policy gradient methods for reinforcement learning with function approximation. In *Neural Information Processing Systems (NeurIPS)*, volume 99, Pp. 1057–1063.

Sutton, R. S., D. Precup, and S. Singh
1999b. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artif. Intell.*, 112(1-2):181–211.

Tang, H. and T. Haarnoja
2017. Learning diverse skills via maximum entropy deep reinforcement learning. *URL http://bair. berkeley. edu/blog/2017/10/06/soft-q-learning*.

Tanner, B. and A. White
2009. Rl-glue: Language-independent software for reinforcement-learning experiments. *The Journal of Machine Learning Research*, 10:2133–2136.

Taylor, M. E. and P. Stone
2007. Cross-domain transfer for reinforcement learning. In *Proceedings of the 24th international conference on Machine learning*, Pp. 879–886. ACM.

Taylor, M. E. and P. Stone
2009. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*.

Taylor, M. E., P. Stone, and Y. Liu
2007. Transfer learning via inter-task mappings for temporal difference learning. *Journal of Machine Learning Research*, 8(Sep):2125–2167.

Taylor, M. E., H. B. Suay, and S. Chernova
2011. Integrating reinforcement learning with human demonstrations of varying ability. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '11, Pp. 617–624, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

Thomaz, A. L. and C. Breazeal
2006. Reinforcement learning with human teachers: Evidence of feedback and guidance with implications for learning performance. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, Pp. 1000–1005. AAAI Press.

Thompson, W. R.
1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3-4):285–294.

Todorov, E., T. Erez, and Y. Tassa
2012. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, Pp. 5026–5033. IEEE.

Vien, N. A. and W. Ertel
2012. Learning via human feedback in continuous state and action spaces. In *Robots Learning Interactively from Human Teachers, Papers from the 2012 AAAI Fall Symposium, Arlington, Virginia, USA, November 2-4, 2012*.

Warnell, G., N. R. Waytowich, V. Lawhern, and P. Stone
2017. Deep TAMER: interactive agent shaping in high-dimensional state spaces. abs/1709.10163.

Watkins, C. J. and P. Dayan
1992. Q-learning. *Machine learning*, 8(3):279–292.

Wexler, B., E. Sarafian, and S. Kraus
2022. Analyzing and overcoming degradation in warm-start reinforcement learning.

Williams, R. J.
1992. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8(3):229–256.

Zhang, A., H. Satija, and J. Pineau
2018. Decoupling dynamics and reward for transfer learning. *arXiv preprint arXiv:1804.10689*.

Zhu, Z., K. Lin, and J. Zhou
2020. Transfer learning in deep reinforcement learning: A survey.