POLITECNICO
MILANO 1863

POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAM IN INFORMATION TECHNOLOGY

BUILDING STRUCTURED HIERARCHICAL AGENTS

Doctoral Dissertation of:
Davide Tateo

Supervisor:
Prof. Andrea Bonarini
Co-supervisors:
Prof. Marcello Restelli

Tutor:
Prof. Francesco Amigoni

Chair of the Doctoral Program:
Prof. Barbara Pernici

2019 – Cycle XXX

Abstract

There is an increasing interest in Reinforcement Learning to solve new and more challenging problems. We are now able to solve moderately complex environments thanks to the advances in Policy Search methods and Deep Reinforcement Learning, even when using low-level data representations as images or raw sensor inputs. These advances have widened the set of application contexts in which machine learning techniques can be applied, bringing in the near future the application of these techniques in other emerging fields of research, such as robotics and unmanned autonomous vehicles. In these applications, autonomous agents are required to solve very complex tasks, using information taken from low-level sensors, in uncontrolled, dangerous, and unknown scenarios.

However, many of these new methods suffer from major drawbacks: lack of theoretical results, even when based on sound theoretical frameworks, lack of interpretability of the learned behavior, instability of the learning process, domain knowledge not exploited systematically, extremely data hungry algorithms.

The objective of this thesis is to address some of these problems and provide a set of tools to simplify the design of Reinforcement Learning agents, particularly when it comes to robotic systems that share some common characteristics. Most of these systems use continuous state and action variables that may need a fine-grained precision, making a good variety of deep learning approaches ineffective. They may exhibit different dynamics between different parts of the system, leading to a natural division based on different time scales, variable magnitudes, and abstraction levels. Finally, some of them are even difficult to formalize as a Reinforcement Learning task, making it difficult to define a reward function, while some human (or non-human) experts may be able to provide behavioral demonstrations.

Based on these assumptions, we propose two approaches to improve the applicability of Reinforcement Learning techniques in these scenarios: hierarchical approaches to Reinforcement Learning, to exploit the structure of the problem, and Inverse Reinforcement Learning, which are a set of techniques able to extract the reward function i.e., the representation of the objective pursued by the agent, and the desired behavior from a set of experts' demonstrations.

From these ideas follow the two major contributions of this work: a new Hierarchical Reinforcement Learning framework based on the Control Theory framework, which is particularly well-suited for robotic systems, and a family of Inverse Reinforcement Learning algorithms that are able to learn a suitable reward function for tasks (or subtasks) difficult to formalize as a reward function, particularly when demonstrations come from a set of different suboptimal experts. Our proposals make it possible to easily design a complex hierarchical control structure and learn the policy either by interacting directly with the environment or providing demonstrations for some subtasks or for the whole system.

Sommario

C'è un crescente interesse nel settore dell'Apprendimento per Rinforzo per la risoluzione di problemi nuovi e impegnativi. Siamo ora in grado di risolvere problemi moderatamente complessi, grazie ai nuovi progressi nei metodi di ricerca nello spazio delle politiche e di Apprendimento per Rinforzo Profondo, anche quando vengono utilizzati dati di basso livello, quali immagini o letture dirette dai sensori. Questi progressi hanno ampliato l'insieme dei contesti applicativi nei quali è possibile applicare le tecniche di apprendimento automatico, portando nel prossimo futuro all'applicazione di queste tecniche in altri contesti emergenti, quali la robotica e i veicoli autonomi senza conducente. In queste applicazioni, è richiesto che gli agenti autonomi risolvano problemi molto complessi, usando le informazioni di basso livello provenienti dai sensori, in contesti non controllati, sconosciuti e pericolosi.

Tuttavia, la maggior parte di questi metodi hanno alcuni lati negativi: mancano di proprietà teoriche, anche quando basati su basi teoretiche solide, mancano di interpretabilità del comportamento appreso, hanno un apprendimento instabile, non viene sfruttata sistematicamente la conoscenza di dominio, richiedono una mole considerevole di dati.

L'obbiettivo di questa tesi è affrontare alcuni di questi problemi e fornire un insieme di strumenti per rendere semplice la progettazione di agenti per l'Apprendimento per Rinforzo in particolare quando si ha a che fare con sistemi robotici, che hanno alcune caratteristiche comuni. La maggior parte di questi sistemi usano spazi di stato e azione continui che potrebbero aver bisogno di una precisione fine, rendendo inefficaci una buona parte degli approcci basati sull'Apprendimento per Rinforzo Profondo. Generalmente, esibiscono dinamiche differenti tra diverse parti del sistema, portando a una naturale suddivisione basata su scale temporali, ampiezze dei segnali e astrazioni differenti. Infine, alcuni di questi problemi sono difficili da formalizzare come problemi di Apprendimento per Rinforzo, poichè è difficile definire la funzione di rinforzo, mentre alcuni esperti (umani o non)

potrebbero fornire dimostrazioni sub-ottime.

Basandoci su queste assunzioni, proponiamo due approcci per rendere maggiormente applicabili le tecniche di Apprendimento per Rinforzo in questi scenari: approcci gerarchici all'Apprendimento per Rinforzo, per sfruttare la struttura del problema, e l'Apprendimento per Rinforzo Inverso, che è un insieme di tecniche per estrarre la funzione di rinforzo, che è la rappresentazione dell'obbiettivo che l'agente sta perseguendo, e il comportamento desiderato dalle dimostrazioni degli esperti.

Da queste idee nascono i due maggiori contributi di questo lavoro: un nuovo framework per l'Apprendimento per Rinforzo gerarchico basato sulla teoria del controllo, che è particolarmente adatto ai sistemi robotici, e una famiglia di algoritmi di Apprendimento per Rinforzo Inverso che sono in grado di imparare una funzione di rinforzo adeguata per obbiettivi (o sotto-obbiettivi) che sono difficili da formalizzare in termini di funzione di rinforzo, soprattutto nel caso in cui le dimostrazioni provengano da un gruppo di esperti sub-ottimi.

Le nostre proposte rendono possibile progettare facilmente un sistema di controllo gerarchico complesso e imparare la politica di controllo sia interagendo direttamente con l'ambiente, sia fornendo dimostrazioni per alcuni sotto obbiettivi o per l'intero sistema.

Contents

List of Figures

List of Tables

Glossary

A2C
: Advantage Actor-Critic. 10

A3C
: Asyncronous Advantage Actor-Critic. 10

AI
: Artificial Intelligence. 1, 2, 82

CSI
: Cascaded Supervised Inverse Reinforcement Learning. 15, 77

DDQN
: Double DQN. 9, 54, 56

DQN
: Deep Q-Network. 9, 10, 54, 56

EM
: Expectation Maximization. 7–9, 16, 69, 70

eNAC
: Episodic Natural Actor-Critic. 8

GIRL
: Gradient Inverse Reinforcement Learning. 15, 16, 30, 64–66, 71, 72, 75, 83

GPOMDP
: Gradient of a Partially Observable Markov Decision Process. 7, 8, 26, 30, 43, 44, 48, 54

HAM
Hierarchy of Abstract Machines. 10, 11, 33, 39–41

HCGL
Hierarchical Control Graph Learning. 34, 38, 40, 41, 44–46, 49–51, 53, 82–84

HER
Hindsight Experience Replay. 14

Hi-REPS
Hierarchical REPS. 9

HRL
Hierarchical Reinforcement Learning. 3, 10, 12–15, 33, 34, 38, 40, 41, 82–84

IRL
Inverse Reinforcement Learning. 4, 5, 15–17, 31, 57, 58, 69, 77, 83, 84

KL
Kullback-Leibler Divergence. 8, 9, 43, 51, 52, 69

LQR
Linear Quadratic Regulator. 70, 72–74, 95

MDP
Markov Decision Process. 11–15, 20, 21, 23–26, 33, 38, 40, 41, 54

MDP$\backslash\mathcal{R}$
Markov Decision Process without Reward. 20

ML
Machine Learning. 1, 2

MLIRL
Maximum Likelihood Inverse Reinforcement Learning. 16

NES
Natural Evolution Strategy. 8

NLS

Non Linear System. 70, 74–76, 95

PG

Policy Gradient. 7, 8, 25, 26, 38, 69

PGPE

Policy Gradients with Parameter-Based Exploration. 8, 43, 44, 48, 58, 65, 66, 71

POMDP

Partially Observable Markov Decision Process. 20, 40

PoWER

Policy learning by Weighting Exploration with Returns. 9

PPO

Proximal Policy Optimization. 10

RBFs

Radial Basis Functions. 27, 28, 74, 76, 78, 79

REINFORCE

REward Increment $=$ Nonnegative Factor $\times$ Offset Reinforcement $\times$ Characteristic Eligibility. 7, 30, 74, 77

REPS

Relative Entropy Policy Search. 9, 43, 44, 51, 53

RL

Reinforcement Learning. 2–5, 9, 12, 13, 19–21, 24, 26, 28, 34, 36, 37, 39–42, 50, 53, 54, 81, 83, 84

RWR

Reward-Weighted Regression. 9, 43, 44, 51–53, 69

SCIRL

Structured Classification-based Inverse Reinforcement Learning. 15, 77

SMDP

Semi-Markov Decision Process. 11, 12, 21, 25, 26, 38–40, 50, 54

SOME-IRL

Single-Objective Multiple-Expert Inverse Reinforcement Learning. 57, 58, 63, 65, 66, 69, 71–78, 83, 84

TRPO

Trust Region Policy Optimization. 10

UAVs

Unmanned Autonomous Vehicles. 1, 3, 41

Chapter 1

Introduction

The exponential growth of Artificial Intelligence (AI) has made possible the development of multiple successful applications in different fields e.g., finance, social networks and robotics.

The growing interest in robotics is due to the new and promising applications in the field of Unmanned Autonomous Vehicles (UAVs), in particular the ones related to autonomous car driving. These domains are extremely difficult and represent a hard challenge for classical AI and control theory, as they give rise to new issues that are not considered in this literature. One of the most prominent issues is that some environments can be highly dynamic, and the agent has to deal with different and unattended scenarios. Agents must behave well even in non-standard scenarios and must ensure the safety of others vehicles and people in any working condition. Safety is one of the most important, and discussed, requirements of this kind of applications [1, 2].

Another issue is that they often require to process data directly from low-level sensors, such as laser scanners, cameras, proximity sensors. These complex input signals were not commonly considered by classical AI and control theory: indeed, control theory often works with vectors of scalar signals, while classical AI is based on an abstract high-level representation of the system to be controlled. While there exist some algorithms to treat these types of signals [3, 4] and to ground semantic knowledge on low-level sensory data, the intrinsic complexity of the environment, the required robustness with respect to errors and unexpected scenarios, and the real-time computational requirements make classical approaches not viable.

To face these scenarios, Machine Learning (ML) techniques have been adopted massively, in particular to solve the perception issue [5, 6]. For tasks such as image recognition and segmentation, scene understanding, and others similar problems, Deep Neural Networks have been shown to be extremely successful and are now a basic building block for most complex robotic systems. Thanks to these approaches, it is now possible to accurately and reliably perceive the environment, interpret it, and build a high-level, compressed, representation. Classic algorithms from planning and control theory can then be applied by the agent to solve the requested tasks autonomously e.g., an autonomous car can pick up some passengers and transport them safely to the desired destination in an urban environment, or a robotic pallet can move goods in a warehouse to the appropriate shelf.

There are many strengths of these classical approaches: they can be carefully designed by exploiting domain knowledge, their behavior can be easily predicted and can be analyzed, often with theoretical tools, in particular, but not only, when dealing with control theory methods. In particular, this feature suits the requirement of the safety of the application, at least in the scenarios considered by the developers. Classical approaches, however, have major drawbacks. They are often difficult to build, as the design complexity scales up with the complexity of the system. Even when it is possible to build reliable control systems, their computational requirements may be too demanding and, often, approximate solutions rather than exact solutions to problems must be used. Finally, even if it is possible to design and implement an efficient solution, it may happen that the agent is forced to face unexpected situations. If the control system is not designed to face the new issue specifically, then the system is extremely likely to fail and this event could have catastrophic consequences.

Recent advances in Reinforcement Learning (RL), in particular Deep Reinforcement Learning, have focused on the control of complex (robotics) systems [7, 8]. These approaches are exactly on the other side of the spectrum w.r.t. classical control and AI approaches. They are able to learn policies that can be applied to previously unseen scenarios. They can change the control policy online to adapt to slow and abrupt changes of the environment e.g., wear of mechanical parts or unattended failures of actuators. The complexity of the design does not depend extremely on the complexity of the task to be solved, but all the effort is put on the design of the appropriate learning algorithm and approximator structure. For this reason, the required expert domain knowledge is reduced, at the cost of a sufficient, often large, amount of data, obtained both from simulations and from direct interaction with the environment. The major drawbacks, particularly affecting the most successful Deep RL control systems, are that the learned control policy is extremely difficult to study and verify, and that often their learning performance is unstable. However, the applications of these innovative AI and ML techniques in this field are still far from being adopted in commercial applications, for the two main reasons mentioned above and for other further reasons, such as the huge amount of data required by the learning process, the difficulties in exploiting the domain knowledge, and the lack of reproducibility of the results [9], particularly in the Deep RL approaches.

A line of research that tries face some of the problems of the RL algorithms described above is Hierarchical Reinforcement Learning (HRL). Hierarchical Reinforcement Learning consists in designing hierarchical agents that are able to perform temporally extended actions i.e., actions that are composed of several control cycles, in which the objective is to reach a sub-goal or implement a specified behavior. There is a wide research literature on HRL, which tackles the above problems in different ways and with different objectives. Some frameworks focus on integrating domain knowledge, while others focus on making more efficient the learning process. This field, which had previously lost some interest, has again gained popularity in the community, particularly in combination with the new Deep RL methods. However, recent approaches, while trying to boost performance, particularly when considering transferring knowledge to new tasks, do not directly tackle the issues highlighted above, as was done by the original HRL approaches. Classic HRL approaches, instead, are particularly suitable and designed for classic RL tasks and they do not scale well for the new challenging problems arising from robotics applications.

In this work, we propose a novel framework for HRL, whose objective is to trade off some of the benefits and disadvantages of both classical and deep hierarchical methods. Although there exist already some frameworks able to face problems with continuous action and state spaces, there is no framework that is able to describe in a general and elegant way the design of a hierarchical agent by decoupling the design of the hierarchy structure from the learning algorithm. This concept is fundamental in order to be able to design complex control structures that exploit the designer's domain knowledge, but can also exploit the new and powerful tools taken from the vast literature about RL algorithms. The approach that we describe in this work is strongly inspired by the control theory framework: we believe that the design methodologies used by classical engineering are a powerful tool that naturally fits the design of complex agents, particularly when facing the new challenges coming from the new modern applications of robotics and UAVs.

While the formalization of a control problem in terms of RL problem s may be beneficial to create agents that can perform well even in the extremely challenging environments described above, sometimes it is not enough. There is a vast variety of tasks that are easier to demonstrate than to define or, more formally, using the RL terminology, tasks for which it is easier to provide optimal (or suboptimal) trajectories rather than defining the actual reward function i.e., the performance metric that describes the task. This kind of problem can easily arise in a wide variety of robotics applications. For instance when transporting passengers, it is difficult to describe a metric that defines how "good" is the driving style of the autonomous driver, however, it is easy to provide demonstrations of a driving style considered acceptable by showing demonstrations by a set of human experts. Sometimes, while task performance can be easily measured with a reward function, some of the subtasks that need to be solved by the agent to complete the original task can be difficult to express or difficult to learn e.g., the reward function is very sparse and the algorithm may need reward shaping to learn efficiently. When facing this kind of problems, there are two types of possible solutions: the first is Imitation Learning, the second Inverse Reinforcement Learning (IRL). Imitation learning is a powerful tool for learning the policy of the demonstrator, however the learned policy may be not able to generalize to different tasks (or subtasks). Inverse Reinforcement Learning is a much more powerful tool.

With IRL it is possible not only to learn the policy of the demonstrators, but also to extract the reward function that "explains" the demonstrations i.e., the reward function w.r.t. which the agent is optimal. As the design of a reward function is extremely difficult and often a crucial point in designing subtasks, particularly when dense reward functions are used, it is important to study and design IRL algorithms as they provide a viable alternative to incorporate domain knowledge in an automated way. For this reason, IRL algorithms should be considered as one of the building blocks of any complex learning system, in particular when considering hierarchical frameworks.

IRL is a relatively young field [10], but a wide variety of approaches have been developed. Most approaches have the major drawback of having to solve the direct learning problem multiple times in order to identify the reward function. Some more recent works have proposed methods to retrieve the reward function from demonstration without solving the direct problem multiple times [11, 12, 13, 14]. These algorithms are interesting for our application scenario, since solving the direct problem is often difficult, as solvers are not available and interaction with the environment is often expensive. Among the IRL algorthms that do not need to solve the direct RL problem multiple times, the GIRL algortihm, proposed by Pirotta & al. [14] is of particular interest for our applicative scenario. The main reason for considering this algorithm is that it is based on the policy gradient framework, which makes it particularly fit for the continuous action space problems. However, this algorithm has some drawbacks: the framework considers a single and stochastic expert, the retrieved reward function can be a local minimum, it requires a fair amount of data to reliably identify the reward function.

To improve the above-mentioned work and overcome these drawbacks, we propose a multiple-expert IRL algorithm, which exploits multiple, possibly suboptimal and deterministic, experts while improving the performance of the previous method. The main reason for preferring the proposed IRL method is not only that it has better performance that to the previous one, but also that the multiple expert framework is more appropriate in our scenario: indeed, it is extremely difficult to find a human expert who behaves optimally, while performing a good exploration to explore sufficiently the state space. Since each expert has a fixed, suboptimal policy, it is reasonable to consider different experts to gather different information from the environment.

Chapter 2

State Of The Art

2.1 Policy search and robotics

Policy search and actor critic approaches are particularly useful when dealing with continuous action spaces, that are one of the main issues of robotics tasks. Policy search algorithms are both model based i.e., they try to learn a model of the environment, and model free. Model based approaches are very similar to adaptive control approaches that can be found in control theory. In this work, we will focus on model free approaches, as learning a good model can be generally unfeasible for an arbitrary task. We extensively use policy search approaches, as we focus only on tasks with continuous action variables.

Model free approaches can be divided in three macro categories: Policy Gradient (PG), Expectation Maximization (EM) and information-theoretic methods.

PG methods are one of the first policy search methods that have been developed. The first basic algorithm is REINFORCE [15], where is shown that the gradient of the objective function w.r.t. the policy parameters can be computed without any transition model when the policy is stochastic. An improved version of this gradient algorithm is the Gradient of a Partially Observable Markov Decision Process (GPOMDP) [16, 17] algorithm. The main idea behind this algorithm is that the past reward doesn't depend on future actions. This algorithm reduces the variance of the gradient estimation, resulting in better updates and faster convergence.

The policy gradient theorem [18] fills the gap between policy search and actor critic approaches, by highlighting the relation between policy gradient and the state-action value function. Also, this work describes compatible function approximators, that are function approximators for the state-action value function such that, when used instead of the actual value function in the gradient computation, the obtained gradient estimation is unbiased. A function approximator is compatible with the policy if the derivative of the function approximator w.r.t. the approximator's weights is equal to the derivative of the logarithm of the policy w.r.t. the policy weights. It can be shown that, when using compatible function approximators, the actor critic algorithm resulting from the policy gradient theorem is exactly equivalent to the GPOMDP algorithm [19]. Other actor critic approaches have been developed for both the on policy [20] and off policy [21] scenario. Furthermore in [22] the authors propose an actor critic approach that computes the gradient of a deterministic policy, while learning off policy using a stochastic version of the previous one.

The traditional gradient methods are not representation invariant i.e., if a linear transformation on the policy parameter is applied, also the resulting gradient is changed. This may cause issues when the scales of the parameters are different. To overcome this issue, instead of the vanilla gradient, it can be used the natural gradient. The natural gradient instead of using the euclidean metric to compute the gradient, uses the Fisher information matrix as metric for the space. As the Fisher information matrix is a second order approximation for the Kullback-Leibler Divergence (KL), by bounding the update on the natural gradient we have an (approximate) bound on the KL between two policy distributions. One algorithm exploiting both the natural gradient and the actor critic scheme suggested by the policy gradient theorem is the Episodic Natural Actor-Critic (eNAC) [23].

While most of the PG approaches are based on differentiable and stochastic policies, there are also black box approaches that, instead of exploring by adding noise to the actions, use a distribution of policies, exploring at parameter level. The main advantages for these approaches are the possibility to use non differentiable policies and the small variance of the gradient estimation. Two of this approaches are the Policy Gradients with Parameter-Based Exploration (PGPE) [24] and the Natural Evolution Strategy (NES) [25] approaches.

Another class of policy search algorithms are the EM approaches. Here, the policy search problem is seen as an inference problem where the reward is treated as an improper probability distribution, and the observed trajectories as latent variables. In order to use the reward as a degenerate probability distribution, the reward must be transformed to be strictly positive. There are many ways to obtain this property, one is to subtract the minimum reward, if the reward is bounded. However, the most useful and generic reward transformation is the exponential transformation, where a temperature parameter β is multiplied by the actual reward. The temperature parameter is useful to give more (or less) relative importance to high rewards. The most important EM approaches are the Reward-Weighted Regression (RWR) [26] and the Policy learning by Weighting Exploration with Returns (PoWER) [27]. In particular, RWR is the simplest black box EM approach that updates the policy distributions by weighted maximum likelihood estimate, using as weights the sum of the discounted returns, after the exponential transformation.

The last class of policy search approaches are the information theoretic approaches, where the problem is formulated as an optimization problem subject to a KL bound. The KL bound is used to mitigate the problems that may occur when the update changes abruptly the policy: this issue, typical of EM techniques, may cause instability during the learning and, if a real robot is used, damages to the agent and the environment. The most important algorithm in this class is the Relative Entropy Policy Search (REPS) [28, 26] algorithm. There are many versions of the REPS algorithm, but in this work we will use the black box formulation. An interesting variant is the Hierarchical REPS (Hi-REPS) [29] algorithm. Despite the name Hi-REPS use an extremely limited hierarchical structure: the algorithm indeed tries to learn different modes of the reward distribution, and selects one of the learned modes at the beginning of each episode. This algorithm is designed to learn multiple solutions for a task more than a hierarchical structure, although future extensions may exploit the same approach to learn a more complex hierarchy.

2.2 Deep Learning approaches

Neural networks have been used in RL since the TD-Gammon [30] algorithm, that exploits a neural function approximator to learn to play the backgammon game. However, neural networks as function approximators have not been used extensively in RL until the success achieved by the Deep Q-Network (DQN) algorithm [31] in solving Atari games using as inputs raw images. After this work, the research has focused in finding network architectures and algorithms to have a faster and more stable training procedure for deep neural approximators for state-action value functions. One of the relevant works on this direction is the Double DQN (DDQN) approach [32], that takes inspiration from the Double Q-Learning [33] algorithm. However the deep approach does not maintain the same theoretical properties of the vanilla Double Q-Learning algorithm, while still improving the performance w.r.t. the DQN algorithm by reducing the overestimation of the state-action value function. Other variants of the DQN approach have been proposed to face different issues, such as the exploration [34] or the state-action value function estimate [35, 36]. To face also continuous actions environments, actor-critic approaches have been developed. One of these approaches is the Deep Deterministic policy gradient [37] algorithm, based on the previous work on deterministic policy gradient actor critic [22]. Another approach is the Asyncronous Advantage Actor-Critic (A3C) [38], that uses the advantage function for the critic and runs in parallel multiple learning instances. However, a synchronous version, the Advantage Actor-Critic (A2C) has been shown to achieve similar performances, highlighting that the asynchronous learning is not needed to achieve that performance. More recent works have focused on finding efficient policy search algorithms to train deep networks. The most successful approaches are the Trust Region Policy Optimization (TRPO) [39], and the Proximal Policy Optimization (PPO) approach [40].

2.3 Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning (HRL) is based on the idea that the solution for complex tasks can be found by decomposing the original task in simpler subtasks. The assumption is that a complex task is easier to solve if it is seen as a set of high-level actions, rather than learning the low-level actions.

There are several approaches, methodologies and issues related to HRL [41]. The fundamental issue is how to model the decomposition in subtasks. The principal frameworks we ca found in literature are three: Hierarchy of Abstract Machines (HAM) [42], MAX-Q [43] and the Options [44] framework.

The HAM framework is one of the first approaches in HRL. In this approach, the hierarchical policy is composed by a set of abstract machines. An abstract machine is a finite state machine with four different types of states: action, call, choice and stop. The action states are states where an action is applied to the environment. The call states suspend the execution of the current machine, and starts the execution of another machine. The stop states terminate the execution of the current machine, and the execution of the machine that called the current one is resumed. The choice states are states in which the next state of the machine is chosen nondeterministically. Every abstract machine H_i is defined by the state set S_i, an initial state function I_i that, given an environment state $x \in \mathcal{X}$, returns in which state each called machine starts, and a stochastic transition function δ_i that defines the transition probability to the next machine state $s_{t+1} \in S_i$ given the current environment state x and the current machine state s_t. A Hierarchy of Abstract Machines H is the closure of all the states of all the machines reachable by the initial machine H_0. The initial machine H_0 should not have a stop state. If $\mathcal{M}$ is an Markov Decision Process (MDP), the only relevant states of the composition $H \circ \mathcal{M}$ are the ones where a choice state of H is reached, as all the other states has a well defined policy forced by the execution of the HAM. Thus, *reduce*$(H \circ \mathcal{M})$ is a Semi-Markov Decision Process (SMDP) derived from the original MDP, that considers only the states where a choice state of H is reached, and standard algorithms for SMDPs can be used. The objective of the HAM framework is to restrict the set of policies that could be learned, by exploiting domain knowledge of the machine designer. Indeed, while the optimal policy for the SMDP can be learned, it can be different from the optimal policy for the original MDP: the performance of the hierarchical policy highly depend on the ability of the designer.

The MAX-Q approach is principally used to solve finite MDPs and is based on the decomposition of the root task in subtasks. The original MDP is decomposed in a hierarchy of SMDPs by means of a direct acyclic graph called task graph. Each subtask is defined by the tuple $M_i = \langle T_i, A_i, \tilde{\mathcal{R}}_i \rangle$ where T_i is a termination predicate i.e., a function that discriminates whether the current state is an absorbing state for the subtask or not; A_i is the set of admissible actions for the subtask, which can be either primitive actions or other subtasks; $\tilde{\mathcal{R}}_i$ is the pseudo-reward function of the subtask, that typically is a reward function that returns a negative value for every state except the desired subtask terminal state, where the reward is 0. Solving the root subtask M_0 means solving the whole task. In MAX-Q the subtasks are executed as subroutines, following a stack discipline. The objective of MAX-Q is to learn the projected value function $V^\pi(u, x)$ at each level of the hierarchy i.e., the expected cumulative reward of each subtask. The learning algorithm is based on the following decomposition of the state-action value function:

$$Q^\pi(i, x, u) = V^\pi(u, x) + C^\pi(i, u, x),$$

Here, the value function of a primitive action is considered to be the expected reward obtained by using such action, while the completion function C^π is the expected discounted cumulative reward of completing subtask M_i after the termination of the selected subtask M_u in state x. In order to exploit the pseudo-reward function, as the completion function is needed to compute the projected value function of each subtask, two different completion function are learned at each level of the hierarchy. One completion function is learned by considering only the original MDP reward function, and the other one uses the pseudo value function. The first completion function is the one that is propagated at the upper levels of the hierarchy, while the second is used locally to compute the subtask's best action. This is done to avoid the propagation of the pseudo-reward function to other levels of the hierarchy. An important remark is that, when this approach is used in finite MDPs, the number of parameters to learn increases with the number of subtasks, as a projected value function for each subtask must be learned. This lead to a structured and useful representation of the value function, that is particularly useful for transfer learning, but can slow down the learning process. To face this issue state, abstraction is applied to group together states that are indistinguishable from the point of view of the considered subtask, reducing the number of parameters to learn at each level. The MAX-Q algorithm is not able to learn globally optimal policy, but it is proven to learn a recursive optimal policy, that is a hierarchical policy where the policy of every subtask is optimal, by assuming fixed the policy of its subtasks. A recursive optimal policy can be a optimal policy for the MDP if the subtask decomposition is done properly for the problem, but it is different w.r.t. the optimal one in general e.g., when concurrent subtasks can be optimized together.

The option framework is the most successful HRL approach. Option are temporally extended actions defined as the tuple $o = \langle I, \pi, \beta \rangle$, where I is the initiation set i.e., the set of states when the option o can be executed; π is the option policy; β is the termination condition. When an option is executed, the policy option is followed to select actions until the option terminates according to β. If the option is Markovian, then the termination condition must be a probability distribution over the states i.e., $\beta(s)$ is the termination probability of the option in the state s. Also the option policy must be a policy over primitive actions. A more useful family of options are the semi-Markov options. These options can terminate after a specified number of steps, using the whole history for the termination condition. An option is semi-Markov also if its policy is defined over the set of options instead of the set of actions. Normally, option policies and termination conditions are built exploiting domain knowledge. However it is possible to learn meaningful options policies by designing option-specific reward functions. As the options induce a SMDP over the original MDP, all the theory for SMDPs can be applied to this framework. In particular, it is quite easy to extend the basic RL algorithms, such as Q-Learning and SARSA, to the options framework.

More advanced algorithms have been developed for learning with options, such as intra-option learning. The idea is to exploit the information before the option terminates and to share the information between multiple options. The most simple example is the one-step intra-option Q-Learning algorithm [45]. The objective of this framework is not to restrict the possible actions performed by the agent, but to enrich the action set to improve exploration. Primitive actions can be seen as one-step options. In this way, the optimal policy over the set of primitive actions is the same as the policy over options. Options are then a way to speed-up learning, particularly in the initial learning phase.

While the first HRL algorithms focus mostly on value based approaches for RL, in [46] Ghavamzadeh & al. have described a hierarchical policy gradient formulation. Furthermore, the authors describe also an hybrid algorithm, where the high-level algorithm is a value based, while the lower level is a policy gradient algorithm.

Some works on RL also tried to learn the task decomposition by learning to extract and find subgoals in the environment. There are several approaches to this problem: some approaches [47, 48, 49, 50] are based on finding bottlenecks states i.e., states that are connections between different partitions of the MDPs, such that is difficult to move between two partition without visiting such states, e.g., all states near a doorway in a building environment. An approach that achieves similar results is described in [51], however instead of searching structural bottlenecks, the subgoals are extracted using an information theory criterion. Others approaches are based on clustering states [52, 53] to generate meaningful subgoals. The last relevant approach in sub-goal discovery is the skill chaining approach [54]. Here, the options are learned sequentially. When a target event is reached, an option to reach that target is learned. After a period of off-policy option learning, also the initiation set for that option is learned. Finally, the new option is added to the list of available options and its initiation set is considered as new target event. This procedure, given an initial target event e.g., the initial goal, grows a tree of skills towards the initial target event.

Automatic subgoal discovery is a very promising and appealing field, however, the algorithms described above are either only meaningful when dealing with finite state environments or need external signals and domain knowledge from the expert or extract meaningful subgoals only in well-defined types of environments. Due to this issue, it is difficult to use one of the existing methods to design hierarchies from scratch in real-world environments, where the handcrafted design of agent's hierarchy is fundamental.

More recent approaches have tried to apply some concepts of HRL in the Deep Learning framework. Feudal Networks [55] are based on one of the first approaches of HRL, the Feudal Q-Learning algorithm [56], where a higher level controller is producing objectives to the lower level controllers. While the original work focuses on state space and abstractions on multiple levels, in this work the authors focus on a two levels approach, where the high-level exploits a special learning algorithm called transition policy gradient, that is able to boost the learning of the high-level controller when the low-level controller is not yet able to reach the target.

In [57] the authors presents a policy gradient formulation that is able to learn options policy and termination condition without an explicit sub-goal discovery algorithm. The learning is only driven by the extrinsic reward, however an intrinsic reward signal can be used to characterize the learned skills and improve learning.

A very promising Deep Learning approach is the Hindsight Experience Replay (HER) [58]. This method tries to exploit the knowledge of bad trajectories to improve learning, particularly in the case of sparse reward signals. In this work, subgoals are sampled from reached states and are used to learn a Universal Value Function Approximator i.e., a neural network that tries to learn a value function for every possible sub-goal in the environment. The key idea is that learning to reach already known states can be helpful by generalizing the policy for unseen ones, as the underling dynamics of the MDP doesn't change while changing the reward function.

The Hierarchical Actor-Critic [59] algorithm exploits the HER formalism, by generalizing to a multiple level structure, where the subgoals for the low-level algorithm are produced by a higher level controller. Differently from [57], where the termination condition is learned, in the Hierarchical Actor Critic algorithm the focus is to learn fixed horizon subgoals.

The focus of this Deep Learning approaches is to improve the learning performances in difficult environments. These algorithms are designed to scale better when the complexity of the task increases, but they do not mitigate the issues of flat Deep Learning approaches, such as policy interpretability, data hungriness, reproducibility, learning stability. Also, they do not allow to exploit easily domain knowledge.

Other Deep HRL methods are based on pre-training the low-level skills. In [60] a pre-training environment is build and then the skills are learned by a single stochastic neural network. In [61], the authors perform high-level planning on a latent trajectory representation learned using an extension of the Variational Auto Encoder [62]. In [63], a hierarchical architecture for lifelong learning is presented. Skills are learned by pre-training, then each network is stored into an array, in order to be reused. To reduce the growth of the memory usage, in order to achieve the lifelong learning objective, a single network model is proposed, where all the skill share the hidden layers, having different output layers trained by policy distillation. At each time step, a high-level controller chooses whether to use an already learned skill or a low-level action.

Deep learning approaches have been used also in control tasks. In [64], the authors propose a two level hierarchical controller, where a simulated two-legs robot is trained to navigate through obstacles and narrow passages, and also to dribble a soccer ball. Here the high-level and low-level controller work at different time scales: the high-level actions, that are the low-level goals, are produced with a frequency lower than the low-level actions, that are the control efforts applied to the simulator. This approach shares some small similarities with the work presented in this thesis however, they do not propose a general framework for the design of HRL agents.

2.4 Inverse Reinforcement Learning

Learning from demonstration is a relatively young, but quickly growing field, that finds application in those domains, such as robotics [65], where learning from scratch is usually unfeasible. Inverse Reinforcement Learning (IRL) is a subfield of learning from demonstration, where the reward function is learned rather than defined a priori.

The approaches presented in literature can be roughly classified into two categories: model-based and model-free. Model-based approaches [10, 66, 67] require the MDP model or the possibility to solve the MDP given a reward function. Since these approaches are sample-inefficient and expensive, focus shifted to the design of model-free approaches that do not require to solve MDPs [68, 69, 14]. The Gradient Inverse Reinforcement Learning (GIRL) algorithm [14], tries to recover the reward function that makes the gradient of the expert's policy vanish. This approach requires that the expert policy is known or that it can be estimated from the expert's samples. Boularias et al. [11] proposed to recover the reward function by minimizing the *relative entropy* between the empirical distribution of the state-action trajectories demonstrated by the expert and their distribution under the learned policy. Although their approach does not need to solve any MDP, it requires samples collected by an exploratory (possibly non-expert) policy. The Classification-based approaches can produce nearly-optimal results under mild assumptions. To this category belong the Structured Classification-based Inverse Reinforcement Learning (SCIRL) [12] and the Cascaded Supervised Inverse Reinforcement Learning (CSI) [13] algorithms. They require the expert to behave deterministically and need to use heuristics to learn when only experts' trajectories are provided. In general settings, they require exploratory trajectories in order to generalize over the system dynamics. Furthermore, they are designed for problems with finite action spaces.

The common property of the mentioned approaches is that they assume the demonstrations to be provided by a single expert, but this assumption is hard to guarantee in practice. In particular, this assumption can be very limiting in real-world applications where it is common to have multiple experts aiming at the same goal, but behaving differently. A few notable exceptions have addressed this problem in literature mainly focusing on the case of multiple reward functions. For instance, [70] have combined EM clustering with IRL. The experts' behaviors are directly inferred from the provided trajectories and reward functions are defined per clusters. This algorithm, named Maximum Likelihood Inverse Reinforcement Learning (MLIRL), generalizes several IRL approaches [71, 66], but, like the approach described in [72], it needs to compute the optimal action-value function through gradient ascent. While MLIRL requires the definition of the number of clusters in advance, the nonparametric Bayesian approach presented in [73] automatically infers such number. It exploits a Dirichlet process mixture model to solve the IRL problem with multiple reward functions. In [74], the authors present a Bayesian approach able to generalize between multiple-intent and multiple-expert scenarios. While this approach is extremely general, as it covers all possible IRL instances, it does not provide an efficient solution to the specific problems, as it is based on approximated sampling techniques.

Later advances in IRL have focused on automatic features construction, focusing on the single expert problem. Some approaches are based on Deep Neural Networks to retrieve the unknown reward function and are based on the Maximum Entropy Framework [66]. For this reason, they require to interact with the environment to provide samples. In particular, the Maximum Entropy Deep Inverse Reinforcement Learning [75] is a Deep Learning extension of the original Maximum Entropy Framework, while the Guided Cost Learning [76] propose a similar framework where learning the expert's policy is interleaved with the learning of the reward function, exploiting the guided Policy Search algorithm [77]. One of the advantages of this approach is that there is no need to solve the direct problem from scratch multiple times, as done in the firsts IRL algorithms. However, it is still necessary to solve the direct learning problem. It was later shown that there is a strong connection between this algorithm and Generative Adversarial Networks [78]. The work proposed in [79], is based on the GIRL framework and exploits linearly parametrized features. As in GIRL, and differently from the Maximum Entropy approaches [75, 76], there is no need to solve the direct problem to extract the reward function. The feature construction problem is solved by considering the agent policy structure: features are built by computing the orthogonal space of the gradient of the policy, such that every possible linear combination of the features makes the gradient of the expert policy vanish.

Some recent advances in IRL have focused on scenarios different from the mentioned ones. Multi agent IRL, in the particular case of two players zero-sum discounted stochastic games, has been explored in [80], where the reward function is retrieved even in the case of suboptimal agents. In [81] the repeated IRL problem is introduced. This problem consist in an autonomous agent that solves multiple tasks, while supervised by a human expert that corrects the agent only when its performance is considered not acceptable.

Chapter 3

Foundations of Reinforcement Learning and notation

In this chapter we outline some of the fundamental concepts in Reinforcement Learning and define the notation that we use in this work.

3.1 The interaction model

In the RL problem is modeled using two main components:

- the agent i.e., the decision maker and the learner;
- the environment i.e., everything outside the agent.

The agent interacts with the environment as in Fig. 3.1 by selecting an action according to the perception of the state of the environment, while the environment reacts to the agent action by evolving to a possibly different state.

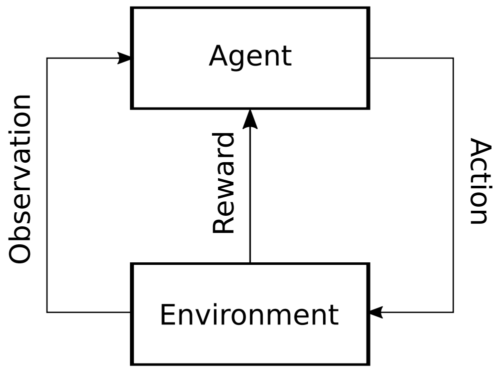


Figure 3.1: The interaction between the agent and the environment

The agent observes the state of the environment and the interaction repeats again. It is important to notice that the agent in Reinforcement Learning is just the decision maker, and everything outside it is the environment, e.g., the state of a robotic agent, such as position, velocity or other state variables are part of the environment, and not of the RL agent. The agent can have an internal state, e.g., when using a non-Markovian policy, and the environment can be partially observable i.e., the observations of the agent are not a complete representation of the state of the environment.

3.2 Markov Decision Process

A Markov Decision Process (MDP) is a tuple $\mathcal{M} = \langle \mathcal{X}, \mathcal{U}, \mathcal{P}, \mathcal{R}, \gamma, \iota \rangle$, where:

- $\mathcal{X}$ is the state space;
- $\mathcal{U}$ is the action space;
- $\mathcal{P}$ is a Markovian transition model where $\mathcal{P}(x'|x, u)$ defines the transition density for reaching state x', starting from state x and applying action u;
- $\mathcal{R}$ is the reward function: $\mathcal{R}(x, u, x')$ is the reward obtained by the agent when it takes the action u in state x, reaching the state x';
- $\gamma \in [0, 1)$ is the discount factor;
- ι is the distribution of the initial state.

In this work, we always consider the state space to be $\mathcal{X} \subseteq \mathbb{R}^n$ or $\mathcal{X} \subseteq \mathbb{N}$, and the action space to be $\mathcal{X} \subseteq \mathbb{R}^n$ or $\mathcal{X} \subseteq \mathbb{N}$.

3.3 Markov Decision Process without Reward

A Markov Decision Process without Reward (MDP$\backslash\mathcal{R}$) is defined similarly to an MDP as a tuple $\mathcal{M}_{\backslash\mathcal{R}} = \langle \mathcal{X}, \mathcal{U}, \mathcal{P}, \gamma, D \rangle$. The only difference w.r.t. an MDP is the absence of the reward function.

3.4 Partially Observable Markov Decision Process

A Partially Observable Markov Decision Process (POMDP) is a tuple $\mathcal{M}_{\text{PO}} = \langle \mathcal{X}, \mathcal{U}, \mathcal{O}, \mathcal{P}, \Omega, \mathcal{R}, \gamma, \iota \rangle$, where:

- $\mathcal{O}$ is the observation space;
- Ω is a the observation model, where $\Omega(o|x, u)$ is the conditional probability of getting the observation o while reaching state x by taking action u.

3.5 Semi-Markov Decision Process

A Semi-Markov Decision Process (SMDP) [82, 43] is a tuple $\mathcal{M}_S = \langle \mathcal{X}, \mathcal{U}, \mathcal{P}, \mathcal{R}, \gamma, \iota \rangle$ where:

- $\mathcal{P}$ is a transition model where $\mathcal{P}(x', N|x, u)$ defines the transition density for reaching the state x' in N time steps starting from state x and applying action u;
- $\mathcal{R}$ is the reward function where $\mathcal{R}(x, u, x', N)$ is the expected discounted reward obtained by the agent when it takes the action u in state x, reaching the state x' in N time steps.

SMDP are MDPs where an action can take different time steps to be completed.

3.6 Objective Function

The objective of an RL agent is to maximize an objective function that measures the performance of the agent in an environment. In RL there are two main objective functions. The first is the average return objective function:

$$\bar{\mathcal{J}} = \mathbb{E}\left[\lim_{T\to\infty} \frac{1}{T}\sum_{t=0}^{T} r(x_t, u_t, x'_t)\right]$$

The second is the expected discounted reward objective function:

$$\mathcal{J} = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(x_t, u_t, x'_t)\right]$$

In this work, we use the latter for every experiment, however most of the algorithms can be easily reformulated in terms of the average reward objective functions.

It is often useful to consider the return of a specified trajectory τ. We define the average return as:

$$\bar{J} = \frac{1}{T}\sum_{t=0}^{T} \mathcal{R}(x_t, u_t, x'_t)$$

And the discounted return as:

$$J = \sum_{t=0}^{T} \gamma^t \mathcal{R}(x_t, u_t, x'_t)$$

3.7 Policy

A (stochastic) policy is defined by a density distribution $\pi(\cdot|x)$ that specifies for each state x the density distribution over the action space $\mathcal{U}$. Deterministic policies can be seen as a limit case for stochastic policies, where the probability density function is degenerate. In this work we will use principally three kinds of policies. The first two policies are based on the state-action value functions, and are the ϵ-greedy and the Boltzmann policy. This two policy are used only when the action space is discrete or when discretizing a continuous action space.

The ϵ-greedy policy is defined as follows:

$$\pi(u|x) = \begin{cases} 1 - \epsilon + \dfrac{\epsilon}{n_{\text{actions}}} & u = \arg\max_{u'} Q(x, u') \\ \dfrac{\epsilon}{n_{\text{actions}}} & \textit{otherwise} \end{cases}$$

where ϵ is the parameter that specifies the probability of taking a random action and n_{actions} is the number of possible discrete actions.

The Boltzmann policy is defined as follows:

$$\pi(u|x) = \frac{e^{\frac{Q(x,u)}{\tau}}}{\sum_{u' \in \mathcal{U}} e^{\frac{Q(x,u')}{\tau}}}$$

where τ is the inverse temperature parameter. When $\tau \to \infty$, the policy is equivalent to a random policy. Decreasing the value of τ towards 0, the probability mass tends to concentrate around the greedy action.

We define a parametric policy as $\pi_{\boldsymbol{\theta}}(\cdot|x)$, where $\boldsymbol{\theta}$ are the policy parameters. A parametric policy used in this work is the Gaussian policy, particularly useful in continuous actions environments. This policy uses a Gaussian noise exploration. The Gaussian policy is defined as follows:

$$\pi_{\boldsymbol{\theta}}(u|x) \sim \mathcal{N}\left(\mu_{\boldsymbol{\theta}}(x), \Sigma_{\boldsymbol{\theta}}(x)\right),$$

where μ is any function approximator that maps the state space into the mean action value, and Σ is a function approximator that maps the state in the state dependent covariance matrix.

There are several ways to implement a Gaussian policy, depending on the parameterization chosen for the covariance matrix, as the covariance matrix must be positive definite by definition. One possible option is to have a fixed covariance matrix i.e., the policy parameters have no effect on the covariance matrix. Another option is to parametrize the covariance matrix as a diagonal covariance matrix, i.e., the Gaussian noise is independent for each action dimension. There are many parametrizations for diagonal covariance matrices. One possible solution is to learn a vector of standard deviations and take the square root of this vector to obtain the diagonal of the matrix. We will refer to a Gaussian policy with this parametrization as diagonal Gaussian policy. Another possibility is that the standard deviations are learned with a regressor and again the value is squared to compute the variance of each action dimension. We will refer to the policies using this parametrization as diagonal state dependent Gaussian policies. These parametrizations have the drawback that the standard deviation may be negative, as nothing in the parametrization prevents this to happen. It is not a problem as the resulting covariance matrix will be well defined, however may be undesirable. One possible solution is to parametrize the logarithm of the variance instead of learning the standard deviations. The last useful parametrization for the covariance matrix is the Cholesky parametrization, where a full covariance matrix is learned by learning the elements of a triangular matrix. The resulting covariance matrix is computed by multiplying the triangular matrix by its transpose.

3.8 Value Function

The value function is a function that measures the performance of a given policy in a state. Formally, the value function of a policy π in a given state is the expected discounted return obtained by following the policy π, starting from the considered state:

$$V^\pi(x) = \mathbb{E}\left[\sum_k \gamma^k r_{t+k} \middle| x_t = x\right],$$

where $r_i = \mathcal{R}(x_i, u_i, x_{i+1})$ and $u_i \sim \pi(\cdot|x_i)$. Value functions can be defined in a recursive way as follows:

$$V^\pi(x) = \mathbb{E}\left[r_t + \gamma V^\pi(x_{t+1}) | x_t = x\right].$$

This expression can be written in an extensive form, to highlight the dependency of the value function with the policy, the transition model, and the reward function of the MDP.

$$V^\pi(x) = \int_{\mathcal{U}} \pi(u|x) \int_{\mathcal{X}} \mathcal{P}(x'|x,u) \left(\mathcal{R}(x,u,x') + \gamma V^\pi(x')\right) dx' du.$$

While state value functions are useful to evaluate how good a state is under a policy π, they are not straightforward to use when we need to evaluate which is the best action to perform in every state. In order to evaluate the best action in every state, we define the state-action value function as follows:

$$Q(x,u) = \mathbb{E}\left[\sum_k \gamma^t r_{t+k} \middle| x_t = x, u_t = u\right].$$

The value function and the state-action value function are related by the following equation:

$$V(x) = \int_{\mathcal{U}} \pi(u|x) Q(x,u) du$$

In any MDP, a policy π^* that is greedy w.r.t. the state-action value function i.e., performs in every state the action with highest value, is an optimal policy. Thus V^* is the optimal value function and Q^* is the optimal state-action value function. The optimal value function is the solution of the Bellman equation:

$$V^*(x) = \max_u \int_{\mathcal{X}} \mathcal{P}(x'|x,u)\left(\mathcal{R}(x,u,x') + \gamma V^*(x')\right) dx'. \tag{3.1}$$

The same can be written for the state-action value function:

$$Q^*(x,u) = \int_{\mathcal{X}} \mathcal{P}(x'|x,u)\left(\mathcal{R}(x,u,x') + \gamma \max_{u'} Q^*(x',u')\right) dx'. \tag{3.2}$$

Notice that the following equality holds:

$$V^*(x) = \max_u Q^*(x,u).$$

3.9 Policy search

Policy search algorithms are RL algorithms that, instead of learning a state-action value function to solve an MDP, directly learn a policy. The policy search approach can be formalized as this optimization problem:

$$\pi^* = \arg\max_{\pi} \mathcal{J}(\pi),$$

where π^* is the optimal policy and $\mathcal{J}(\pi)$ is the expected discounted return where the actions are sampled according to the policy π.

Most of policy search algorithms are based on parametrized policies. These algorithms try to maximize the objective function w.r.t. the policy parameters to find the optimal policy:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}),$$

where $\boldsymbol{\theta}^*$ are the optimal policy parameters.

Some policy search algorithms use a distribution of parametrized policies instead of a single parametrized policy. Generally, a parametric distribution is used. We define a parametric distribution $\mathcal{D}_{\boldsymbol{\rho}}$ where $\boldsymbol{\rho}$ are the distribution hyperparameters. These algorithms try to maximize the objective function w.r.t. the policy distribution hyperparameters to find the optimal policy distribution:

$$\boldsymbol{\rho}^* = \arg\max_{\boldsymbol{\rho}} \mathop{\mathbb{E}}_{\boldsymbol{\theta}\sim\mathcal{D}_{\boldsymbol{\rho}}} \left[\mathcal{J}(\boldsymbol{\theta})\right],$$

where $\boldsymbol{\rho}^*$ are the optimal policy distribution parameters.

3.9.1 Adaptive gradient steps

To improve the convergence of PG algorithms, an adaptive learning rate can be used. Adaptive learning rates can be defined in multiple ways. One option is to constrain the parameters update to be limited in norm, given a metric M, instead of moving proportional to the gradient, as in [83].

$$\begin{aligned} \Delta\boldsymbol{\theta} &= \arg\max_{\Delta\vartheta} \Delta\vartheta^t \nabla_{\boldsymbol{\theta}} J \\ s.t.: &\ \Delta\vartheta^T M \Delta\vartheta \leq \varepsilon \end{aligned}$$

where ε is the norm bound. This update technique prevents undesired parameter jumps, avoiding to use other techniques such as gradient clipping. In this work we use the euclidean metric i.e., the identity matrix, however another good metric can be the Fisher information matrix.

3.10 Learning in SMDPs

When dealing with SMDPs, there are some minor details that must be taken in consideration in order to learn the correct solution of the problem. Differently from version for MDPs, reported in Eq. (3.2), the Bellman equation becomes:

$$Q^*(x,u) = \int_{\mathcal{X}} \sum_N \mathcal{P}(x', N|x,u) \left(\mathcal{R}(x,u,x',N) + \gamma^N \max_{u'} Q^*(x',u')\right) dx'.$$

It is important to notice that the discount factor γ has been raised to the power of N. This will have a major impact in the development of this work. Indeed, this is the main reason why it is not possible to solve an SMDP with plain MDP algorithms, as this will lead to bad policies: we are not taking in account the fact that the actions may lead to the same state, but with a different amount of steps. It is easy to derive an SMDP version of most MDP algorithms e.g., the Q-Learning update rule for SMDPs becomes the following:

$$Q(x,u) \leftarrow Q(x,u) + \alpha \left(\sum_{i}^{N} \gamma^i r_i + \gamma^N \max_{u'} Q(x',u') - Q(x,u) \right)$$

It is possible also to derive policy gradient algorithms in the same way e.g., the gradient computation of the GPOMDP algorithm can be defined as:

$$\nabla_{\boldsymbol{\theta}} \widehat{\mathcal{J}} = \sum^{\mathcal{T}} \sum_{j}^{T} \left(\sum_{t}^{j} \nabla_{\boldsymbol{\theta}} \log \pi(u_t|x_t) \right) \left(\sum_{k}^{N} \gamma^{j+k} r_{j+k} - b_j \right)$$

Notice that, for most of PG algorithms the only difference between the SMDP and the MDP is that the reward of each step is the sum discounted reward of the intermediate time steps.

While deriving SMDP versions of already known algorithms is not a major issue, it may be convenient to use off-the-shelf MDP algorithms to solve in an approximate way an SMDP. We will further discuss this point in detail in the next chapter.

3.11 Features

The use of features is extremely important in RL, in particular when dealing with classical (not Deep) RL approaches to continuous state problems. Most of the time linear policies and action value function approximators on state variables are not sufficiently expressive. With features such as Radial Basis Functions and Tiles, it is possible to work in a space with higher dimensionality, and to represent properly similar states, improving generalization. In this work we will use mainly four types of features: the polynomial features, the tiles, the radial basis functions and the Fourier basis. We describe here below the features as used in our experiments.

3.11.1 Tile coding

Tile coding is one of the most popular feature construction algorithms in classical RL. Each state space variable is discretized in a defined number of partitions. The most simple discretization, and the one used in this work, is the uniform one, where each state variable is discretized in a fixed number of segments of equal length in the range. This results in a discretized version of the state space, where each resulting hypercube, called tile, is indexed. The

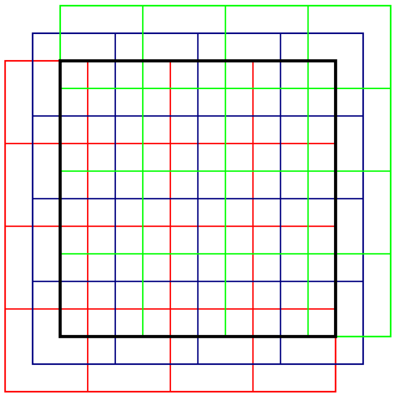


Figure 3.2: Example of tiling in a 2 dimensional space. here 3 tilings, each composed by 4×4 tiles, covers the full state space with uniform displacement. The state space is delimited by the bold black line.

resulting feature vector is a vector of zeros, except a one, corresponding to the index of the active tile. Normally, more than one tiling is used. If more tilings are used, as in Fig. 3.2, they are generated such that there is an offset between each tiling. This offset can be uniform in every dimension, or it can follow a different policy, e.g., use the first odd numbers as proposed in [84].

3.11.2 Radial Basis Functions

Radial Basis Functions (RBFs) are non binary features that measure the distance of the current element w.r.t. the prototype that the feature represents, returning a value in the interval $[0, 1]$. If the current element is exactly the prototype the considered basis represents, then the feature will provide a value of 1, whereas if the distance between the prototype and the considered element tends to infinity, the feature value will approach 0. In this work, we will use Gaussian RBFs that use bell-shaped membership functions with the following form:

$$\varphi_k(x) = e^{-\sum_i \frac{(x_i-\mu_i)^2}{\sigma_i}}$$

where μ is the prototype feature and σ is the scale parameter that describe the “width” of the feature i.e., how fast the feature will go towards zero. In order to cover all the state space, the prototype features are generated over a uniform grid with $n = \prod_i n_i$ prototypes point, where n_i is the number of distinct prototype points projected in the i-th element dimension. The scale is calculated in order to have 25% of overlapping between each feature. This behavior is obtained by computing the scale of the i-th dimension with the

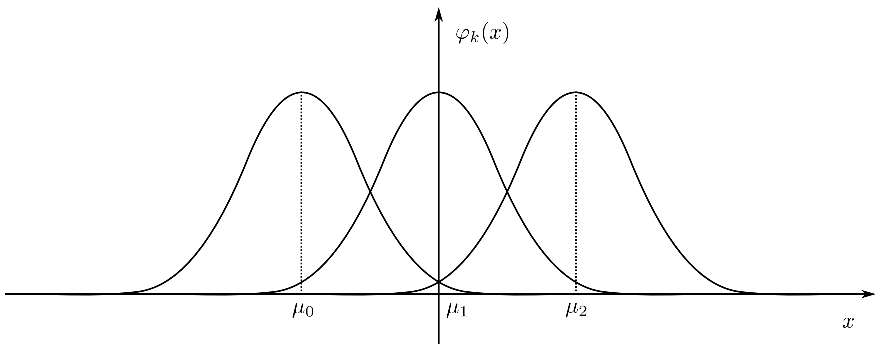


Figure 3.3: Example of one dimensional Gaussian RBFs. Here 3 RBFs are represented, where μ_0, μ_1 and μ_2 are the three prototype features.

following formula:

$$\sigma_i = \frac{(\max x_i - \min x_i)^2}{n_i^3}$$

An example of one dimensional Gaussian RBFs is shown in Fig. 3.3

3.11.3 Fourier Basis

Fourier basis are very well known function approximators for periodic functions. Their use in RL has been introduced in [85]. Their properties can be easily exploited when state variables are bounded to create an universal function approximator on a well specified hypercube of the state space. This is done using the fact that the cosine Fourier basis can approximate every even periodic function on an interval v. Then, if we focus on the semi interval $\frac{v}{2}$, we can approximate any possible function. For building these features, we consider the interval $v = 2$, and we map the state vector on the interval $[0, 1]$. The normalization is performed as follows:

$$x_{\text{norm}} = \frac{x - x_{\text{low}}}{x_{\text{high}} - x_{\text{low}}},$$

where x_{high} is the vector of maximum values for each state variable while x_{low} is the vector of minimum values.

Each feature is defined by a frequency vector c_k, that contains the (integer) frequency selected for each state element for the $k - th$ feature. The feature is then defined as follows:

$$\varphi_k(x) = cos(\pi c_k^T x_{\text{norm}})$$

A feature vector is the set of all possible basis with all possible c_k vectors, where c_k is composed by all the possible combinations of frequencies for each state variable, given a maximum frequency n e.g., if $n = 2$ and the state is composed of 2 state variables, the c_k vectors are:

$$\begin{array}{lll} c_0 = [0,0]^T & c_1 = [0,1]^T & c_2 = [0,2]^T \\ c_3 = [1,0]^T & c_4 = [1,1]^T & c_5 = [1,2]^T \\ c_6 = [2,0]^T & c_7 = [2,1]^T & c_8 = [2,2]^T. \end{array}$$

It must be noted that the vector c_0 corresponds to the constant function i.e., the constant 1.

3.11.4 Polynomial Basis

These features are built such that the resulting linear combination of features is a polynomial of fixed degree. Each feature is obtained by multiplying state variables, each one raised to a given power. The constant feature is always included. Suppose that we want a second order set of polynomial features for the following state vector:

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

The resulting feature vector would be:

$$\varphi(x) = \begin{bmatrix} x_1^2 \\ x_2^2 \\ x_1 x_2 \\ x_1 \\ x_2 \\ 1 \end{bmatrix}.$$

In general, a set of polynomial features of degree n is built by taking all the features built for the degree $n-1$ and adding all the possible polynomials of degree n that uses all the state variable. Each $k-th$ feature of degree n is built as follows:

$$\varphi_k(x) = \prod_i x_i^{d(i)}$$

where $d(i)$ is the degree of the i-th component of the state vector, and is constrained as follows:

$$\sum_i d(i) = n$$

3.12 Gradient Inverse Reinforcement Learning

We now briefly outline the GIRL algorithm, as it is a very important algorithm for some of the topics presented in this work.

The objective of the GIRL algorithm is to recover an unknown parametric reward function $\mathcal{R}_{\boldsymbol{\omega}}(x, u, x')$ given a differentiable parametric expert policy $\pi_{\boldsymbol{\theta}}$ and a set of expert's trajectories $\mathcal{T} = \{\tau_1, \dots, \tau_N\}$. Here, $\boldsymbol{\theta}$ denotes the vector of parameters of the expert's policy, and $\boldsymbol{\omega}$ denotes the vector of parameters of the reward function.

Given a reward function, if the expert policy is stochastic, it is possible to compute the policy gradient w.r.t. the objective function $\mathcal{J}(\boldsymbol{\theta}, \boldsymbol{\omega})$:

$$\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}, \boldsymbol{\omega}) = \int_{\mathbb{T}} p(\tau|\boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log p(\tau|\boldsymbol{\theta}) J_{\boldsymbol{\omega}}(\tau) \mathrm{d}\tau,$$

where $p(\tau|\boldsymbol{\theta})$ is the probability of the trajectory τ given the policy $\pi_{\boldsymbol{\theta}}$, and $J_{\boldsymbol{\omega}}(\tau)$ is the discounted return accumulated by the agent along the trajectory τ. This quantity can be easily estimated using policy gradient methods such as REINFORCE [15] or GPOMDP [19].

The key idea of the GIRL algorithm is the fact that if the expert is optimal w.r.t. a reward function parametrized by $\boldsymbol{\omega}$, then the vector of policy parameters that describe the expert behaviour is a stationary point for the reward function $\mathcal{R}_{\boldsymbol{\omega}}$ i.e., the gradient $\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}, \boldsymbol{\omega})$ will be the null vector.

As the actual gradient is not known and must be estimated, the GIRL algorithm leverages the following optimization to retrieve the expert's parameters:

$$\hat{\boldsymbol{\omega}} = \arg\min_{\boldsymbol{\omega}} \left\| \nabla_{\boldsymbol{\theta}} \widehat{\mathcal{J}}(\boldsymbol{\theta}, \boldsymbol{\omega}) \right\|_2^2 .$$

The GIRL algorithm recovers the reward function that produces the minimum norm gradient i.e., the reward that induces the minimum change in the policy parameters. The main idea behind this algorithm is to recover the reward weights in the chosen space that better explain the behavior of the expert's policy. This assumption holds reasonably well if the magnitude of the gradient is small enough.

A particular interesting scenario is the linear parametrized reward function. In this case, the reward function can be written as:

$$\mathcal{R}(x, u, x') = \varphi(x, u, x')^T \boldsymbol{\omega},$$

where $\varphi(x, u, x')$ is an arbitrary feature vector that depends only on the current state, the current action, and on the next state. If the reward is linearly parametrized, the optimization problem is ill-posed, as it is invariant to scalar factors and admits a trivial solution (the null vector). As commonly done in the literature [10], the issue is solved by constraining the reward weights in the simplex. In this scenario, the IRL problem can be seen as an inverse multi-objective optimization problem, where each component of the feature vector represent an objective that the expert is trying to maximize, and the parameter vector $\boldsymbol{\omega}$ represents the weight given by the agent for each objective. When using a linear reward parametrization, the optimization is particularly simple, as it becomes a convex quadratic programming problem with linear constraints. This makes the resulting implementation particularly simple and computational efficient.

Chapter 4

Hierarchical Reinforcement Learning

In this chapter, we describe a novel approach to Hierarchical Reinforcement Learning. Most of the current approaches are based on the concept of subtasks: the original task is decomposed in a set of smaller tasks. The decomposition can be done using a hierarchical structure such as the task graph of the MAX-Q algorithm, or by creating structured policies as in the option or in the HAM frameworks.

These approaches work particularly well when dealing with MDPs with finite action and state space. However, they are really difficult to adapt to more complex scenarios, particularly when dealing with complex robotics systems. The main issues are that not all the systems can be modeled easily with a simple hierarchical structure, such as the one proposed by the aforementioned methods, particularly when we want to exploit particular aspects of the environment such as symmetry or translation-invariance. Furthermore, in most cases, it is not possible to write a generic hierarchical algorithm, but each task must be solved by a particularly hand-made algorithm, that exploits all the relevant characteristics of the environment and encodes the prior knowledge in the learning process.

In the following, we present how we exploit some basic ideas from control theory that can solve these issues and support the creation of a flexible, generic, and well defined framework.

4.1 A control theoretic approach

One of the fundamental building blocks of control theory is the block diagram. Block diagrams can model complex systems composed by plants, sensors, controllers, actuators and signals. With block diagrams, it is easy to describe the control architecture of any kind of control system, from the simple control of an electrical motor, to a complex industrial plant.

The block diagram is based on blocks and connections. A block can represent a dynamical system or a function, and a connection represents the flow of the input and output signals from and towards other blocks.

Our approach is based on the same core idea, with some modifications in order to fully describe an HRL system. We believe that such representation is beneficial in general for an HRL system, but in particular for robotics applications, where the control system is often already structured (totally or partially) as a block diagram. We will call this approach Hierarchical Control Graph Learning (HCGL).

4.1.1 Formal definition

In order to describe the formal structure of HCGL, we modify the standard interaction model for RL. Instead of having a system composed only by agent and environment we describe the interaction by a Control graph.

The Control Graph is defined as follows:

$$\mathcal{G} = (\mathcal{E}, B, D, C, A) \tag{4.1}$$

where:

- B is the node set;
- D is the data edges set;
- C is the reward edges set;
- A is the alarm edges set;
- $\mathcal{E}$ is the environment.

Each node $b \in B$ represents a subsystem of the control structure. We refer to the nodes in the set B as blocks. Each block can contain either a dynamical system, a function, or an RL agent.

Each block is characterized by a set of input and output signals, that can come from other blocks or from the environment. The input signals are represented in the graph by the data edges $d \in D$. For each block, at each control cycle, we call *state* the vector of the input signals, and *action* the vector of the output signals.

If a block contains an RL agent, then it needs a reward signal in order to measure its performance and to optimize its objective function. The reward signal can be produced either by the environment or by another block. Every edge $c \in C$ represents a reward signal connection.

Finally, each block can produce an auxiliary output called alarm. The alarm signal can be read by other blocks. Every edge $a \in A$ represents an alarm signal connection.

We call the environment $\mathcal{E}$ everything that is outside the set of blocks B, that represents, along with the edges, the hierarchical agent. The environment is modeled in the graph by three special blocks, called interfaces:

- The state interface, that contains the current observation from the environment.
- The action interface, that contains the action to be applied of the environment at the beginning of each cycle, but can be used by other blocks, e.g., to compute a quadratic penalty over the last action used.
- The reward interface, that contains the last reward produced by the environment.

All the edges of $\mathcal{G}$ are directed, and the graph must be acyclic, when considering all edges except the ones that are connected to the action interface. At least one block must be connected to the action interface.

The Control Cycle is the cycle of interaction of the control system with the environment. Each cycle starts by reading an observation from the environment into the state interface. Then, each block of the graph is evaluated by concatenating all their inputs into the state vector, and producing an action, the vector of outputs of the block. Also the information coming from the reward and alarm connections are read, if they exists. Every block must be evaluated after all its inputs, reward and alarm signals have been already computed. This means that the blocks must be evaluated following a topological ordering, where the last block to be evaluated must be the action interface. After the action interface has been updated, the action is applied to the environment and a new control cycle starts. The action interface maintains the stored value until the end of the next control cycle.

The same structure can be used for a continuous time systems, as it is often done in control theory, by defining the control cycle in an appropriate way. However, we focus in this work just on discrete time systems. The temporal evolution of the system is not depending on a fixed clock, but we consider the environment as an event generator that triggers the control cycle when the new state is ready.

4.1.2 Connections

In this model it is very important to have three different types of connection, both theoretically and from the point of view of the practical implementation of the model.

The data connection provides to the model the control input, and must be differentiated from the reward connection, because they have different semantics. Indeed the reward connections carry a signal about the performance of the (sub)system, and they are not needed by the control system to interact with the environment. This semantic difference is also helpful in the implementation, because it makes immediately clear what signals are the ones the algorithm needs to optimize, avoiding any ambiguity with the other input signals.

The last type of connections is the alarm connection. This type of connection is really important to implement control systems that have different time scales. Indeed, in every control cycle, all blocks are evaluated, and this behavior makes impossible to have controllers that work with a different time scale. However, every block can maintain the previous output signal until an event occurs. Events can be notified to others blocks by the use of alarm signals. Every block can generate alarm signals and can exploit the information of the alarm in any useful way. In our model, alarm signals are binary signals that notify the occurrence of an event, but they can be easily extended to carry any type of information needed.

With alarms, event based control can be implemented, e.g., wake one of the RL agents in the control graph only when a particular state of the environment is reached. Also it is possible to create temporally extended actions by raising an event from a lower level agent whenever its episode terminates. By doing so, the higher level agent can only choose different actions when the effect of the related temporally extended action is finished.

4.1.3 Blocks and Induced Environments

Blocks contain the subsystems of the hierarchical agent. In this model, we can look at the single block as an isolated agent and as the remaining part of the graph as a part of the environment. The observations from the environment are represented by the input vector, and the actions applied to the environment are represented by the output vector. We call the environment $\mathcal{E}_i$ seen by each block b_i *induced environment*. The nature of each $\mathcal{E}_i$ depends on the structure of the control graph, but in general it can be a partially observable, time variant, stochastic, non-Markov environment. This makes it impossible to derive general convergence results for arbitrary control graphs, as the theoretical characteristics of the induced environment for each block are arbitrary. This consideration is particularly important, and makes both a careful design of the graph structure, the RL agents, and the performance metrics crucial. A bad design can lead to bad convergence properties. To mitigate this issue, specific learning algorithms can be used, or a multi block algorithm could be exploited.

Blocks are a very general and versatile tool as they can contain either a function, a dynamical system, or an RL agent. When considering blocks that contain a learning algorithm, it is important to consider their induced environments. Induced environments differ from the original environment not only for the different state and input variables but also in the way they evolve: the steps of an induced environment depend on the block activation; furthermore, it is possible to define episodes for the induced environments e.g., after a fixed number of steps or when a particular set of state is reached.

We will introduce some of the most useful blocks we have developed, that are the basic building blocks of most control graphs.

Interface Block is the basic interface to the environment. The environment can read and write data from this type of blocks. The interface block is really important to define a common interface with the environment. In this way, everything is represented by nodes of a graph.

Function Block has the only task of computing a function from the input vector. No learning algorithm is present. All mathematical operations over vectors are function blocks. Function blocks can be used also to compute features vectors from state vectors. Function blocks are generally stateless blocks; however, it is possible to use them with objects in order to build a stateful block.

Control Block is the most important type of block, as it contains a learning algorithm. The purpose of the control block is to interface a generic learning algorithm to the control graph. The control block applies the policy of the agent on the incoming state, and produces the output action. It also collects the transition dataset and feeds it to the learning algorithm when needed.

Control blocks also handle the environment absorbing states. The control block can also consider a termination condition, that can be either a fixed number of steps or a locally absorbing state, i.e., an absorbing state for the subtasks. However, in this framework local absorbing states are fundamentally different w.r.t. environment absorbing states. This is due to the control cycle behavior. Indeed at every control cycle, each block is evaluated, and has to return an action. This implies that in a locally absorbing state, the control block has to provide an action, even if the subtask has been terminated. This behavior is fundamentally different from others HRL methods. This means that a policy for terminal state must be defined for subtasks. This may be a problem when considering finite states MDPs that are often object of interest of most of hierarchical frameworks. However it is not a problem if we do not consider subtasks with absorbing states or when dealing with continuous environments with fine-grained time discretization, where the effect of a single action is not affecting heavily the general behavior. As we will focus on this kind of environments, this is not a major drawback of HCGL.

The final purpose of the control block is to handle signals and synchronization with others blocks. In order to do so, it raises the alarm signal whenever the episode terminates, either if a locally absorbing state or the horizon has been reached. The control has two synchronization behaviors: If there is no alarm connection, the agent policy will be called at every control cycle. If any alarm signal is connected to the agent, then the agent policy will be executed only when one of the connected alarms has been raised. Only the states when the alarm is raised (or the initial environment state) are considered in the collected dataset. The control block maintains the last selected output value until the next event occurs.

Reward Accumulator Block is a particularly important stateful block. This is very important when used in conjunction with cascade control systems that operates at a different timescale. It is possible to accumulate the discounted reward, the sum of the reward, or the mean reward. It is important to notice that for some algorithms, e.g., some PG approaches, it is possible to substitute the discount performed inside the algorithm, with an appropriate accumulator block i.e., use $\gamma = 1$ in the algorithm and discount the reward accumulated with the appropriate discount factor, computed by raising the discount factor of the environment to the number of steps performed until that point. The same is not possible when dealing with value-based algorithms for learning in SMDP, that needs to discount the reward from the last action, and also needs to know how many steps have been performed between the decision points.

Selector Block is a very particular block that allows to select one chain from a list of chains of blocks. The selection is done by the first input of the block, that must be an integer value. Further inputs are passed as inputs of the first block of the selected chain. The output of the selector block is the output of the last block in the chain. The blocks in the chain can be connected to others blocks through the alarm and reward connections. This enables the conditional computation of blocks, allowing to have different low-level RL agents that represent different temporally extended actions.

4.2 Comparison with others frameworks

Existing approaches are mostly based on the concept of macro actions. This is a really important concept of all information technology and it is at the basis of the vast majority of programming languages. The execution of a macro is done following the stack principle, where each macro can call another macro, until a primitive action is called. After the macro (or the primitive action) has been executed, the control returns to the one that activated it. This description particularly fit, at least from the point of view of the RL algorithm, for the options and the MAX-Q frameworks. The HAM framework is also based on this idea, but the learning is performed on the "reduced" SMDP, while the actions are performed by the abstract machine, so the stack discipline does not affect the learning algorithm, as it does when dealing with the other frameworks.

Although this is a powerful approach, at the basis of the exponential growth of information technology in the last century, it is not the most natural approach, neither the most used in control systems. Most of control systems for robotics and automation consist in decentralized controllers that work in parallel, where each controller regulates a specific part of the system. Information about the environment is retrieved by a set of signals. Most of the controllers are driven by the error of a specified state variable w.r.t. a setpoint e.g., the distance to a specified target or the relative orientation of a joint w.r.t. a desired setpoint.

These ideas are fundamentally different from the macro concept. There are two main concepts that must be considered when comparing our framework to other existing frameworks. The first consideration is that the hierarchy is formed by the structure of the control system, and not by the stack discipline. At every time step, data flow following the direction of the connection, into the next block, propagating through the whole graph. This has a major impact on how subtasks are defined. Indeed, in our framework, a subtask is not anymore a function call that executes until termination, but can be better seen as a setpoint to be reached by a lower level controller. While in general it is not a problem, this different view has a major impact on subtask absorbing states. Indeed, no state, excluding the last state of the environment, where no action is required, can be a state in which the subtask terminates. At every control cycle, a controller can either hold the last value or produce a new one, as the data flow is mono directional and there is no way to give back the control to a upper level controller. As it is extremely useful to terminate the episodes of a controller when a particular state is reached, there is the possibility to define task specific absorbing states, however they do not behave as normal absorbing states, as the semantics of the task graph prescribes the generation of an action. This makes the implementation of temporally extended actions with termination a bit problematic, particularly when applying this framework on finite MDPs. We believe that is not a major issue in a continuous environment, if the subtask is seen in an appropriate way e.g., as a setpoint to be reached within a specified threshold, where another action that brings the system closer to the setpoint is not harmful. Another option can be to define dummy, random or fixed actions for this states. The second consideration is on the different approach for the state representation. In most HRL approaches the state seen by each level of the hierarchy is the environment state itself. Some approaches, adopt state abstraction [43] or ad hoc state transformation for subtasks [46]. HCGL is fundamentally different on this point as we consider each block as an independent RL agent. State and action are then related to the induced environment. This makes it possible to use any suitable RL agent e.g., algorithms for SMDPs, POMDPs, non-stationary environment or even classical approaches for simple MDPs. Furthermore, using HCGL the definition of parametrized continuous subtasks is straightforward, and does not need any deviation from the framework definition or special handling. Differently from the Feudal Q-Learning [56], its deep learning version [55] and other deep approaches, we do not add a sub-goal specification as input, with the current state, to the system. We suggest instead of provide a state that contains the interesting information to describe the current state of the system. This is fundamental in order to reuse the classical theory and algorithms of RL into HRL.

All classical HRL frameworks are designed to exploit domain knowledge; however, this is achieved in very different ways. With the MAX-Q approach, the designer's knowledge can be exploited by specifying the decomposition of each subtask in the system and by defining state abstraction: the first is useful to prescribe a structure of the policy, the latter is useful to speedup learning and reduce memory requirements. In the option framework it is possible to specify partially, or totally, the policy of the options. Both HCGL and the HAM framework are based on constraining the structure of the controllers. However, the HAM framework is based on finite state machines. HCGL is extremely powerful, but it may fail in continuous state and actions environments, where the objective could be, e.g., a robot pose. Also, we believe that a block diagram design is more similar than the state machine design to what is already done in this kind of systems e.g., robotics. In most of the HRL frameworks, including ours, it is possible to exploit domain knowledge to specify the subtasks reward functions.

HCGL has been developed with a slightly different objective w.r.t. classical and deep HRL frameworks. The options framework objective is to enrich the MDPs action set in order to improve exploration by following a sub-policy for an extended period of time. The possibility of achieving optimal performance is one of the major selling points of this framework, as it is always possible to take a low-level action instead of the high-level one. Intra-option learning [45] is a further step on this direction. The objective of the MAX-Q framework is not to achieve optimal performances, but to have a factored representation of the state-action value function, in order to reuse the information for different high-level tasks and for transfer learning. The objective of the HAM framework is to exploit the expert knowledge to constrain the policy, improving learning speed by reducing exploration and parameters to learn and imposing the desired behavior. The objective of our framework is twofold: simplify the design of hierarchical agents, by providing a flexible design tool, and favor the reuse of existing RL algorithms. We believe that these two characteristics are particularly suitable for industrial applications and fast prototyping of complex robotics applications, such as e.g., UAVs.

4.3 Experiments

We evaluated HCGL in three different domains: the Ship Steering environment, the Segway environment, and the Prey-predator environment. For each environment, we provide a graphical representation of the task graph. Each box represents a block. The environment is represented by a block containing the three interfaces: the action interface u, the reward interface r and the state interface x. The solid line represents an input connection. The dotted line represents a reward connection. The dot-dash line represents an alarm connection. The selector block is represented by a block containing the block chains. The selector signal is connected to the external block, while the signal that is provided to the block chains is connected to all block chains.

For every experiment, all the parameters of the algorithm are hand-tuned, in order to get the best performance possible. We call an *epoch* a fixed amount of environment episodes, where a number of episodes are used for training, while the others are used only for evaluating the performances i.e.,

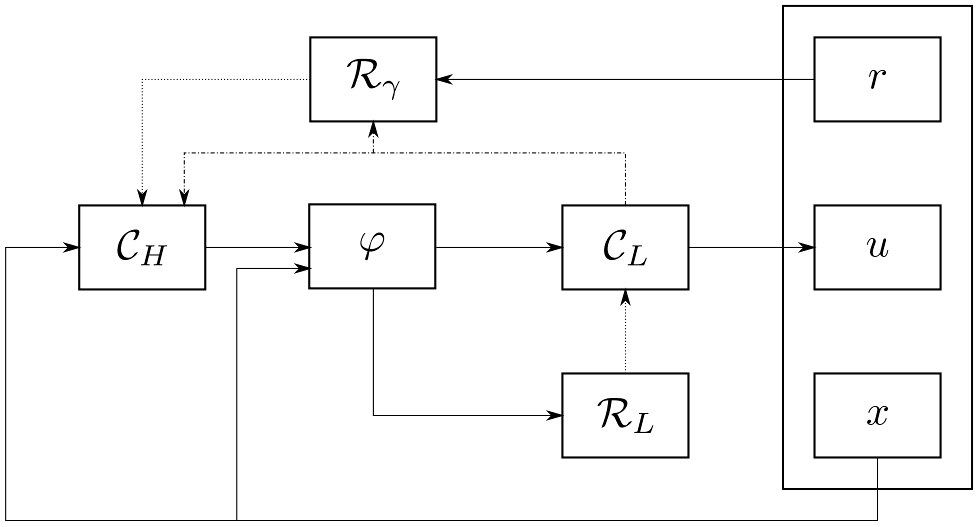


Figure 4.1: Control Graph used in the ship steering environment experiment

no learning is performed, the policy is applied as is on the environment.

For the sake of brevity, we refer to non-hierarchical RL algorithms as *flat* algorithms.

4.3.1 Ship steering

This problem consists of driving a ship through a gate, without going outside a defined region (see [46] and Appendix A.3 for details).

For these experiments we use two different versions of the ship steering environment: the small and the big environment. In these environments the action is applied for three iteration steps.

Small environment

We use this version of the environment to show how a hierarchical formalization of the environment with our framework is beneficial for learning also in environments that are easy to learn with flat policy search methods.

For each algorithm, the experiment consists in 25 epochs of 200 episodes. After every train epoch an evaluation run (with no learning) of 50 episodes is performed.

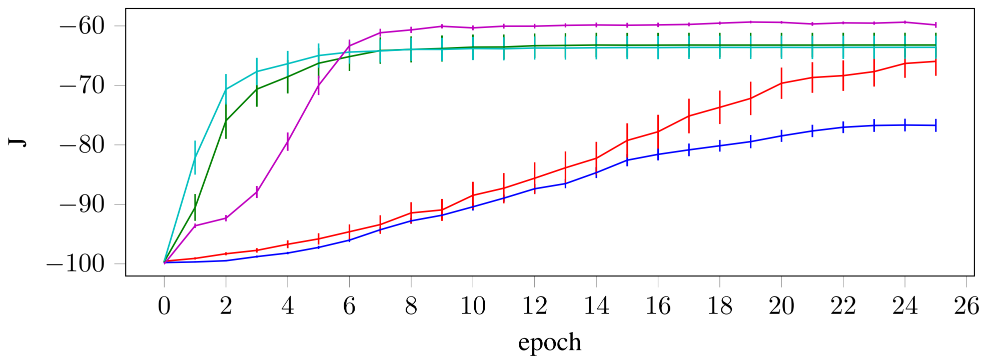
The hierarchical structure we devised is shown in Fig. 4.1. The high-level controller $\mathcal{C}_H$ selects a position setpoint for the ship. The function block φ transforms this position setpoint into the error of the heading of the ship and the distance of the current position and orientation of the ship w.r.t. the given setpoint. The low-level controller $\mathcal{C}_L$ learns a policy to drive the ship towards the selected setpoint, and its output is the desired turning rate of the ship, that is the imposed control action to the environment. The high-level control change its setpoint only when a low-level environment episode terminates. The low-level environment episode terminates after 100 steps or when the distance to the setpoint is less than $0.2m$. The high-level controller uses the reward function of the environment as performance metric. The reward between each different setpoint is accumulated by the reward accumulator block $\mathcal{R}_\gamma$, which properly discounts the reward at each step using the discount factor $\gamma = 0.99$ of the environment. The low-level reward is provided by the function block $\mathcal{R}_L$, that computes the cosine of the heading error.

For this experiment, the high-level controller uses the GPOMDP algorithm and a diagonal Gaussian policy. The policy is a constant mean policy with initial mean $\mu = [75, 75]$ and initial standard deviation $\sigma = [40, 40]$, i.e., the initial policy is a Gaussian centered in the center of the map, and the sampled setpoints are distributed across all the map. The learning rate used for this controller is the adaptive learning rate with parameter $\varepsilon = 10$.

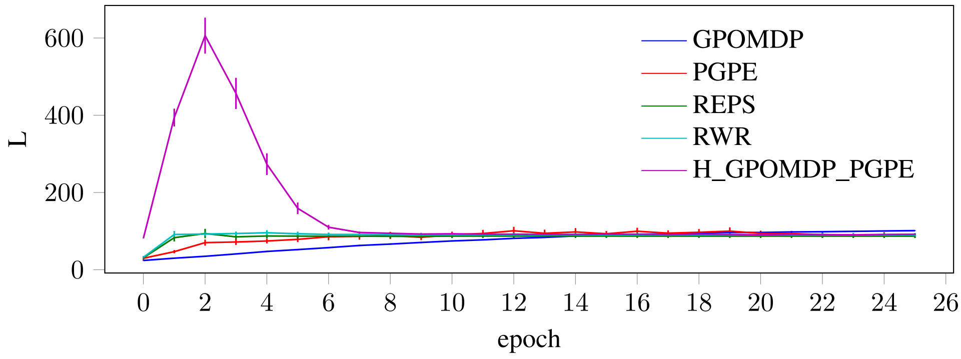
The low-level controller uses the PGPE algorithm. The policy family is a deterministic proportional controller, with forced positive gain parameter, i.e., $u = |k|x$. The initial policy parameters distribution is a Gaussian distribution with mean $\mu = 0$ and standard deviation $\sigma = 10^{-3}$. Also for this controller the selected learning rate is the adaptive learning rate with parameter $\varepsilon = 5 \cdot 10^{-4}$. The high-level controller updates the parameters every 20 episodes of the environment, while the low-level controller updates the parameters after 10 episodes of its induced environment.

All the flat algorithms use tiles as features over the state. A single tiling of $5 \times 5 \times 8$ tiles over the first three dimensions of the state space (x, y and θ coordinates i.e., position and orientation of the ship) is adopted. The GPOMDP algorithm uses a diagonal Gaussian policy that is linear in the state features, with initial standard deviation $\sigma = 0.1$. The other algorithms (PGPE, RWR and REPS) are episodic black box algorithms and use a diagonal Gaussian distribution, with initial standard deviation $\sigma_i = 0.4$. The policy family is a deterministic linear policy over the features. The flat GPOMDP algorithm uses an adaptive learning rate with $\varepsilon = 10^{-5}$. Also the flat PGEP algorithm uses the adaptive learning rate with $\varepsilon = 1.5$. REPS uses a KL bound $\epsilon = 1.0$. RWR uses as temperature for the exponential reward transformation $\beta = 0.7$. The GPOMDP algorithm updates the parameters every 40 episodes. The black box algorithms update the parameters every 20 episodes.

Results are shown in Fig. 4.2. By looking at Fig. 4.2a it is clear that



(a) Objective function



(b) Episode length

Figure 4.2: Learning curves for the ship steering small environment

the policy gradient approaches, PGPE and GPOMDP are the slowest and achieve the worst performance. RWR and PGPE quickly converge to good performances, however they may get stuck to slightly suboptimal policies. RWR converges faster than REPS, but it is slightly more prone to premature convergence, as the distribution variance converges to zero too rapidly. HCGL achieves the best performance on this task, and the results are much less variant, particularly at the end of training. The performance gain and the reduced variance of the learning curve is due to the fact that the HCGL approach can represent a more structured policy, with fewer parameters and more expressiveness in terms of possible behaviors. The hierarchical decomposition splits in two the problem by decomposing the high-level problem (reach a point) from the low-level control task (steer the ship), making possible to reuse the low-level control policy from each point and each orientation of the ship. This results in a policy that is more interpretable and can be easily used in other contexts e.g., the low-level policy learned in the small environment may be used also in the big ship steering environment; the learned high-level policy is a good policy for most of the possible alternative starting points of the environment.

The learning curve behavior can be easily understood by looking at Fig. 4.2b. Flat algorithms tend to increase the episode length progressively, to avoid going outside the bounds. As the gate is located at the end of the diagonal, they will find the gate tending to produce longer trajectories. The HCGL approach, instead, behaves very differently: at the beginning most of the trajectories loop around the center of the environment, increasing the episode length, while the low-level starts to learn to reach the setpoint appropriately and the high-level learns to avoid the map boundaries. When some trajectories are able to cross the gate, the high-level starts to learn the position of the gate and the variance of the high-level policy reduces towards 0. This, jointly with the learning of the optimal low-level policy, reduces the episode length until the optimal performance is reached.

The hierarchical method has some disadvantages w.r.t. the other methods. The first is that the number of steps needed to learn is greater than the other methods, by reducing, however, the number of failed episodes (going outside the bounds). The second is that the learning algorithms and policy must be chosen carefully. Indeed, the low-level policy of the hierarchical algorithm is forced to be stable by computing the absolute value of the proportional gain parameter. This is needed because the induced environment of the low-level policy is partially observable: this causes some ambiguities in interpreting the results of the selected behavior. Without the forced stability, the low-level algorithm can converge to a suboptimal behavior of selecting an unstable policy to force the ship to go out of bounds and terminate the episodes earlier. If this happens, the high-level controller could learn to move the setpoint in the opposite direction w.r.t. the actual goal, which is an undesired behavior.

The possibility of strange interactions between learning processes are a major issue of the framework. To avoid this, a careful design of the policy, good initialization for the policies from expert knowledge could be exploited. While policy initialization is often straightforward e.g., using control theory or prior knowledge, it is not so easy to design a proper control structure and reward function, as it is necessary to analyze the characteristics of the block's induced environment, taking in account possible suboptimal undesired behaviors that may occur. A further step can be done by considering multi-block learning algorithms, in order to let the low and high-level algorithm know the context under which are operating. This could be done by exploiting context based policy search algorithms [86] e.g., by adding as context the current parameters of others controller blocks.

Big environment

With this version of the environment, we aim at showing how our method is able to scale to bigger and more complex problems and to compare it, from both the point of view of the performances and the design of the hierarchical agent, with an existing state of the art method, the hierarchical policy gradient approach shown by Ghavamzadeh & al. [46]. We focus our experimental comparison on this approach as it is the most successful approach that is designed to consider both continuous and state spaces, and consider a control structure totally designed by an expert. More recent works such as the option-critic framework [57], focus instead on learning both the policies and the control structure: this approach may indeed yield better results, particularly for simple problems, but it may fail in real-world scenarios, where the expert knowledge is crucial. For this reason, we focus primarily on the modeling point of view, thus comparing the modeling expressivity of the hierarchical policy gradient approach with HCGL. In this environment, the state space dimension ($1000 \times 1000m$) makes difficult to build generic features that allows for both a complete coverage of the state space and a fine discretization needed to have a fine-grained action selection.

For each algorithm, the experiment consists in 50 epochs of 800 episodes. After every epoch an evaluation run (with no learning) of 50 episodes is performed. It must be noticed that, differently from the small environment, the initial state of the environment is sampled uniformly from $\mathcal{X}$.

For our method we will use the same control graph used for the small environment (Fig. 4.1). The Ghavamzadeh's method is implemented using our framework. This has a minor impact, as our framework is not able to represent absorbing states of local controller as real absorbing states, this implies that the policy must be defined also in these states. However, given the dimension of the environment, the properties of the policy and of the low-level task, and the irrelevance of a single isolated action in the environment make these differences negligible.

The control graph used to implement the Ghavamzadeh method is shown in Fig. 4.3. The function block φ discretizes the state of the environment using a 20×20 tiling discretization over the first two state variables (the position of the ship in the environment). The high-level controller $\mathcal{C}_H$ takes as input the discretized state and selects one of the 8 possible direction to reach. The function block S transforms one of the possible direction in a

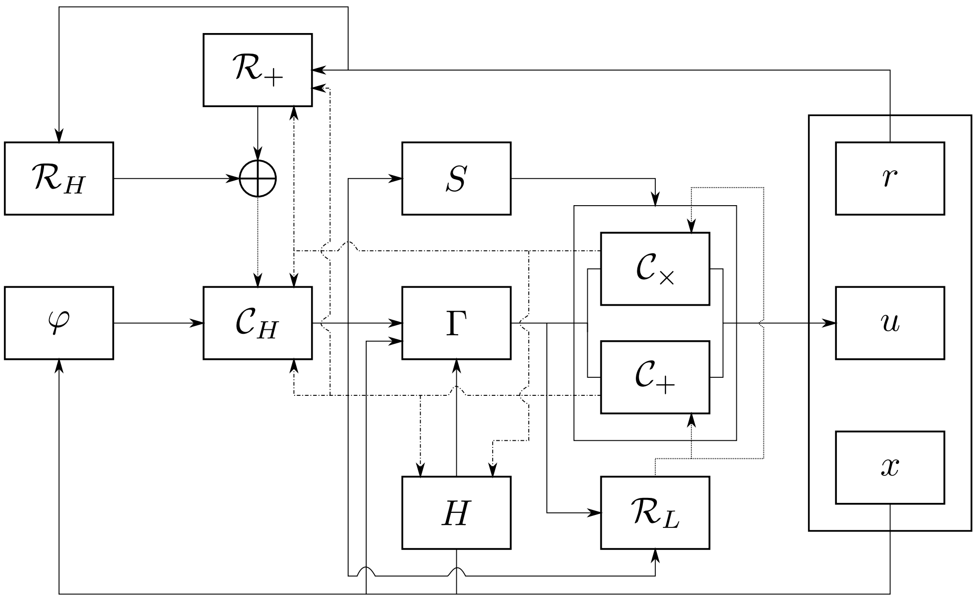


Figure 4.3: Control Graph used to implement the Ghavamzadeh algorithm

binary value, to select the straight/diagonal subtask of the connected selector block. This is done because in the original work, instead of learning each subtask independently, symmetry is exploited to group together the subtasks. The function block H is used to hold the value of the starting position of the subtask. The H is needed because the Γ block performs the needed transformation to map the state of the original environment in the subtask by rototranslating appropriately the current state w.r.t. the position in which the subtask has begun. For the straight subtask the initial state is mapped into $[40, 75]$, while for the diagonal is mapped into $[40, 40]$. All straight subtask are rotated into the “right” subtasks, while the diagonal subtasks are rotated into the “up-right” subtask. The two control blocks $\mathcal{C}_+$ and $\mathcal{C}_\times$ are the two controllers that learns the straight and horizontal subtask respectively. The reward for the high-level block is the sum of the additional reward computed by the function block $\mathcal{R}_H$, that gives a reward of 100 for crossing the gate, and the reward accumulator block $\mathcal{R}_+$, that computes the sum of every reward during the low-level episodes. The reward for the low-level controller is computed by the function block $\mathcal{R}_L$, that gives -100 for going outside the low-level environment area (a squared region of $150m$), +100 for being close less than 10 meters to the objective of the low-level task ($[140, 75]$ for the straight and $[140, 140]$ for the diagonal), plus the following penalty r_{extra}

for each step:

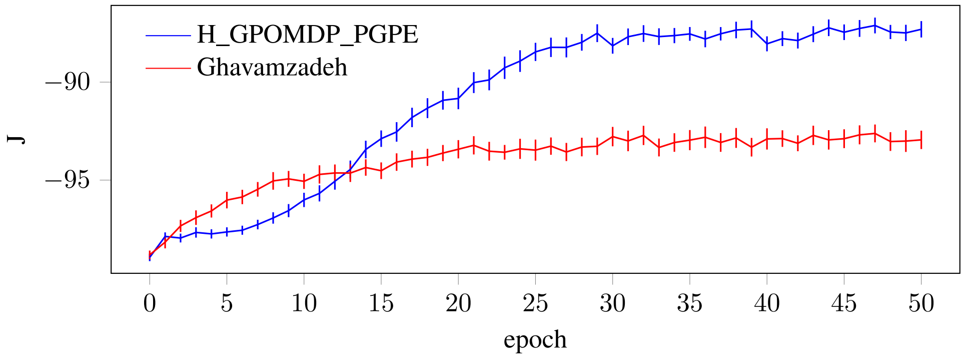
$$r_{\text{extra}} = \exp\left(-\frac{\Delta\theta^2}{(\frac{\pi}{6})^2}\right) - 1,$$

where $\Delta\theta$ is the current heading error of the ship w.r.t. the objective of the low-level task. The low-level episodes terminates when the ship reaches the goal or goes outside the low-level task region.

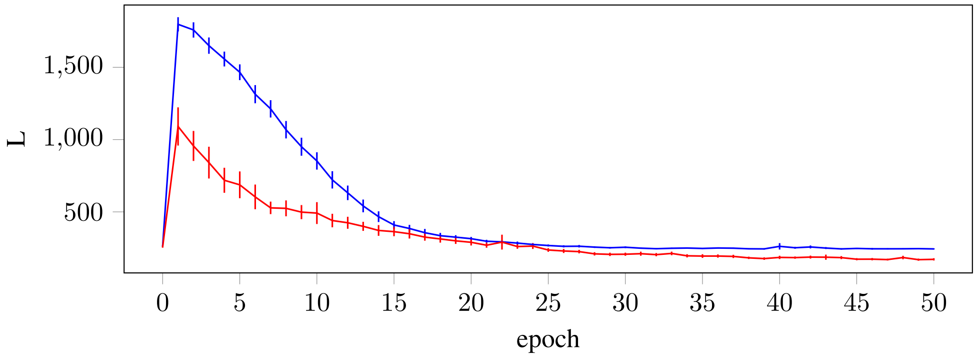
We have used the same Q(λ) algorithm described in [46], with $\gamma = 0.99$, $\lambda = 0.9$ and ϵ-greedy policy. In this implementation of the Ghavamzadeh approach the epsilon parameter is fixed to 0.1 for the first 4 epochs, then decays by 1% every 50 episodes. Instead of using the step based gradient algorithm of Ghavamzadeh, we have used the GPOMDP algorithm, that is able to learn rapidly both the subtasks. We have used a tiling $5 \times 5 \times 10$ as features, and a Gaussian diagonal policy with mean linear in the features and initial standard deviation $\sigma = 0.03$. The learning rate for the GPOMDP algorithm is fixed at 0.08.

The settings for our framework are the same of the small environment experiment. The difference is the initialization of the high-level controller, that is initialized with initial mean $\mu = [500, 500]$ and initial standard deviation $\sigma = [255, 255]$ and the learning rate for the high-level PGPE and low-level GPOMDP algorithms, that are both adaptive learning rates with bounds $\varepsilon = 50$ for the high-level and $\varepsilon = 5 \cdot 10^{-4}$ for the low-level.

The results of this experiment are shown in Fig. 4.4. As it can be seen from Fig. 4.4a, the hierarchical agent learning is slower than the Ghavamzadeh approach in the beginning. This is due to the time spent by the ship moving around the map, while the position of the gate is learned and the optimal proportional gain for the low-level is found. After the position of the gate is found by the high-level controller, the length of the trajectories quickly decreases and the algorithm converges rapidly to good performances. This is due to the fact that the position of the gate is an information that does not depends on the current position of the ship, thus, learning this information results in a policy that generalize to the whole state space, instead of being useful only locally, as the policy learned by the Ghavamzadeh's approach is. From Fig. 4.4b we can see that Ghavamzadeh's system converge to shorter trajectories, and this is due to suboptimal behavior learned by the agents along the borders, where the agent prefers to go outside of the map instead of going towards the center of the environment. This is due to the fact that the Q values must be propagated from the center of the map towards the boundaries, and this propagation is affected negatively by several factors. One issue is the dimensionality of the state space and that most of the trajectory samples are concentrated toward the objec-



(a) Objective function



(b) episode length

Figure 4.4: Learning curves for the ship steering environment

tive, that is in the center of the map. Another problem is the exploration behavior of the ϵ-greedy policy, that may affect the mean length of the paths towards the goal, slowing down the learning process; this can be an issue even in an off-policy setting such as Q(λ), as it may increase the number of samples needed to learn a good trajectory. Finally, the low-level performances at the beginning of the learning process can affect the initial updates of the Q-function. HCGL, instead, learns the general objective, leading to a good generalization in almost any state, which is particularly convenient when starting in new unseen states.

Another important aspect that should be analyzed is the design of the algorithm. Our framework allows to easily design a hierarchical system by defining just the needed function blocks, to exploit expert domain knowledge. The learning algorithm can be any off-the-shelf learning algorithm taken from the state of art, there is no need for custom versions of the algorithm. Thus, the domain experts only need to focus on the structure of the problem and not on the learning algorithms. This is instrumental to bring RL tools to industrial applications. Furthermore, the design tool, the block diagram, is closer to what engineers of other system designers use, as it is inspired by control theory block diagrams. The Ghavamzadeh's approach, instead, cannot be implemented in a generic fashion, but must be re-implemented for any problem instance, in order to exploit domain knowledge. Furthermore, while the Ghavamzadeh's approach looks extremely intuitive from the point of view of RL researchers, which are used to work with options and SMDPs, it is not intuitive from the point of view of a control system. This can be easily seen by trying to implement the Ghavamzadeh's approach in our framework: it is still possible to implement it with minor changes, but it is clear that the resulting design looks overly complicated, particularly if comparing it with HCGL. We believe that, particularly for robotic tasks, our tool is helpful also during the design phase of a hierarchical algorithm, leading to simple and natural design of hierarchical (robotic) agents.

4.3.2 Segway

This problem consists of balancing a 2D Segway and moving it towards a fixed point. This environment is similar to the one described in [87], but we added also the position control w.r.t. the ground, and not only the balancing problem. See Appendix A.4 for details.

The experiment for this environment shows how HCGL can be suitable in the design of classical control applications, and that there are some advantages in using such hierarchical learning approach instead of using black box optimization on a complex non-differentiable equivalent policy.

The control scheme is reported in Fig. 4.5; it is extremely simple, as it matches existing control schemes for similar platforms. The high-level controller $\mathcal{C}_H$ is a simple proportional controller over the linear position that computes the angular setpoint. The function block φ takes in input this setpoint and the current state, returning the state without the linear component and with the error w.r.t. the desired setpoint instead of the actual angle. The low-level controller is another proportional controller over the full state computed by the function block φ. Also in this situation, like in the ship steering environment, we compute the absolute value of the selected parameters before multiplying them to the state. The reward performance of the high-level controller is the reward coming from the environment, while the reward of the low-level controller is the sum of a quadratic cost on diagonal weight ma-

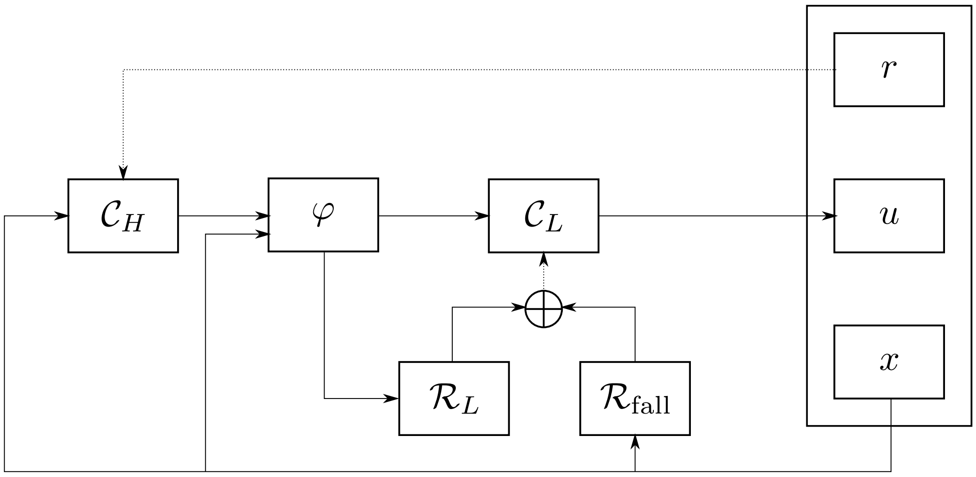


Figure 4.5: Control Graph used in the Segway experiments

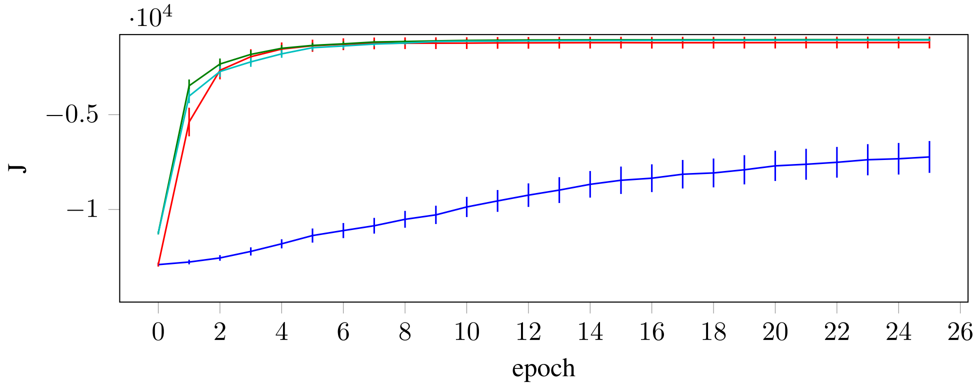
trix $M = \text{diag}([3.0, 0.1, 0.1])$ and a fall punishment of -10000. Notice that in this experiment we do not have alarm signals, thus we do not have any synchronization between low and high-level environment, both controllers run independently at each step of the environment. During learning we keep the high-level controller blocked for the first epoch, since the low-level task is particularly difficult to learn from scratch.

The black box agent uses a parametric non-differentiable policy that is equivalent to the one proposed by our approach.

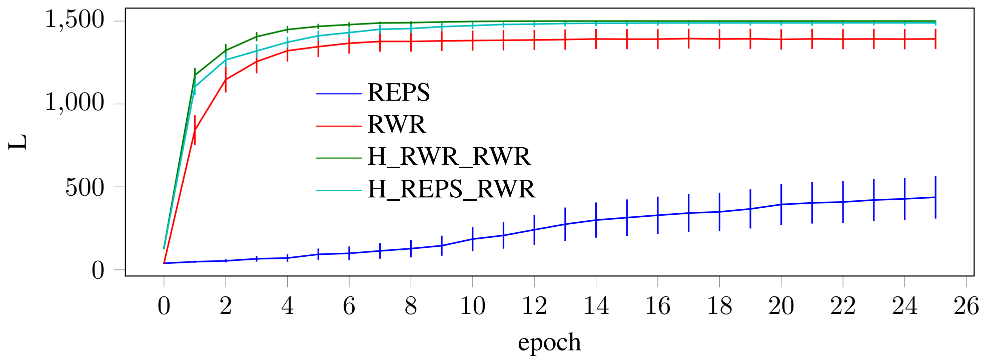
We tested four configurations, two flat algorithms, RWR and REPS, and two hierarchical agents, one with RWR for both low and high-level agent, and one with REPS for the high-level agent and RWR for the low-level one. For HCGL, the learning of the high-level controller starts at the third epoch i.e., the policy for the high level is not updated for the first two epochs.

For every black box algorithm we used a deterministic policy and a diagonal Gaussian distribution as parameter distribution. Flat black box algorithms have a standard deviation $\sigma_i = 2$ for each component. The KL bound for REPS is set to $\epsilon = 5 \cdot 10^{-2}$. The exponential reward transformation temperature for RWR is set to $\beta = 2 \cdot 10^{-3}$. For HCGL, the high-level initial standard deviation is $\sigma_{\text{high}} = 0.02$ and the low-level standard deviation is $\sigma_{\text{low}} = 2$. For the double RWR approach the parameters are $\beta_{\text{low}} = 0.002$ and $\beta_{\text{high}} = 0.01$. The RWR and REPS approach uses $\beta_{\text{high}} = 0.01$ and $\epsilon_{\text{high}} = 0.05$.

Results are shown in Fig. 4.6. By looking at Fig. 4.6a, it is clear that the REPS algorithm is outperformed by the others in this task. This is due to the



(a) Objective Function



(b) Episode length

Figure 4.6: learning curves for the Segway experiment

fact that the updates of the algorithm are bounded by the KL, and between the initial parametrization (that is the null vector) and the optimal one there are many suboptimal parametrizations; this makes extremely difficult to reach a good configuration by bounded updates. The RWR method is able to reach good performances as it rapidly jumps away from the initial parametrization. Our method does the same as the low-level algorithm is RWR, and the low-level stabilization loop causes the stability issues. While the performance of our method is comparable or slightly better than the flat approach, we can see from Fig. 4.6a that the performance of our method is considerably less variant than the performance of the flat RWR algorithm, while the adopted policy is exactly the same. The improved stability of our method can be seen also by looking at Fig. 4.6b, where, keeping in mind that this environment

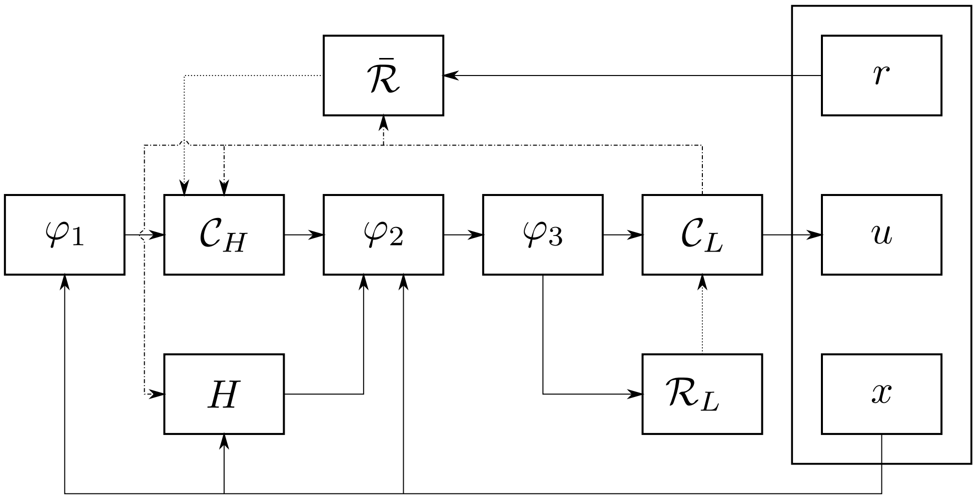


Figure 4.7: Control Graph used to implement the Prey-Predator environment experiment

has no target terminal states, but only failure terminal states, it is clear that our method reaches the horizon more frequently and with less variance w.r.t. the flat RWR algorithm. In this environment, the double RWR approach is able to learn slightly better than the REPS+RWR approach, but the latter is still able to get good performances on this task.

4.3.3 Prey-predator

The Prey-predator environment consists in learning the most efficient way to catch a prey in an environment with obstacles (see Appendix A.5 for details). The objective of this experiment is to show that HCGL scales well also with more complex systems and can be used jointly with Deep RL methods.

The control scheme is reported in Fig. 4.7; it is possible to notice that it is similar to the control graphs already presented for the other problems. The function block φ_1 takes as input the state and feeds to the high-level controller a reduced state with only the position of the prey and the predator. The high-level controller $\mathcal{C}_H$ selects one of the possible actions, that can be either to go in the direction of the target or to move in eight possible directions around the current point. The function block φ_2 transforms the action setpoint into a target position setpoint: if the selected action is to go towards the prey, it returns the current position of the prey, otherwise the target position is computed by adding to the position stored by the hold block H a vector of length $1.0m$ in the selected direction i.e., the targets points are computed around a circle of radius $1m$. The hold block activates synchronously with the high-level controller, so the stored state is the state where the temporal extended action is activated. The low-level controller $\mathcal{C}_L$ learns a policy to drive the predator towards the selected setpoint, its output is the desired linear and angular velocity of the robot. High-level control changes its setpoint only when a low-level environment episode terminates. The low-level environment episode terminates after 10 steps. The high-level controller uses the reward function of the environment as performance metric. The reward between each different decision point is accumulated by the reward accumulator block $\bar{\mathcal{R}}$, which computes the mean reward accumulated during the low-level episodes. This is done to consider the task as an MDP, and not as an SMDP. This makes sense if we want to use off-the-shelf Deep RL algorithms for the high-level, as there is not a wide literature for Deep RL algorithms for SMDPs. Indeed the high-level controller discount factor is accommodated to consider that the next action will be taken after 10 steps. Even if this is an approximation, we believe that is a reasonable one. The low-level reward is provided by the function block $\mathcal{R}_L$, and it is computed as follows:

$$r_{\text{low}} = \cos(\Delta\theta) - \rho$$

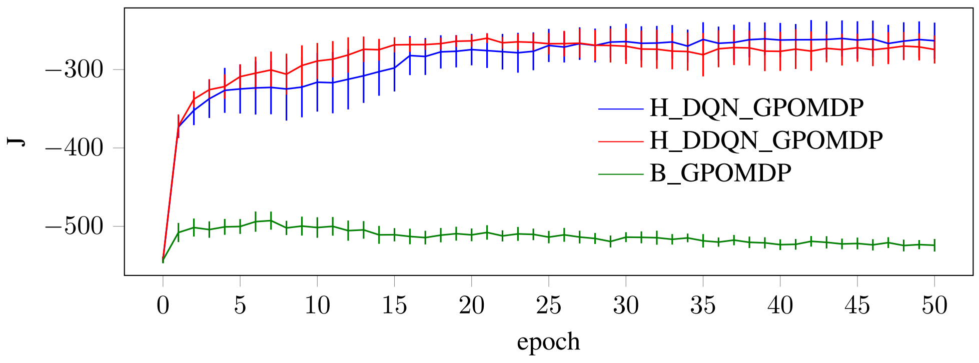
For this experiment, the high-level controller uses the DQN or the DDQN algorithm and a Boltzmann policy. The temperature parameter of the policy decreases as follows:

$$\tau(t) = \frac{1}{\sqrt[5]{t}}$$

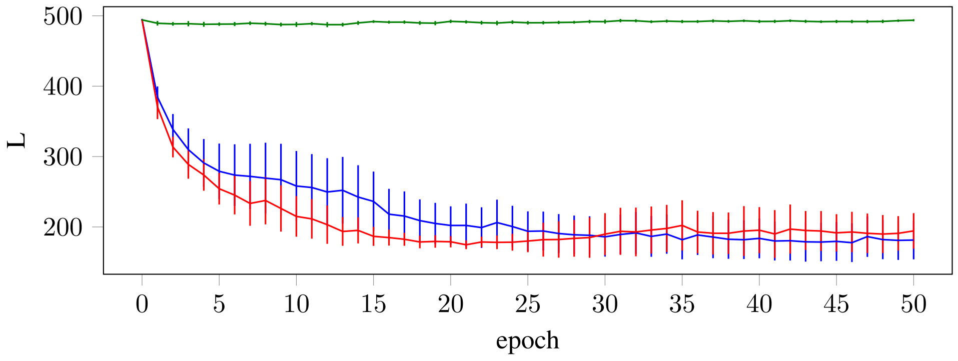
The initial replay memory size is set to 5000, the maximum dimension of the replay memory is set to 100000. The target is updated every 200 steps. The batch size used at every DQN update is 500.

The low-level controller uses the GPOMDP algorithm. The policy is a diagonal Gaussian policy, linear in a Fourier basis features, with maximum frequency $n = 10$. The initial policy parameters distribution is a Gaussian distribution with initial standard deviation $\sigma = 2.5 \cdot 10^{-1}$. For this controller the selected learning rate is the adaptive learning rate with parameter $2 \cdot 10^{-4}$. Parameters are updated after 10 episodes of the low-level controller's induced environment.

To prove that the learning framework works also in this scenario, we compared it with a baseline where the high-level agent always uses the "follow the prey" action at every step. We selected a very general low-level policy to demonstrate that the hierarchical framework is beneficial even in the settings where a complex controller may be needed e.g., because the dynamics of the subsystem are not known. Better results for the low-level controller can be achieved if a specific low-level controller for the differential drive



(a) Objective Function



(b) Episode length

Figure 4.8: learning curves for the Prey-predator experiment

is selected. The high-level controller has been designed with discrete actions because learning a continuous setpoint, in this environment, can be extremely sample-inefficient, as the environment presents strong nonlinearities e.g., due to the presence of physical obstacles. By using discrete actions it is also possible to define the basic action that selects as target the prey itself. Learning this behavior from scratch can be extremely difficult with a generic approximator such as a neural network, without inserting prior knowledge. Results are shown in Fig. 4.8. As it can be easily seen from Fig. 4.8a, both hierarchical agents are able to outperform the baseline agent. Indeed the baseline agent performance degrades along time, this is probably due to the fact that the induced environment is partially observable, and following the prey may cause the predator to get stuck into obstacles. This may lead to unexpected learning behaviors, as the subtasks and the policy do not take in consideration this aspect. As Fig. 4.8b highlights, the performance gain is due not only to the fact that the predator learns to stay closer to the prey, but also to catch the prey sooner as learning goes on.

It is also interesting to compare the behavior of DQN and DDQN. The learning of DDQN seems to be more stable than the one of the DQN approach. Indeed, we can see that the performance of DQN does not increase smoothly at the beginning of the learning. This may be caused by the interaction between the two learning algorithms that is mitigated by the more conservative DDQN updates.

Chapter 5

Inverse Reinforcement Learning

In this chapter we describe a multiple expert, model-free, IRL approach. As already discussed in the introduction, IRL is a powerful tool that can be useful in practical applications in general, but in particular for hierarchical systems, when it is difficult to design a reward function for a sub-task, or when the reward function provided is not sufficiently informative e.g., when it is a step constant cost, or when the reward is sparse i.e., when the only reward value different than zero is either for task success or failure. In this scenarios, IRL can be an alternative method to shape the reward, as the reward function is optimal w.r.t. the desired behavior.

We focus, as in [14], on reward recovery instead of behavior cloning, as learning a reward function is more general: with a reward function it is possible to learn the optimal policy even if the underlying environment dynamics changes w.r.t. the environment used by the experts. This is also useful when a sub-task needs to be solved in different parts of the state space, where the dynamics, and thus the experts' policy may change. Also, sometimes the imitator may not be able to exactly reproduce the behavior of a human demonstrator, but can share the same goals, e.g., a humanoid robot.

To improve both the applicability and the performance of IRL methods, we focus on the multiple expert scenario. We assume that the experts share the same reward function, but their policies may be different due to sub-optimal learning. We call this algorithm Single-Objective Multiple-Expert Inverse Reinforcement Learning (SOME-IRL).

5.1 Algorithm outline

The high-level structure of the SOME-IRL approach is reported in Algorithm 1. The input of the algorithm are the experts' trajectories. Also, a

Algorithm 1 Single Objective-Multiple Experts IRL

```
function SOME_IRL($\mathcal{T} = \{\tau_1, \dots, \tau_N\}$)
    $\widehat{\boldsymbol{\rho}} \leftarrow f_{\text{DM}}(\mathcal{T})$        ▷ Phase 1: distribution matching
    $\widehat{\boldsymbol{\omega}} \leftarrow \arg\min_{\boldsymbol{\omega}} \|\nabla_{\boldsymbol{\rho}} \mathcal{J}(\widehat{\boldsymbol{\rho}}, \boldsymbol{\omega}|\mathcal{T})\|^2$        ▷ Phase 2: gradient minimization
            s.t. constraints
    return $\widehat{\boldsymbol{\omega}}$
```

parametric family of policies for the experts and a parametric distribution must be provided. The algorithm can be divided in two steps. The first step is to find both the policy parameters for each expert and the hyper-parameters of the experts' distribution. Once the hyper-parameters of the density function defined over the experts' policy parameters have been directly estimated using the observed demonstrations, we search for the reward function that better explains the experts' behaviors. To accomplish this, we use an approach inspired by the one proposed in [14] for the single expert case. Since the gradient of the expected return of the optimal policy w.r.t. the policy parameters is zero, the proposal of [14] is to find a reward function that minimizes such gradient. In our case, instead of considering the gradient of the expected return w.r.t. policy parameters, we want to minimize the norm of gradient of the expected return obtained from the experts' policies w.r.t. the hyper-parameters of the density function. The estimation of such gradient is a well-known problem in reinforcement learning literature, since it is the core of the Policy Gradients with Parameter-Based Exploration (PGPE) [24]. PGPE-like methods have been successfully employed in many reinforcement learning problems showing better performance than classical policy gradient approaches thanks to a reduction of the variance of exploration, that allows to perform good gradient estimates even with a limited number of trajectories. This is a key point even for our IRL approach, since it allows to learn good reward functions even with a few demonstrations. Furthermore, since the nullity of the gradient is only a necessary condition for a policy to be optimal, in order to guarantee to find a maximum we constrain the Hessian matrix to be negative definite. Other constraints can be imposed over the reward function parametrization to eliminate undesired degrees of freedom.

5.2 Estimating the Distribution of the Experts' Policies

The gradient minimization faced by SOME-IRL leverages on the assumption that the distribution of the experts' policies $p(\boldsymbol{\theta}|\boldsymbol{\rho})$ is known, which is often not the case in practice. We show that the distribution parameters $\boldsymbol{\rho}$ can be directly recovered from the experts' demonstrations even though the policy

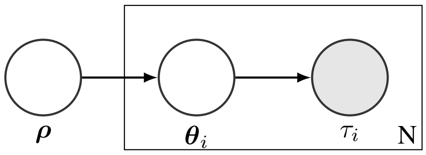


Figure 5.1: Plate notation for the Maximum likelihood for distribution matching.

parameters $\boldsymbol{\theta}$ are never observed. We formulate this problem as a *distribution matching* problem with hidden variables and we describe two solution methods.

Given a family of parametric policies describing the experts' behaviors, we estimate from the available demonstrations the parameters $\boldsymbol{\rho}$ of the probability density function $p(\boldsymbol{\theta}|\boldsymbol{\rho})$ representing the distribution of the experts' policies. We discuss both the naïve Maximum Likelihood Estimation (MLE) approach, using a simple hierarchical model, and a more complex Bayesian approach. Both formulations exploit hidden variables to represent the unobserved policy parameters.

5.2.1 Maximum Likelihood Estimation

The computation of the maximum likelihood solution is straightforward when we consider the simple hierarchical model reported in Fig. 5.2 (dashed red box). Given the policy parameters, each trajectory τ_i is independent from the parameters $\boldsymbol{\rho}$ leading to the following likelihood formulation:

$$\mathcal{L} = p(\tau_1, \ldots, \tau_N|\boldsymbol{\rho}, \boldsymbol{\theta}) = \prod_{i=1}^{N} p(\tau_i|\boldsymbol{\rho}, \boldsymbol{\theta}_i) = \prod_{i=1}^{N} p(\tau_i|\boldsymbol{\theta}_i).$$

The worst case scenario is obtained when each trajectory is generated by a different policy parametrization, thus leading to a number of policy parameters to be estimated equal to the number of demonstrated trajectories. As a consequence, the maximum likelihood estimate reduces to a simple two-step procedure. First, for each trajectory compute the maximum likelihood for each expert's policy parameters, then compute the maximum likelihood distribution w.r.t. the estimated policy parameters.

MLE can be done in closed form, if the policy is a multivariate Gaussian with state-dependent mean that linearly depends on state features. Assuming fixed covariance matrix, we first write the log likelihood and we compute the

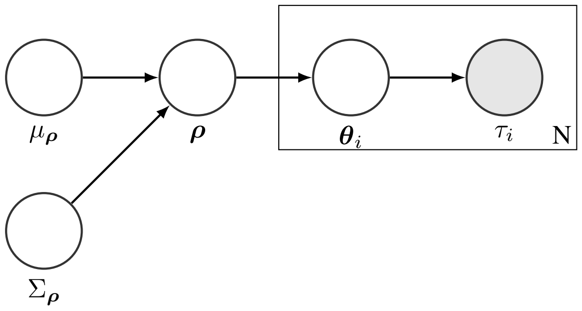


Figure 5.2: Plate notation for the Maximum a posteriori model for distribution matching.

derivative:

$$\log p(\tau|\boldsymbol{\theta}) = -\frac{1}{2}\sum_{t=1}^{T_\tau} \left(\varphi(x_t)^T\boldsymbol{\theta} - u_t\right)^T \Sigma^{-1} \left(\varphi(x_t)^T\boldsymbol{\theta} - u_t\right) + k,$$

$$\begin{aligned}\frac{\partial}{\partial\boldsymbol{\theta}}\log p(\boldsymbol{\theta}|\tau) &= -\sum_{t=1}^{T_\tau}\left(\varphi(x_t)^T\boldsymbol{\theta} - u_t\right)^T\Sigma^{-1}\varphi(x_t)^T \\ &= -\boldsymbol{\theta}^T\sum_{t=1}^{T_\tau}\varphi(x_t)\Sigma^{-1}\varphi(x_t)^T + \sum_{t=1}^{T_\tau}u_t^T\Sigma^{-1}\varphi(x_t)^T.\end{aligned}$$

By forcing the derivative to be the null vector and solving for $\boldsymbol{\theta}$ we obtain:

$$\boldsymbol{\theta}_{\text{MLE}} = \left(\sum_t^\tau \varphi(x_t)\Sigma^{-1}\varphi(x_t)^T\right)^{-1}\left(\sum_t^\tau \varphi(x_t)\Sigma^{-1}u_t\right).$$

Then, the distribution can be estimated in the canonical way.

5.2.2 Maximum A Posteriori

By adding a prior to the distribution parameters $\boldsymbol{\rho}$ we obtain the hierarchical Bayesian model reported in Fig. 5.2. In this scenario it is not possible to exploit the two-step algorithm since the estimate of the parameters no longer depends only on the expert's policy.

In the MAP framework, the parameters $\boldsymbol{\rho}$ and $\boldsymbol{\theta}$ are obtained by maximizing the log posterior that is given by:

$$\begin{aligned}\log p(\boldsymbol{\rho}, \boldsymbol{\theta}|\tau_1, \dots, \tau_n) = &\log p(\boldsymbol{\rho}) + \sum_{i=1}^{N} \log p(\tau_i|\boldsymbol{\theta}_i) \\ &+ \sum_{i=1}^{N} \log p(\boldsymbol{\theta}_i|\boldsymbol{\rho}) + C.\end{aligned} \tag{5.1}$$

In the general case, this posterior cannot be exactly optimized, but we can resort to approximated methods, like Monte Carlo Markov Chain or variational inference [88]. However, we can derive a more efficient algorithm for particular cases. Consider the following property:

Property 1. *If the likelihood of trajectories w.r.t. parameters $\boldsymbol{\theta}$, $\boldsymbol{\rho}$ is Gaussian and we use a Normal-Wishart prior, Eq.* (5.1) *is a continuous and differentiable multi-convex function w.r.t. $\boldsymbol{\theta}$, $\boldsymbol{\rho}$.*

Proof. The MAP posterior probability can be computed as:

$$\begin{aligned}p(\boldsymbol{\rho}, \boldsymbol{\theta}|\tau_1, \dots, \tau_N) &\propto p(\tau_1, \dots, \tau_N|\boldsymbol{\rho}, \boldsymbol{\theta})p(\boldsymbol{\rho}, \boldsymbol{\theta}) \\ &= \prod_{j=1}^{N} p(\tau_j|\boldsymbol{\rho}, \boldsymbol{\theta})p(\boldsymbol{\theta}|\boldsymbol{\rho})p(\boldsymbol{\rho}) \\ &= p(\boldsymbol{\rho}) \prod_{j=1}^{N} p(\tau_j|\boldsymbol{\rho}, \boldsymbol{\theta}_j) \prod_{i=1}^{N} p(\boldsymbol{\theta}_i|\boldsymbol{\rho}) \\ &= p(\boldsymbol{\rho}) \prod_{i=1}^{N} p(\tau_i|\boldsymbol{\theta}_i)p(\boldsymbol{\theta}_i|\boldsymbol{\rho}).\end{aligned}$$

Computing the logarithm of the posterior we obtain:

$$\log p(\boldsymbol{\rho}, \boldsymbol{\theta}|\tau_1, \dots, \tau_n) = \log p(\boldsymbol{\rho}) + \sum_{i=1}^{N} \log p(\tau_i|\boldsymbol{\theta}_i) + \sum_{i=1}^{N} \log p(\boldsymbol{\theta}_i|\boldsymbol{\rho}) + C.$$

Notice that this function is non convex in general. If a Normal-Wishart prior [88] is used for $\boldsymbol{\rho}$, then it follows that:

- if the mean and the precision matrix of the distribution are fixed, then the log posterior is a sum of positive definite quadratic functions w.r.t. the policy parameters.
- if the precision matrix of the distribution and the policy parameters are fixed, then the log posterior is a sum of positive definite quadratic functions w.r.t. the mean of the distribution.

- if the mean of the distribution and the policy parameters are fixed, then the log posterior is a sum of linear terms plus the logarithm of the Wishart probability density function, which is a convex function w.r.t. the precision matrix of the distribution.

Then, the function is multi-convex. It is also trivial to prove that the function is continuous and differentiable in the whole domain of the policy distribution and the policy parameters. □

Accordingly to Property 1, it is possible to use block coordinate ascent [89] to attain the *global* optimum of the posterior in Eq. (5.1).

This approach, reported in Algorithm 2, is iterative, but can exploit closed-form parameter updates. First, the fitting of $\boldsymbol{\theta}$ is solved via MAP estimate by considering as fixed both the precision matrix and the mean of the distribution. To derive the closed form, first we write the log posterior and compute its derivative:

$$\begin{aligned}\log p(\boldsymbol{\theta}|\tau) &= \log p(\tau|\boldsymbol{\theta}) + \log p(\boldsymbol{\theta})\\ &= -\frac{1}{2}\sum_{t=1}^{T_\tau}\left(\varphi(x_t)^T\boldsymbol{\theta} - u_t\right)^T \Sigma^{-1}\left(\varphi(x_t)^T\boldsymbol{\theta} - u_t\right)\\ &\quad -\frac{1}{2}\left(\boldsymbol{\theta} - \mu_\rho\right)^T \Sigma_\rho^{-1}\left(\boldsymbol{\theta} - \mu_\rho\right) + k,\end{aligned}$$

Algorithm 2 Coordinate ascend MAP Algorithm

```
function CA_MAP($\mu_{\boldsymbol{\rho}_0}, \Sigma_{\boldsymbol{\rho}_0}, \mathcal{T} = \{\tau_1, \dots, \tau_N\}$)
    $t \leftarrow 0$
    $\boldsymbol{\rho}_0 \leftarrow [\mu_{\boldsymbol{\rho}_0}, \Sigma_{\boldsymbol{\rho}_0}]$        ▷ Initialize the parameters $\boldsymbol{\rho}$ to the prior mode
    repeat
        $t \leftarrow t + 1$
        for $i \leftarrow 1, N$ do
            $\boldsymbol{\theta}_{i,t} \leftarrow \arg\max \log p(\boldsymbol{\theta}_i, \tau_i)$        ▷ Optimize $\boldsymbol{\theta}_i$ with fixed $\boldsymbol{\rho}$
        $\boldsymbol{\rho}_t \leftarrow \arg\max p(\boldsymbol{\rho}|\boldsymbol{\theta})$        ▷ Optimize $\boldsymbol{\rho}$ with fixed $\boldsymbol{\theta}$
    until $p(\boldsymbol{\rho}_t, \boldsymbol{\theta}_t|\mathcal{T}) \approx p(\boldsymbol{\rho}_{t-1}, \boldsymbol{\theta}_{t-1}|\mathcal{T})$
    return $\theta, \rho$
```

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{\theta}} \log p(\boldsymbol{\theta}|\tau) =& \sum_{t=1}^{T_\tau} \left(u_t - \varphi(x_t)^T \boldsymbol{\theta}\right)^T \Sigma^{-1} \varphi(x_t)^T + (\mu_\rho - \boldsymbol{\theta})^T \Sigma_\rho^{-1} \\ =& - \boldsymbol{\theta}^T \sum_{t=1}^{T_\tau} \varphi(x_t) \Sigma^{-1} \varphi(x_t)^T \\ & - \sum_{t=1}^{T_\tau} u_t^T \Sigma^{-1} \varphi(x_t)^T - \boldsymbol{\theta}^T \Sigma_\rho^{-1} + \mu_\rho^T \Sigma_\rho^{-1}.\end{aligned}$$

Forcing the derivative to be the null vector and solving for $\boldsymbol{\theta}$ we obtain:

$$\boldsymbol{\theta}_{\text{MAP}} = \left(\sum_t^\tau \varphi(x_t) \Sigma^{-1} \varphi(x_t)^T + \Sigma_{\boldsymbol{\rho}}^{-1} \right)^{-1} \left(\sum_t^\tau \varphi(x_t) \Sigma^{-1} u_t + \Sigma_{\boldsymbol{\rho}}^{-1} \mu_{\boldsymbol{\rho}} \right).$$

The other updates to perform are a Gaussian MAP mean estimation with known precision matrix, and a Wishart MAP estimation with known mean, w.r.t. the $\boldsymbol{\theta}$ parameters. Note that this algorithm can be naturally parallelized, since the policy parameters $\boldsymbol{\theta}_i$ can be independently estimated.

As the MAP approach is jointly estimating the parameters of the policies under a common prior, this method is more robust and less prone to overfitting than the MLE approach, especially if the experts' trajectories explore only a small subspace of the state-action space.

5.2.3 Selecting the Appropriate Representation

The considered probabilistic inference approaches (MLE and MAP) perform well when the chosen policy representation sufficiently matches the experts' one. In real-world applications, the experts' policies are only partially known. For instance, we can guess which are the features exploited by an expert to draw his decision, but we are not certain that such information is complete or even used by the expert. Usually, a richer space is created by combining a set of basic features.

Unfortunately, when using a policy representation that is highly expressive, i.e., it has more degrees of freedom than those actually needed to represent the expert's policy, the recovery of the rewards may fail. The main reason for this behavior is that the "useless" parameters may have a very low variance, resulting in a very noisy gradient estimate. Fortunately, the property mentioned here below allows the use of dimensionality reduction techniques to select the most appropriate representation.

Property 2. *If the distribution $p(\boldsymbol{\theta}|\boldsymbol{\rho})$ is Gaussian, SOME-IRL is invariant to affine transformations.*

Proof. From the definition of the policy gradient estimate:

$$\nabla_{\boldsymbol{\rho}}\mathcal{J}(\boldsymbol{\rho},\boldsymbol{\omega}) = \int_{\Theta}\int_{\mathbb{T}} p(\tau|\boldsymbol{\theta})p(\boldsymbol{\theta}|\boldsymbol{\rho})\nabla_{\boldsymbol{\rho}}\log p(\boldsymbol{\theta}|\boldsymbol{\rho})J_{\boldsymbol{\omega}}(\tau)\mathrm{d}\tau\mathrm{d}\boldsymbol{\theta},$$

we can notice that an affine transformation, only affects the $\nabla_{\boldsymbol{\rho}}\log p(\boldsymbol{\theta}|\boldsymbol{\rho})$ component. Thus, it is sufficient to show that $\nabla_{\boldsymbol{\rho}}\log p(\boldsymbol{\theta}|\boldsymbol{\rho})$ is invariant to the desired transformations.

Let K and b denote the matrices involved in the linear transformation, where K is a matrix that incorporates the rotation and the stretch and b denotes the translation. Let $p = \mathcal{N}(\boldsymbol{\rho},\Sigma)$ be the original distribution whose gradient is:

$$\nabla_{\boldsymbol{\rho}}\log p(\boldsymbol{\theta}|\boldsymbol{\rho}) = \Sigma^{-1}(\boldsymbol{\theta}-\boldsymbol{\rho}).$$

By applying the mentioned affine transformation to the distribution parameters $\boldsymbol{\rho}$ and Σ, we obtain a new distribution $p_{\text{aff}} = \mathcal{N}\left(K\boldsymbol{\rho}-b, K\Sigma K^T\right)$ with gradient:

$$\begin{aligned}\nabla_{\boldsymbol{\rho}}\log p_{\text{aff}}(K\boldsymbol{\theta}-b|K\boldsymbol{\rho}-b) =& K^T K^{-T}\Sigma^{-1}K^{-1}(K\boldsymbol{\theta}-b-(K\boldsymbol{\rho}-b))\\ =&\Sigma^{-1}K^{-1}K(\boldsymbol{\theta}-\boldsymbol{\rho}) = \Sigma^{-1}(\boldsymbol{\theta}-\boldsymbol{\rho})\\ =&\nabla_{\boldsymbol{\rho}}\log p(\boldsymbol{\theta}|\boldsymbol{\rho}).\end{aligned}$$

Note also that the evaluation point $\boldsymbol{\theta}$ is transformed according to the affine transformation. □

Given Property 2, it is possible to use Principal Component Analysis (PCA) [88] as a pre-processing step to remove the useless degrees of freedom by projecting the parameters of each policy into a smaller, but sufficiently expressive subspace. This pre-processing step significantly increases the performance of the algorithm.

5.3 Learning the reward function

In order to recover the reward $\mathcal{R}$, we follow a gradient minimization approach similar to the one used by GIRL, [14]:

$$\begin{aligned}\boldsymbol{\omega}^* &= \arg\min_{\boldsymbol{\omega}} \|\nabla_{\boldsymbol{\rho}}\mathcal{J}(\boldsymbol{\rho},\boldsymbol{\omega})\|_2^2\\ &= \arg\min_{\boldsymbol{\omega}} \left\|\int_{\Theta}\int_{\mathbb{T}} p(\tau|\boldsymbol{\theta})p(\boldsymbol{\theta}|\boldsymbol{\rho})\nabla_{\boldsymbol{\rho}}\log p(\boldsymbol{\theta}|\boldsymbol{\rho})J_{\boldsymbol{\omega}}(\tau)\mathrm{d}\tau\mathrm{d}\boldsymbol{\theta}\right\|_2^2. \quad (5.2)\end{aligned}$$

GIRL method leverages on the policy gradient formulation in order to recover the reward function that makes the observed expert to behave optimally. Similarly, our approach seeks for the reward parameter vector $\boldsymbol{\omega}^*$ that makes *all* the experts (nearly) optimal w.r.t. the *same* objective. It exploits the optimization framework defined by the PGPE [24] in order to model the distribution of the experts' policies. The resulting SOME-IRL algorithm, can be seen as a generalization of GIRL where instead of minimizing the policy gradient, we search to minimize the PGPE gradient i.e., the gradient of the policy distribution described in Eq. (5.2). The advantages of PGPE w.r.t. the policy gradient are:

1. the possibility to consider also deterministic and non-differentiable policies;
2. a significant reduction in the variance of the gradient estimates, thus requiring less samples and iterations to attain good performances.

As we will see in the experimental section, such advantages are inherited by the SOME-IRL approach. However, the nullity of the gradient is only a necessary condition for the optimality and, in practice, may be poorly informative. Differently from GIRL, we propose and show how to extend the minimization problem in Eq. (5.2) to consider the constraint that the Hessian of $\mathcal{J}(\boldsymbol{\rho}, \boldsymbol{\omega})$ is negative definite.

5.3.1 Distribution gradient minimization

Using the approaches described in the previous section, we are able to estimate the experts' distribution parameters $\boldsymbol{\rho}$ and the policy parameters $\{\boldsymbol{\theta}_i\}_{i=1}^N$ associated to the N trajectories. Given these three elements $(\boldsymbol{\rho}, \boldsymbol{\theta}, \{\tau_i\}_{i=1}^N)$, we can empirically estimate the gradient (minimized in Eq. (5.2)) of the expected return w.r.t. the distribution parameters:

$$\nabla_{\boldsymbol{\rho}} \widehat{\mathcal{J}}(\boldsymbol{\rho}, \boldsymbol{\omega}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\rho}} \log p(\boldsymbol{\theta}_i | \boldsymbol{\rho}) \left(J_{\boldsymbol{\omega}}(\tau_i) - b \right), \tag{5.3}$$

where b is an arbitrary baseline that can be chosen in order to minimize the variance of the gradient estimate, see [90, 91]. In order to recover the reward function optimized by the experts, we search for the reward function parameters $\hat{\boldsymbol{\omega}}$ that minimize the norm of the estimated gradient in Eq. (5.3):

$$\hat{\boldsymbol{\omega}} = \arg\min_{\boldsymbol{\omega}} \left\| \nabla_{\boldsymbol{\rho}} \widehat{\mathcal{J}}(\boldsymbol{\rho}, \boldsymbol{\omega}) \right\|_2^2 . \tag{5.4}$$

One of the key properties of SOME-IRL, that is inherited from GIRL, is that the objective function in Eq. (5.4) is *convex* whenever the parametric reward model is linear w.r.t. $\boldsymbol{\omega}$.

Proof. Let $\Phi(\boldsymbol{\theta})$ be the feature expectations under policy $\pi_{\boldsymbol{\theta}}$:

$$\Phi(\boldsymbol{\theta}) = \frac{1}{N} \sum_{\tau \sim \pi_{\boldsymbol{\theta}_i}} \sum_{t=1}^{T_\tau} \phi(x_t, u_t, x_{t+1}).$$

When linear parametrization of the reward is used, the optimal baseline can be expressed as:

$$b_{\boldsymbol{\omega}} = b^T \boldsymbol{\omega},$$

where b is the matrix of the optimal baselines w.r.t. the single features. The optimal PGPE baseline of each single feature can be computed as shown in [26]. The objective function is:

$$\begin{aligned}
\left\| \nabla_{\boldsymbol{\rho}} \widehat{\mathcal{J}}(\boldsymbol{\rho}, \boldsymbol{\omega}) \right\|_2^2 &= \nabla_{\boldsymbol{\rho}} \widehat{\mathcal{J}}(\boldsymbol{\rho}, \boldsymbol{\omega})^T \nabla_{\boldsymbol{\rho}} \widehat{\mathcal{J}}(\boldsymbol{\rho}, \boldsymbol{\omega}) \\
&= \left(\sum_{i=1}^{N} \nabla_{\boldsymbol{\rho}} \log p(\boldsymbol{\theta}_i|\boldsymbol{\rho}) \left(\Phi(\boldsymbol{\theta}_i) - b\right)^T \boldsymbol{\omega} \right)^T \left(\sum_{i=1}^{N} \nabla_{\boldsymbol{\rho}} \log p(\boldsymbol{\theta}_i|\boldsymbol{\rho}) \left(\Phi(\boldsymbol{\theta}_i) - b\right)^T \boldsymbol{\omega} \right) \\
&= \boldsymbol{\omega}^T \left(\sum_{i=1}^{N} \left(\Phi(\boldsymbol{\theta}_i) - b\right) \nabla_{\boldsymbol{\rho}} \log p(\boldsymbol{\theta}_i|\boldsymbol{\rho})^T \right) \left(\sum_{i=1}^{N} \nabla_{\boldsymbol{\rho}} \log p(\boldsymbol{\theta}_i|\boldsymbol{\rho}) \left(\Phi(\boldsymbol{\theta}_i) - b\right)^T \right) \boldsymbol{\omega} \\
&= \boldsymbol{\omega}^T \, \nabla\Phi^T \nabla\Phi \, \boldsymbol{\omega},
\end{aligned}$$

that is convex, as it is a quadratic dyadic form. □

The optimization reduces to a quadratic programming problem as the matrix $\nabla\Phi = \sum_{i=1}^{N} \left(\Phi(\boldsymbol{\theta}_i) - b\right) \nabla_{\boldsymbol{\rho}} \log p(\boldsymbol{\theta}_i|\boldsymbol{\rho})^T$ needs to be calculated only once.

5.4 Optimization Constraints

Estimating only the gradient magnitude is not always sufficient to extract a good reward function. We discuss in this section the additional constraints that are needed by the optimization algorithm.

5.4.1 Expert distribution constraints

When dealing with MDPs, the optimal distribution of policies expert may coincide with a distribution that gives probability mass only to the optimal policy parametrization. This may not be always the case, because even if the optimal policy of an MDP is always deterministic, the policy distribution may be over a policy parametrization such that a subset of policies may be equivalent. However, in practical scenarios e.g., reasonable parametrizations and Gaussian experts' distribution, this can be an issue. It is important to constrain the policy distribution in order to exclude unwanted distributions from the considered distribution space. This can be easily done by a reparametrization of the distribution. The most effective and simple reparametrization for Gaussian expert distributions is to consider the variance of the distribution fixed, thus removing all the parameters related to the variance from the optimization problem, obtaining a gradient that only depends on the mean value.

5.4.2 Simplex Constraint

Although the formulation in Eq. (5.4) is sound, in practice it is useful to constrain the shape of the reward function on a bounded set or to chose a representation that is scale-invariant. Particularly, when the reward function is linear we must introduce a constraint in order to avoid the ambiguity problem [10], e.g., by constraining the weights in the simplex (see [14] for an extensive analysis). Despite the need of an additional constraint, the linear reward representation is interesting since it can be efficiently solved and can be theoretically analyzed in order to derive guarantees on the recovered weights (as done in [14]).

5.4.3 Second Order Constraint

In general, the nullity of the gradient is only a necessary condition for optimality. If the expected return of the expert's policy is concave (or log-concave) w.r.t. the policy parameters, then the gradient-minimization approach is sound, as all the stationary points of the expected return w.r.t. the reward function parameters are global optima. However, in many situations, it is possible to have a reward function that has multiple stationary points. The problem is that these points may not be exclusively maxima, but they may be also saddle points or local minima. Such points are unstable points in the learning process and the behavior induced by the retrieved reward function diverges from the one demonstrated by the experts.

Negative Definite Constraint

In order to discard local minima and saddle points, we need to add a second order constraint. Specifically, the Hessian ($\mathcal{H}_\rho$) of the expected return w.r.t. the reward parameters must be (strictly) negative definite. This is the same as requiring that the maximum eigenvalue (λ_i) of the Hessian is negative.

Given an estimate of the Hessian [92]:

$$\begin{aligned}\mathcal{H}_{\boldsymbol{\rho}}\widehat{\mathcal{J}}(\boldsymbol{\rho},\boldsymbol{\omega}) =& \frac{1}{N}\sum_{i=1}^{N}\big(\nabla_{\boldsymbol{\rho}}\log p(\boldsymbol{\theta}_i|\boldsymbol{\rho})\nabla_{\boldsymbol{\rho}}\log p(\boldsymbol{\theta}_i|\boldsymbol{\rho})^T \\ &+\mathcal{H}_{\boldsymbol{\rho}}\log p(\boldsymbol{\theta}_i|\boldsymbol{\rho})\big)\left(J_{\boldsymbol{\omega}}(\boldsymbol{\theta}_i)-b_{\boldsymbol{\omega}}\right),\end{aligned} \tag{5.5}$$

where $b_{\boldsymbol{\omega}}$ is a baseline that depends on $\boldsymbol{\omega}$, the negative definite constraint is given by:

$$\max\lambda(\mathcal{H}_{\boldsymbol{\rho}}\widehat{\mathcal{J}}(\boldsymbol{\rho},\boldsymbol{\omega})) = \max_{\nu|\nu^T\nu=1}\nu^T\mathcal{H}_{\boldsymbol{\rho}}\widehat{\mathcal{J}}(\boldsymbol{\rho},\boldsymbol{\omega})\nu < 0. \tag{5.6}$$

As seen before, the linear reward function induces nice properties in the problem. In the linear case, the fact that the Hessian matrix is linearly dependent on the parameter vector $\boldsymbol{\omega}$ induces a convex constraint. Formally, such constraint is equivalent to the maximum of a set of linear constraints.

In practical cases, the constraint in Eq. (5.6) may not be sufficient as the Hessian estimate is noisy. Instead of requiring that the largest eigenvalue is strictly less than zero, we introduce a soft threshold ϵ (positive and small enough). The value of such threshold is problem-dependent, and should be selected such that the true Hessian is negative definite with high probability.

Local Curvature Heuristic

Since an accurate estimate of the eigenvalues of the Hessian matrix usually requires a large amount of samples, we define the following alternative constraint:

$$\nabla_{\boldsymbol{\rho}}\widehat{\mathcal{J}}(\boldsymbol{\rho},\boldsymbol{\omega})^T\mathcal{H}_{\boldsymbol{\rho}}\widehat{\mathcal{J}}(\boldsymbol{\rho},\boldsymbol{\omega})\nabla_{\boldsymbol{\rho}}\widehat{\mathcal{J}}(\boldsymbol{\rho},\boldsymbol{\omega}) \leq K, \quad K<0. \tag{5.7}$$

This constraint is equivalent to requiring that the curvature of the objective function in the direction of the gradient is negative. When a small amount of data is available, the evaluation of the constraint in Eq. (5.7) is more reliable than the evaluation of constraint Eq. (5.6). On the other hand, the constraint in Eq. (5.7) (differently from the constraint in Eq. (5.6)) does not represent a sufficient condition for local maxima. Another drawback of this constraint is that it is a cubic, non-convex function of the reward parameters $\boldsymbol{\omega}$. However, in practice, the optimization under this constraint does not pose particular issues.

5.5 Expectation-maximization approach

We propose an alternative method to gradient minimization in order to retrieve the reward function parameters, that does not require Hessian constraints. The main idea of the gradient-based inverse reinforcement learning algorithms is to find a policy that is optimal by exploiting the fact that the optimal policy is a policy for which the gradient of policy parameters w.r.t. the reward function is the null vector. Thus, we can say that the (locally) optimal parametrization can be seen as an attractor for any policy gradient algorithm. Policy Gradient algorithms are not the only policy search methods that converge to optimal policies or distributions. Expectation Maximization (EM) methods are methods that exploit the EM update, and they converge to a locally optimal parametrization. We can exploit the same idea of policy gradient IRL methods in the EM framework. The basic idea of this approach is to minimize the "distance" between the original expert distribution and the distribution after the EM update.

The first issue to face is the selection of an appropriate metric. When considering the gradient it is straightforward to use the L2 norm i.e., the gradient magnitude, as metric. This metric may not be reasonable to measure the distance between two distributions. We propose the KL divergence as (pseudo) metric to be minimized. The reward weights are retrieved by solving the following optimization problem:

$$\boldsymbol{\omega}^* = \arg\min \mathrm{KL}(\mathcal{D}_{\boldsymbol{\rho}} \parallel \mathcal{D}_{\hat{\boldsymbol{\rho}}(\boldsymbol{\omega})}), \tag{5.8}$$

where $\mathcal{D}_{\boldsymbol{\rho}}$ is the expert distribution with probability density function $p(\boldsymbol{\theta}|\boldsymbol{\rho})$, and $\mathcal{D}_{\hat{\boldsymbol{\rho}}(\boldsymbol{\omega})}$ is the same distribution after the EM update, given that the reward function is parametrized by $\boldsymbol{\omega}$.

We propose an instance of this method where the EM update is the update performed by the RWR [26] algorithm. This update is a weighted maximum likelihood estimate, where the samples from the experts distribution are weighted using an exponential transformation of the trajectories return. We define the weights d_i for each sample $\boldsymbol{\theta}_i$ of the reward function as:

$$d_i = e^{J_{\boldsymbol{\omega}}(\boldsymbol{\theta}_i) - \max\limits_k J_{\boldsymbol{\omega}}(\boldsymbol{\theta}_k)}, \tag{5.9}$$

where $J_{\boldsymbol{\omega}}(\boldsymbol{\theta}_i)$ is the return of the i-th expert under the reward function parametrized by $\boldsymbol{\omega}$. We call the vector of weights for every policy as d.

A practical implementation of this algorithm can be obtained when considering a Gaussian expert distribution with mean μ and variance Σ. We make the assumption that the expert covariance matrix is fixed under the EM update i.e., the distribution parameters $\boldsymbol{\rho}$ is just the mean vector μ. This constraint on the distribution update is due to the same reasons highlighted for the SOME-IRL algorithm. Under these assumptions, the minimization problem in Eq. (5.8) can be written as:

$$\boldsymbol{\omega}^* = \arg\min(\hat{\mu}(\boldsymbol{\omega}) - \mu)^T \Sigma^{-1} (\hat{\mu}(\boldsymbol{\omega}) - \mu), \tag{5.10}$$

where $\hat{\mu}(\boldsymbol{\omega})$ is the mean vector of the distribution after the EM update:

$$\hat{\mu}(\boldsymbol{\omega}) = \boldsymbol{\theta} \frac{d(\boldsymbol{\omega})}{\| d(\boldsymbol{\omega}) \|_1}, \tag{5.11}$$

Eq. (5.10) shows that, when using a Gaussian experts distribution, the resulting optimization problem is a norm minimization problem, with the inverse of the covariance matrix Σ as metric. When Σ is a multiple of the identity matrix, this problem reduces to the L2-norm minimization of the difference between the original mean vector and the updated mean vector.

It can be easily seen, by looking at Eqs. (5.10) and (5.11), that the resulting function to be minimized is a continuous and differentiable function w.r.t. $\boldsymbol{\omega}$ (notice that the max operation in Eq. (5.9), that is not differentiable, is factored out in the division). However, it is not linear w.r.t. $\boldsymbol{\omega}$. This is a major issue of this approach, as the optimization may become impractical for large dataset and complex reward function parametrizations. Notice that, differently from the gradient-based approach, there is no practical way to reduce the optimization problem to a quadratic programming constrained problem, even when the reward function is linearly parametrized. However, at least for simple problems, the optimization is not problematic, and the reward function retrieved is close to the one obtained by the gradient approach, without using the Hessian constraints.

This approach may appear similar to the Relative Entropy Inverse Reinforcement Learning proposed in [11], however there are major differences, both practical and conceptual, between the two. This approach can be used to replace the gradient approach described above, and thus faces the multiple expert problem, instead of the single expert one. Another major difference is that the Relative Entropy approach is computing an approximate gradient of the exact objective function, while we use an exact gradient (or any other optimization technique) on an approximate objective function. This is a major difference as our approach does not need further samples from the environment, thus being fully batch. In our framework the objective function is approximate as we suppose known the policy distribution, while this assumption does not always hold and the distribution must be estimated using the techniques described above.

5.6 Experimental results

We evaluate the approach in three different domains: the Linear Quadratic Regulator (LQR) domain [14], the 2-dimensional Non Linear System (NLS) defined in [93] and the Ship Steering problem [46]. All the experiments consider 100 runs for each setting.

5.6.1 LQR

This environment corresponds to the task of solving the optimal control for a linear time invariant discrete system. We consider the same experimental setting as in [14]. We used this environment with 2 and 3 states variables. The task is to find the optimal parameters for the linear control rule under a quadratic cost function. In this experiment we will consider the reward function of this environment, described in Appendix A.1, to be linearly parametrized with parameter vector $\boldsymbol{\omega}$. This can be expressed in terms of the R and Q matrices as follows:

$$R(\boldsymbol{\omega}) = \sum_k R_k \boldsymbol{\omega}_k, \qquad Q(\boldsymbol{\omega}) = \sum_k Q_k \boldsymbol{\omega}_k,$$

where the (i,j)-th component of matrices R_k and Q_k are:

$$R_k|_{ij} = \begin{cases} 0.1 & i=j=k \\ 0.9 & i=j \\ 0 & \textit{otherwise} \end{cases}, \quad Q_k|_{ij} = \begin{cases} 0.9 & i=j=k \\ 0.1 & i=j\neq k \\ 0 & \textit{otherwise} \end{cases}.$$

In this experiment, the parametric reward function contains the experts' reward function. The expert parametrization is indicated as $\boldsymbol{\omega}^*$.

The distribution of the experts' policies is a normal distribution whose mean is the optimal linear policy parameters, that can be found by solving the associated Riccati equation, and diagonal covariance matrix, with $1e-3$ variance for each parameter.

When the distribution of the policy parameters $p(\boldsymbol{\theta}|\boldsymbol{\rho})$ is known, the SOME-IRL approach outperforms the GIRL algorithm, as can be seen in Figures 5.3 and 5.4 (note different scale on y-axes).[1] This is mainly due to the low variance of the PGPE gradient estimates.

When the distribution and the policy parameters are not known, they must be estimated: either the faster MLE approach, or the MAP approach, more robust when the experts' policies are stochastic, can be used. In Figures 5.5 and 5.6 is shown the performance of the two methods, tested with stochastic expert policies where the actions are sampled from Gaussian distributions with standard deviation of 0.01. The performance of the MAP method is,

[1]The double lines in some of the plots are used to break the y-axis in order to improve visualization when there are some points that are out of scale.

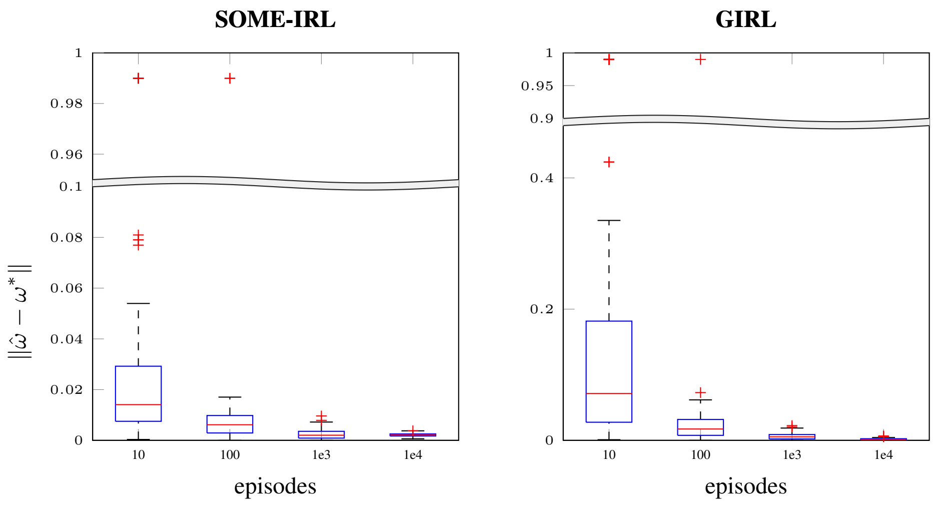


Figure 5.3: 2D LQR. Error of the reward parameters found by SOME-IRL and by GIRL with exact policy.

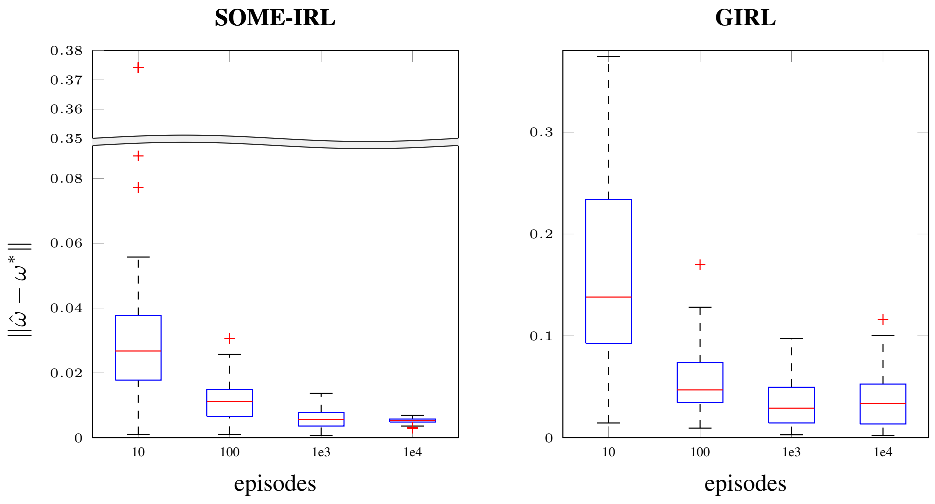


Figure 5.4: 3D LQR. Error of the reward parameters found by SOME-IRL and by GIRL with exact policy.

in this scenario, better than that of MLE, even when the number of trajectories grows. The reason for this is that the MAP method takes into account the inter-policy similarity to avoid overfitting when the experts' policies are stochastic.

We also investigated the performance of the SOME-IRL algorithm when the policy model is not known: in this scenario, when the considered para-

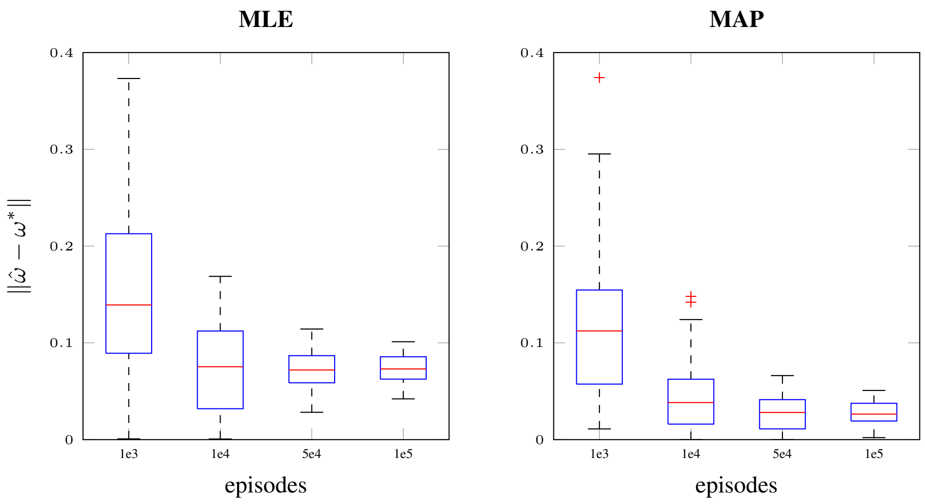


Figure 5.5: 2D LQR. Error of the reward parameters found by SOME-IRL with MAP and MLE estimation of ρ.

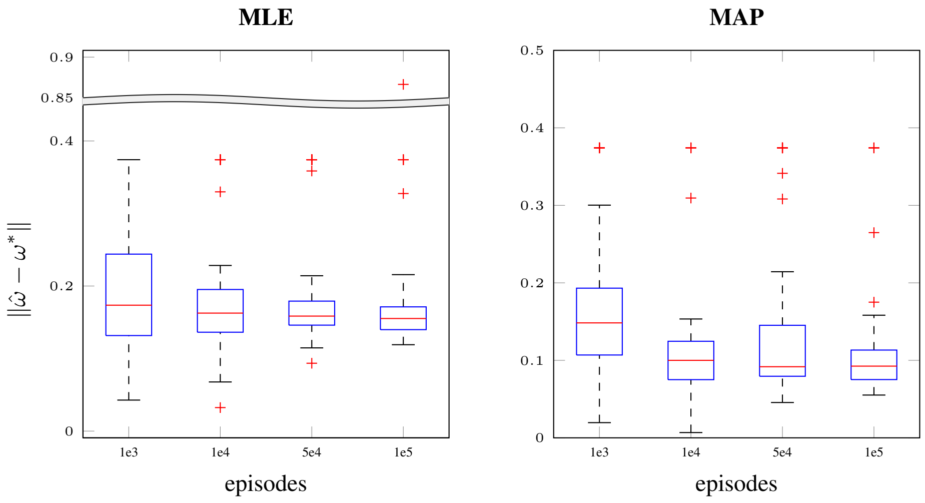


Figure 5.6: 3D LQR. Error of the reward parameters found by SOME-IRL with MAP and MLE estimation of ρ.

metric policy model is not sufficiently expressive, the algorithm does not obtain good results. When the policy model is sufficiently expressive, then SOME-IRL achieves good performance with enough samples and with PCA post-processing of distributions. However, when the policy becomes too expressive, the performance of the algorithm degrades significantly, as it can be noticed in Fig. 5.7 (right). This is not a major drawback, as the quality

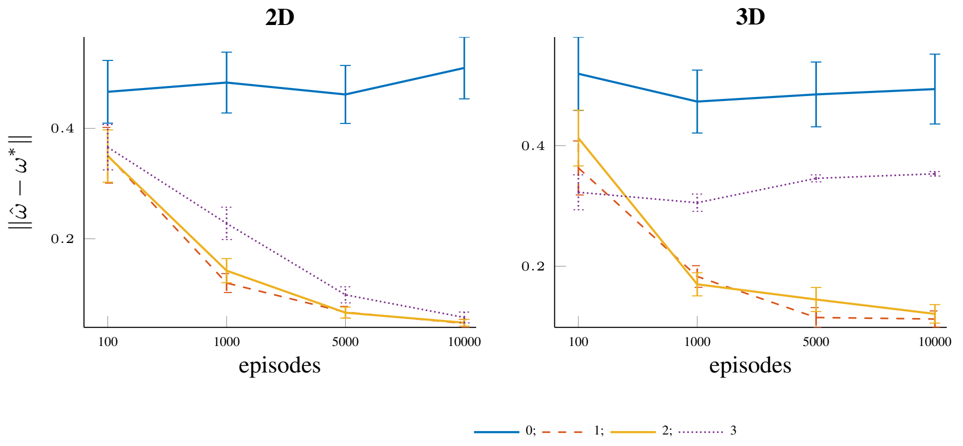


Figure 5.7: LQR. Performance of SOME-IRL with MLE and policies approximated using polynomials of different degrees (from 0 to 3).

of the reward function is correlated with the magnitude of the estimated gradient that can be quickly computed offline. In practice, the batch nature of SOME-IRL allows to perform an optimization of the meta parameters, i.e., it is possible to test several policy approximators and choose the best one.

5.6.2 Non Linear System

This problem is similar to the LQR problem, but with a more complex dynamics and reward function. NLS is the nonlinear system described in [93]. See Appendix A.2 for further details.

The features of the reward function are 25 (5×5) Gaussian RBFs distributed evenly in the range $[-3, 3]^2$. The real reward function cannot be exactly represented by this approximator, however the nearest reward function is the one that gives maximum weight to the central feature.

The distribution of the experts' policies is a normal distribution of linear policies with parameters $\boldsymbol{\theta} = [6.5178, -2.5994]^T$ and identity covariance matrix with 0.1 variance for each parameter. This sub-optimal policy was obtained via REINFORCE [26].

In this domain, the simple minimization of the gradient is not enough, because the solution with minimum gradient corresponds to a saddle point, thus the feature expectations of the policy learned with the recovered reward function diverge w.r.t. the ones of the observed experts.

We can see in Fig. 5.8 that the best performance is achieved by imposing the Hessian to be negative definite, but it has higher variance due to the dif-

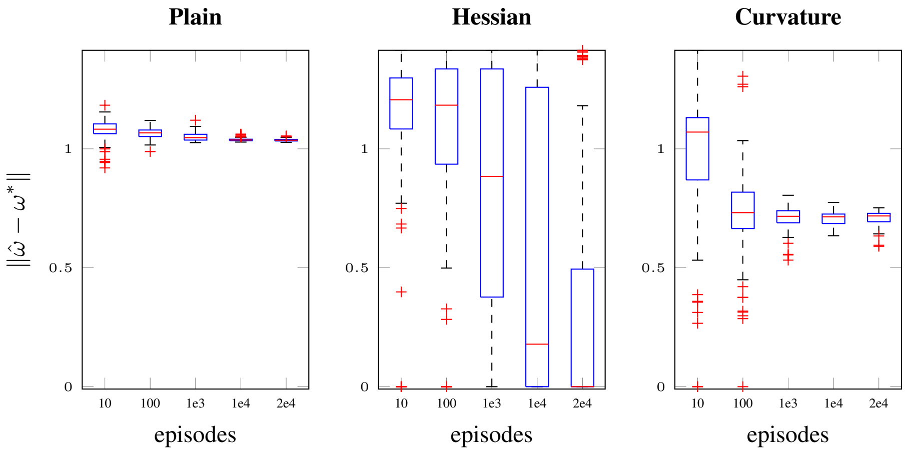


Figure 5.8: NLS. Error of the reward parameters found by different versions of SOME-IRL.

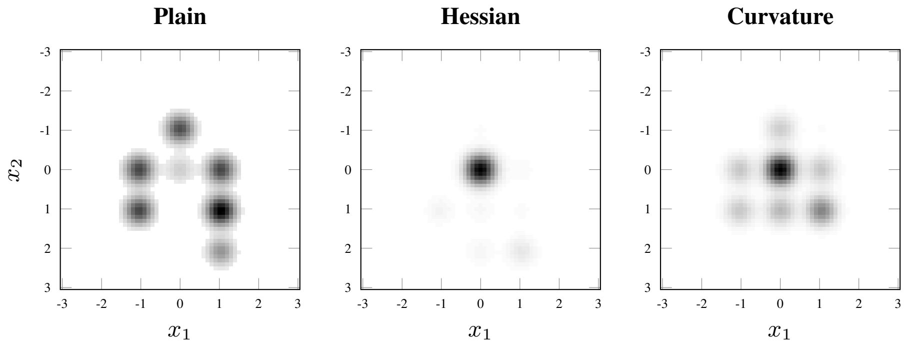


Figure 5.9: NLS. Mean reward functions after 20000 trajectories.

ficulty of the Hessian estimation. The curvature constraint recovers a biased reward function, as it can be seen in Fig. 5.9, but the estimate is less variant, and the feature expectations match comparably with the exact solution, as shown in Fig. 5.10. Notice that the GIRL algorithm cannot recover a good reward function, as it does not consider any second order constraint to enforce the maximum condition. An implementation of the Hessian constraint in the GIRL setting is not feasible, as the large variance of the estimate of the Hessian makes the number of needed samples intractable. Refer to Fig. 5.11 for examples of recovered trajectories.

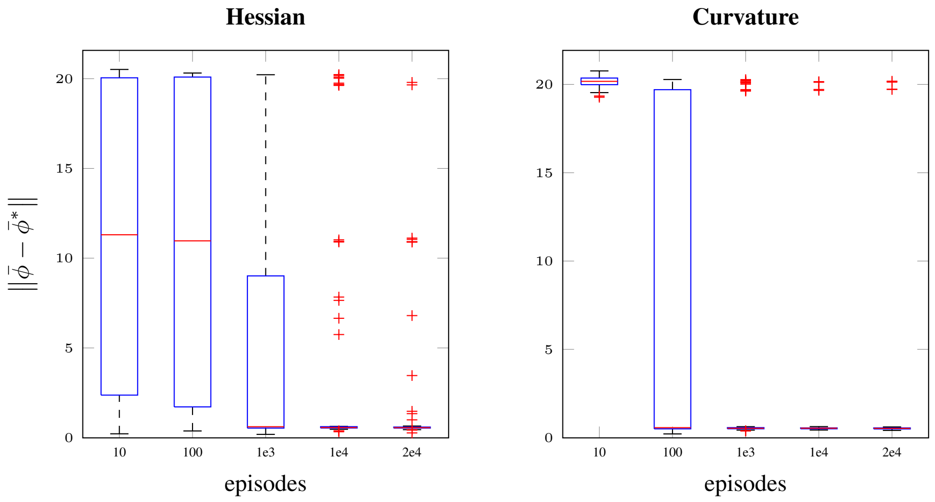


Figure 5.10: NLS. Error in the features expectations of different versions of SOME-IRL.

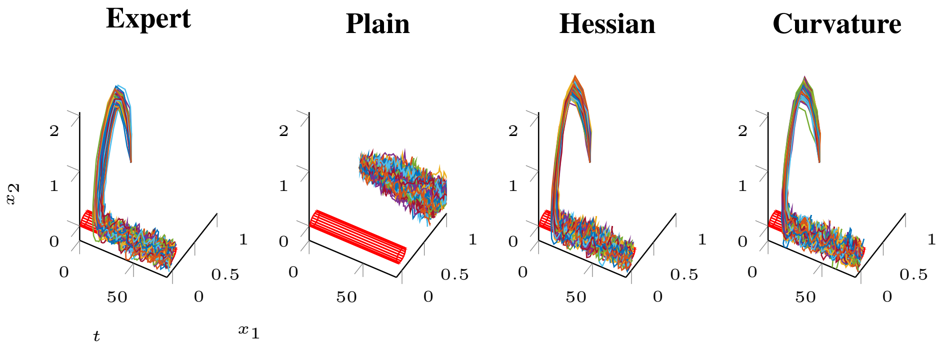


Figure 5.11: NLS trajectories, the goal region is the red cylinder.

5.6.3 Ship Steering

This problem consists of driving a ship through a gate, without going outside a defined region (see [46] and Appendix A.3 for details). For these experiments we have used the difficult version of the environment i.e., the version with control action computed at every step.

The approximate reward function is a linear reward function that uses as features a set of 400 (20×20) tiles in the two spatial dimensions. The exploited policy is a linear policy w.r.t. a grid of 108 $(3 \times 3 \times 6 \times 2)$ Gaussian RBFs with 25% of overlapping. The distribution of the experts' policies is a normal distribution with diagonal covariance matrix with 0.1 variance for each parameter and mean represented by the sub-optimal parameters learned

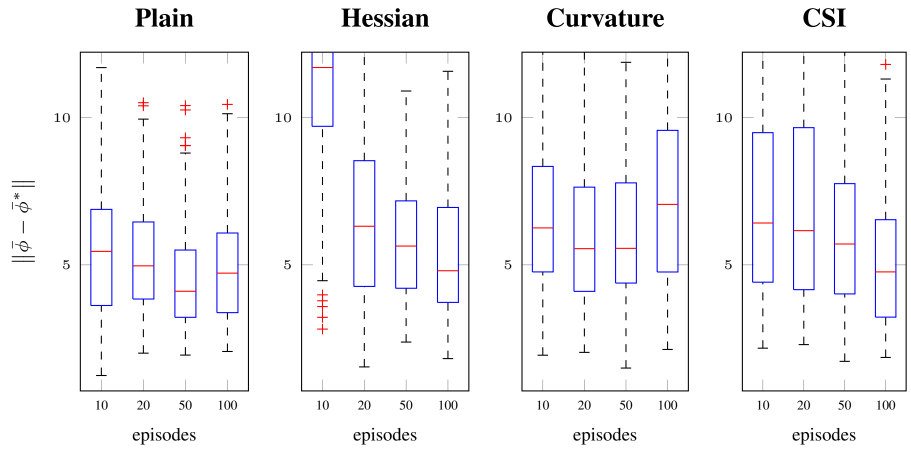


Figure 5.12: Ship Steering. Error in the features expectations of different versions of SOME-IRL and CSI.

via REINFORCE. The matching of the features expectation is calculated from an expert trained with REINFORCE, using the same policy family of the experts on the recovered reward function, and then compared to the features expectations of the experts.

This last domain is designed to test the proposed algorithms against two batch IRL algorithms, Structured Classification-based Inverse Reinforcement Learning (SCIRL) [12] and Cascaded Supervised Inverse Reinforcement Learning (CSI) [13], on a more complex problem. Since CSI and SCIRL are not able to deal with continuous actions, we have used 10 uniformly spaced actions bins. The experts' policies are modeled as a Gaussian distribution around the parameters of the policy found using REINFORCE and variance $0.1\, I$.

Fig. 5.12 shows that SOME-IRL variants perform comparably to CSI, and, if a sufficient number of samples is given, SOME-IRL with the Hessian constraint performs similarly to CSI. However, we have empirically noticed that SOME-IRL requires much less computational resources (both time and memory) than CSI, since the CSI algorithm works with matrices that depends on the number of steps performed, while our approach is episode based. This is particularly true if the dimensionality of the action space increases, due to the "curse of dimensionality". We have not reported the performance of SCIRL since it was completely unable to match the feature expectation of the expert. We guess that the issue of SCIRL in this setting is the heuristic used by the algorithm.

We performed another experiment on the ship steering domain to highlight the advantages of the curvature heuristic when the Hessian estimate is difficult. In this experiment a 20×20 RBFs grid is considered. Examples of trajectories recovered by SOME-IRL using RBFs are reported in Fig. 5.13. We can clearly see that omitting the Hessian constraint, it is possible that the obtained behavior diverges w.r.t. experts' demonstration. The trajectories obtained by adding the Hessian constraint are better, but they fail to describe the real objective of the environment. This can be due to an inaccurate eigenvelues estimate. Curvature heuristic, while showing demonstration that does not match exaclty the expert ones, probably due to the bias added in the optimization that makes impossible to obtain zero gradient solutions, is able to capture more closely the objective of the environment.

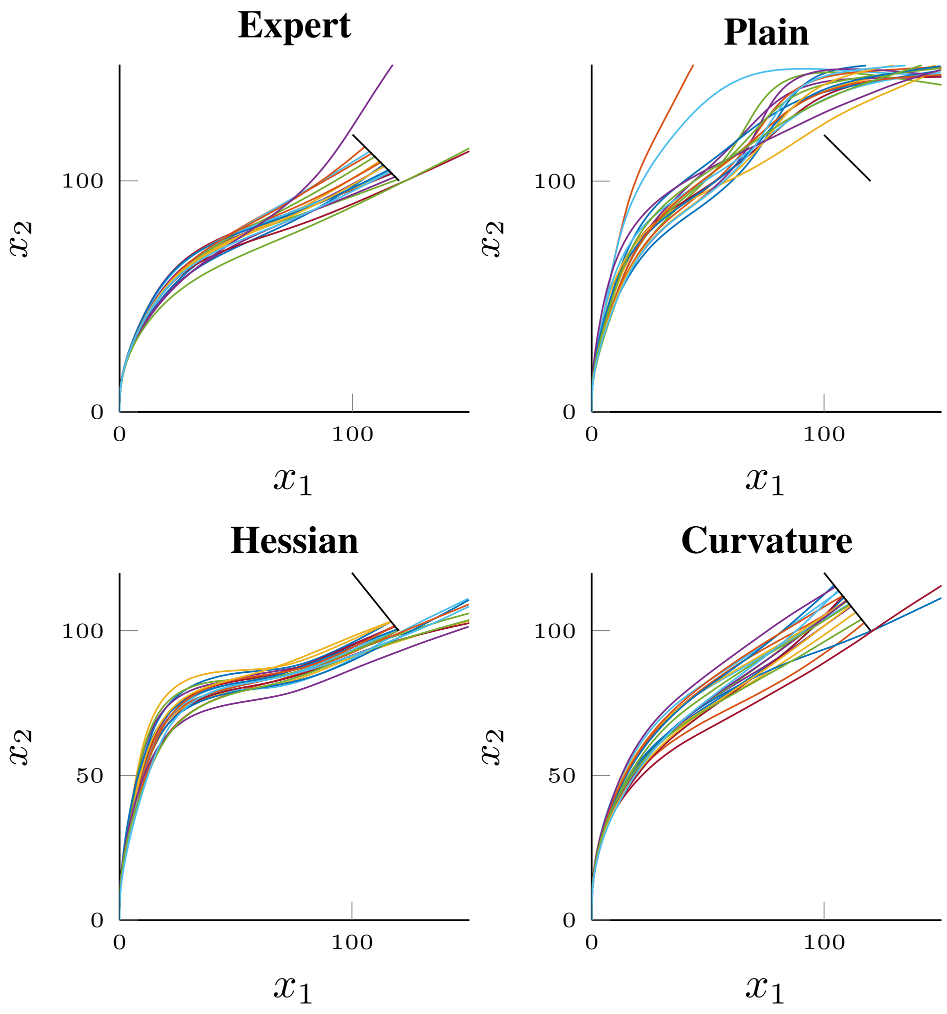


Figure 5.13: Ship steering trajectories, with RBFs as reward features.

Chapter 6

Conclusions

We have presented a set of techniques to face one of the most important issues in Reinforcement Learning: the design of hierarchical agents. Hierarchical agents are an extremely important approach to reduce the gap between RL research and practical industrial applications.

While the model-free RL algorithms are now able to learn moderately complex tasks without any prior knowledge, for real world environments these algorithms are still not applicable, as they require a lot of interactions to learn. The exploitation of prior knowledge, both by manually inserting it in the system, or by inferring it automatically from the demonstration, is one of the key methods to face complex tasks, and the literature often lacks on this aspect as, at least in recent years, the focus has shifted towards completely automatic learning systems. These systems, mostly based on neural approximators, are really appealing, as their objective is to design agents that are able to learn tasks without any human intervention and are having an outstanding success for solving simulated tasks e.g., electronic and board games [31, 94, 95]. However, while the algorithms are improving year by year, we are still far from learning algorithms that can run without careful preprocessing and parameter tuning. We believe that it is not reasonable, at the present time, to run one of this algorithms in most real world scenarios: not only the amount of data required to learn is not compatible with real world applications, but also the instability of the learning process, the possibility of undesired behavior in unexplored regions of the state space and the amount of preprocessing and implementation details to make the learning work, can have catastrophic consequences.

While in this work we do not provide a complete and definitive solution to any of the problems highlighted in the literature and above, we have put the focus on one of the possible ways towards the application of RL in the real world: the structure of the agent.

In the first part of the thesis we discussed the structure of the agent, in particular in the framework of HRL, and what are the key issues raised by its design. We believe that the design and the implementation of an agent are tightly coupled, and it is important to provide a framework in which it is natural and easy to design a complex control structure. For this reason, we have presented HCGL, a novel HRL framework, inspired by the block diagram used in control theory, that not only provides an easy way to specify the design of the hierarchical agent, but also represents directly both the policy and the learning structure. Our idea is strongly opposed to the idea of procedure, that is one of the fundamental ideas of classical HRL, in most of the previous frameworks. While procedures and macros are one of the fundamental building blocks of information technology and AI, they might not be the best framework for control systems that live in a continuous world, where states cannot be easily described by an abstract, high-level representation, at least at the actuator level. We propose a decentralized learning framework, where each subsystem is learned independently, as opposed to the centralized learning view of classical HRL systems.

The centralized view that characterizes in particular the option framework and the intra-option learning, has major advantages, as it can coordinate different parts of the systems and subtasks towards the goal. However, in most branches of engineering it has been shown that centralized approaches do not scale well with the complexity of the system to be controlled. Decentralized approaches may give rise to problems due to the complex interaction between different subsystems and learning algorithms, but they have major advantages both during the design and the implementation of the systems. With a decentralized approach it is possible to decouple different parts of the problem, and use the best solution for each component. Decentralized approaches are easier to train, as it is possible to train each part of the subsystem separately or initialize the subsystem policy using prior knowledge e.g., using control theory and models of the environment to initialize the parameters and then let the system to automatically tune them in a model-free fashion, by overcoming issues related to poor model and parameter identification. Decentralized approaches make easier to analyze both the learning and the behavior of isolated subsystems, thus it is easier to spot which are the problematic parts of the agent structure. For this reason, the concept of induced environment i.e., the environment seen by a single subsystem, is important to study the properties and the design of the proposed distributed learning algorithm.

An important part of HRL is the definition of the reward function each subtask should optimize. Every RL system must optimize the expected discounted return computed from the extrinsic reward function that measures how well a given agent behaves in the considered environment. However, intrinsic reward functions are one of the most important parts of HRL systems as they are useful to characterize a given subtask. It is possible to learn temporally extended actions and sub-policies even without any intrinsic reward functions, as done in some recent works [57], however it is impossible to impose or suggest a specified behavior without specifying this performance metric. For this reason, the computation of reward signals is a key part of HCGL, and in our framework they are considered as an important part of the control system: autonomous systems should be able to compute the intrinsic reward signals using just the information available from the measures of their sensors, and this is the main reason why the computation of intrinsic reward functions is tightly coupled with the interaction of the control system with the environment.

As the design of the reward function is one of the key elements of hierarchical agents, in the second part of this work we focused on Inverse Reinforcement Learning, with the main objective of retrieving the reward function, instead of learning the expert policy from demonstrations. With IRL, it is possible to automatically encode the expert knowledge about a task, or a subtask, into a reward function that specifies what is the objective the expert is trying to maximize. This property is particularly useful when we need to specify a subtask. The SOME-IRL approach we developed takes inspiration from previous works presented in the literature, extending the gradient based method to the framework of multiple experts. The reason for this extension is twofold: it is a more reasonable setting w.r.t. the stochastic expert framework presented in the original GIRL algorithm, and it is more sample efficient. The latter feature of SOME-IRL, allows to estimate more accurately second order constraints, that were missing in the original algorithm. Without these constraints it is possible to obtain reward functions that the expert is trying to minimize instead of maximize. This extremely undesirable behavior cannot happen in HCGL, at least if a sufficient number of samples is used to estimate the hessian constraint. We argue that the multiple, suboptimal, deterministic, expert model is more reasonable than the optimal stochastic expert one, as human demonstration are rarely optimal, and often humans, while not behaving always consistently, do provide nor good neither sufficient exploration in order to compute a stochastic gradient accurately. We believe, instead, that is reasonable that the expert distribution is concentrated around the optimal policy. In this work, we assume that the expert distribution is defined by the parameters distribution: this is a strong assumption, that however could hold in general, by defining an appropriate parameter space and an appropriate distribution.

The ideas proposed in this thesis could be extended in various directions. It would be interesting to study the interaction between different RL algorithms in the distributed learning scenario we have presented. We believe that, by assuming some properties of each induced environment, it is possible to derive some theoretical properties about the convergence of the described system, and for distributed learning algorithm in general. This analysis could be interesting also in other frameworks, such as the multi-agent RL.

Another interesting aspect of the HCGL framework that should be analyzed is the off-policy learning scenario, where multiple controllers produce an action, but only part of them is actually considered by the rest of the system. Currently, HCGL does not support this type of learning, but a further extension, that considers if the signals are effectively applied at each time step, can be developed, and the implications of this model on off-policy learning can be analyzed. The work on IRL can be easily extended to consider automatic features construction either by using deep neural networks or a method similar to the one proposed in [79].

It would be also interesting to study empirically the effect of IRL in the HRL framework, particularly when we want to transfer the knowledge of a known subtask from a toy, limited, scenario, into a more realistic one: an objective can be specified as a target set of states, by a sparse reward function, in an easy-to-solve environment, and then the learned reward function, extracted with IRL techniques, could be transferred to learn a dense reward function in a more complex scenario. In the setting described above, the agent's policy is known, thus the behavior cloning phase is not needed. This makes the techniques described in this work more applicable, together with other related algorithms. Another possible extension for the SOME-IRL algorithm is to consider the multiple objective scenario. This scenario could possibly be modeled by a multimodal expert distribution, where each mode is optimal w.r.t. a different mode of the objective function resulting from the reward function and the environment transition model. The performance of SOME-IRL is not completely clear in this scenario, however, it is possible, whenever each distribution mode is described by different parameters e.g., when considering Gaussian mixture models, to compute a different reward function for every mode. However, the properties of the extension to the general case, in particular when we want to extract a single reward function for all the experts, are difficult to highlight and more investigation is needed.

Bibliography

[1] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*, pages 303–314. ACM, 2018.

[2] Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. Robust physical-world attacks on machine learning models. *arXiv preprint arXiv:1707.08945*, 2017.

[3] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[4] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *European conference on computer vision*, pages 404–417. Springer, 2006.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

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

[8] OpenAI. Learning dexterous in-hand manipulation. *arXiv preprint arXiv:1808.00177*, 2018.

[9] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *AAAI*, pages 3207–3214. Phoenix, AZ, 2018.

[10] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proc. ICML*. ACM, 2004.

[11] Abdeslam Boularias, Jens Kober, and Jan Peters. Relative entropy inverse reinforcement learning. In *Proc. AISTATS*, pages 182–189, 2011.

[12] Edouard Klein, Matthieu Geist, Bilal Piot, and Olivier Pietquin. Inverse reinforcement learning through structured classification. In *Proc. NIPS*, pages 1007–1015, 2012.

[13] Edouard Klein, Bilal Piot, Matthieu Geist, and Olivier Pietquin. A cascaded supervised learning approach to inverse reinforcement learning. In *Machine Learning and Knowledge Discovery in Databases*, pages 1–16. Springer, 2013.

[14] Matteo Pirotta and Marcello Restelli. Inverse reinforcement learning through policy gradient minimization. In *Proc. AAAI*, pages 1993–1999, 2016.

[15] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[16] Jonathan Baxter and Peter L. Bartlett. Direct gradient-based reinforcement learning: I. gradient estimation algorithms. 1999.

[17] Peter L. Bartlett and Jonathan Baxter. Infinite-horizon policy-gradient estimation. *J. Artif. Intell. Res.*, 15:319–350, 2001.

[18] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[19] Jan Reinhard Peters. *Machine learning of motor skills for robotics*. PhD thesis, University of Southern California, 2007.

[20] Thomas Degris, Patrick M Pilarski, and Richard S Sutton. Model-free reinforcement learning with continuous action in practice. In *American Control Conference (ACC), 2012*, pages 2177–2182. IEEE, 2012.

[21] Thomas Degris, Martha White, and Richard S Sutton. Off-policy actor-critic. *arXiv preprint arXiv:1205.4839*, 2012.

[22] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014.

[23] Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomput.*, 71(7-9):1180–1190, March 2008.

[24] Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559, 2010.

[25] Daan Wierstra, Tom Schaul, Jan Peters, and Juergen Schmidhuber. Natural evolution strategies. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 3381–3387. IEEE, 2008.

[26] Marc Peter Deisenroth, Gerhard Neumann, and Jan Peters. A survey on policy search for robotics. *Foundations and Trends in Robotics*, 2(1-2):1–142, 2013.

[27] Jens Kober and Jan R Peters. Policy search for motor primitives in robotics. In *Advances in neural information processing systems*, pages 849–856, 2009.

[28] Jan Peters, Katharina Mülling, and Yasemin Altun. Relative entropy policy search. In *AAAI*, pages 1607–1612. Atlanta, 2010.

[29] Christian Daniel, Gerhard Neumann, and Jan Peters. Hierarchical relative entropy policy search. In *Artificial Intelligence and Statistics*, pages 273–281, 2012.

[30] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[31] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[32] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 2, page 5. Phoenix, AZ, 2016.

[33] Hado V Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621, 2010.

[34] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped dqn. In *Advances in neural information processing systems*, pages 4026–4034, 2016.

[35] Oron Anschel, Nir Baram, and Nahum Shimkin. Averaged-dqn: Variance reduction and stabilization for deep reinforcement learning. In *International Conference on Machine Learning*, pages 176–185, 2017.

[36] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[37] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *International Conference in Learning Representations (ICLR)*, 2016.

[38] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[39] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[40] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[41] Andrew G Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13(1-2):41–77, 2003.

[42] Ronald Parr and Stuart J. Russell. Reinforcement learning with hierarchies of machines. In *NIPS*, 1997.

[43] Thomas G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *J. Artif. Intell. Res.*, 13:227–303, 2000.

[44] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artif. Intell.*, 112:181–211, 1999.

[45] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Intra-option learning about temporally abstract actions. In *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, pages 556–564, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[46] Mohammad Ghavamzadeh and Sridhar Mahadevan. Hierarchical policy gradient algorithms. In *Proc. ICML*, pages 226–233. AAAI Press, 2003.

[47] Amy McGovern and Andrew G Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. 2001.

[48] Özgür Şimşek and Andrew G Barto. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 95. ACM, 2004.

[49] Özgür Şimşek, Alicia P Wolfe, and Andrew G Barto. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proceedings of the 22nd international conference on Machine learning*, pages 816–823. ACM, 2005.

[50] David Jardim, Luís Nunes, and Sancho Oliveira. Hierarchical reinforcement learning: learning sub-goals and state-abstraction. In *Information Systems and Technologies (CISTI), 2011 6th Iberian Conference on*, pages 1–4. IEEE, 2011.

[51] Sander G van Dijk and Daniel Polani. Grounding subgoals in information transitions. In *Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), 2011 IEEE Symposium on*, pages 105–111. IEEE, 2011.

[52] Bram Bakker and Jürgen Schmidhuber. Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In *Proc. of the 8-th Conf. on Intelligent Autonomous Systems*, pages 438–445, 2004.

[53] Scott Niekum and Andrew G Barto. Clustering via dirichlet process mixture models for portable skill discovery. In *Advances in neural information processing systems*, pages 1818–1826, 2011.

[54] George Konidaris and Andrew G Barto. Skill discovery in continuous reinforcement learning domains using skill chaining. In *Advances in neural information processing systems*, pages 1015–1023, 2009.

[55] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1703.01161*, 2017.

[56] Peter Dayan and Geoffrey E Hinton. Feudal reinforcement learning. In *Advances in neural information processing systems*, pages 271–278, 1993.

[57] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *AAAI*, pages 1726–1734, 2017.

[58] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058, 2017.

[59] Andrew Levy, Robert Platt, and Kate Saenko. Hierarchical actor-critic. *arXiv preprint arXiv:1712.00948*, 2017.

[60] Carlos Florensa, Yan Duan, and Pieter Abbeel. Stochastic neural networks for hierarchical reinforcement learning. In *International Conference in Learning Representations (ICLR)*, 2017.

[61] John D Co-Reyes, YuXuan Liu, Abhishek Gupta, Benjamin Eysenbach, Pieter Abbeel, and Sergey Levine. Self-consistent trajectory autoencoder: Hierarchical reinforcement learning with trajectory embeddings. *arXiv preprint arXiv:1806.02813*, 2018.

[62] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. In *International Conference in Learning Representations (ICLR)*, 2014.

[63] Chen Tessler, Shahar Givony, Tom Zahavy, Daniel J Mankowitz, and Shie Mannor. A deep hierarchical approach to lifelong learning in minecraft. In *AAAI*, volume 3, page 6, 2017.

[64] Xue Bin Peng, Glen Berseth, KangKang Yin, and Michiel Van De Panne. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, 36(4):41, 2017.

[65] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009.

[66] Brian D. Ziebart, Andrew L. Maas, J. Andrew Bagnell, and Anind K. Dey. Maximum entropy inverse reinforcement learning. In *Proc. AAAI*, pages 1433–1438, 2008.

[67] Julien Audiffren, Michal Valko, Alessandro Lazaric, and Mohammad Ghavamzadeh. Maximum entropy semi-supervised inverse reinforcement learning. In *Proc. IJCAI*, pages 3315–3321. AAAI Press, 2015.

[68] Krishnamurthy Dvijotham and Emanuel Todorov. Inverse optimal control with linearly-solvable mdps. In *Proc. ICML*, pages 335–342, 2010.

[69] Sergey Levine, Zoran Popovic, and Vladlen Koltun. Nonlinear inverse reinforcement learning with gaussian processes. In *Proc. NIPS*, pages 19–27, 2011.

[70] Monica Babes, Vukosi N. Marivate, Kaushik Subramanian, and Michael L. Littman. Apprenticeship learning about multiple intentions. In *Proc. ICML*, pages 897–904. Omnipress, 2011.

[71] Deepak Ramachandran and Eyal Amir. Bayesian inverse reinforcement learning. *Proc. IJCAI*, 51:2586–2591, 2007.

[72] Gergely Neu and Csaba Szepesvári. Apprenticeship learning using inverse reinforcement learning and gradient methods. In *Proc. UAI*, pages 295–302, 2007.

[73] Jaedeug Choi and Kee-Eung Kim. Nonparametric bayesian inverse reinforcement learning for multiple reward functions. In *Proc. NIPS*, pages 314–322, 2012.

[74] Christos Dimitrakakis and ConstantinA. Rothkopf. Bayesian multitask inverse reinforcement learning. In *Recent Advances in Reinforcement Learning*, volume 7188 of *Lecture Notes in Computer Science*, pages 273–284. Springer Berlin Heidelberg, 2012.

[75] Markus Wulfmeier, Peter Ondruska, and Ingmar Posner. Maximum entropy deep inverse reinforcement learning. *arXiv preprint arXiv:1507.04888*, 2015.

[76] Chelsea Finn, Sergey Levine, and Pieter Abbeel. Guided cost learning: Deep inverse optimal control via policy optimization. In *International Conference on Machine Learning*, pages 49–58, 2016.

[77] Sergey Levine and Pieter Abbeel. Learning neural network policies with guided policy search under unknown dynamics. In *Advances in Neural Information Processing Systems*, pages 1071–1079, 2014.

[78] Chelsea Finn, Paul Christiano, Pieter Abbeel, and Sergey Levine. A connection between generative adversarial networks, inverse reinforcement learning, and energy-based models. *arXiv preprint arXiv:1611.03852*, 2016.

[79] Alberto Maria Metelli, Matteo Pirotta, and Marcello Restelli. Compatible reward inverse reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2050–2059, 2017.

[80] Xingyu Wang and Diego Klabjan. Competitive multi-agent inverse reinforcement learning with sub-optimal demonstrations. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 5143–5151, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[81] Kareem Amin, Nan Jiang, and Satinder Singh. Repeated inverse reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1815–1824, 2017.

[82] Balaraman Ravindran. Smdp homomorphisms: An algebraic approach to abstraction in semi markov decision processes. 2003.

[83] Gerhard Neumann. Lecture notes: Constraint optimization. 2015.

[84] WT Miller and FH Glanz. Unh cmac version 2.1: The university of new hampshire implementation of the cmac, 1996.

[85] George Konidaris, Sarah Osentoski, and Philip S Thomas. Value function approximation in reinforcement learning using the fourier basis. In *AAAI*, volume 6, page 7, 2011.

[86] Andras Gabor Kupcsik, Marc Peter Deisenroth, Jan Peters, Gerhard Neumann, et al. Data-efficient generalization of robot skills with contextual policy search. In *Proceedings of the 27th AAAI Conference on Artificial Intelligence, AAAI 2013*, pages 1401–1407, 2013.

[87] Xueli Jia. Deep learning for actor-critic reinforcement learning. 2015.

[88] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[89] Yangyang Xu and Wotao Yin. A block coordinate descent method for regularized multiconvex optimization with applications to nonnegative tensor factorization and completion. *SIAM Journal on imaging sciences*, 6(3):1758–1789, 2013.

[90] Frank Sehnke and Tingting Zhao. Baseline-free sampling in parameter exploring policy gradients: Super symmetric pgpe. In *Artificial Neural Networks*, pages 271–293. Springer, 2015.

[91] Tingting Zhao, Hirotaka Hachiya, Gang Niu, and Masashi Sugiyama. Analysis and improvement of policy gradient estimation. In *Proc. NIPS*, pages 262–270. Curran Associates, Inc., 2011.

[92] Giorgio Manganini, Matteo Pirotta, Marcello Restelli, and Luca Bascetta. Following newton direction in policy gradient with parameter exploration. In *Proc. IJCNN*, pages 1–8. IEEE, 2015.

[93] Nikos Vlassis, Marc Toussaint, Georgios Kontes, and Savas Piperidis. Learning model-free robot control by a monte carlo em algorithm. *Autonomous Robots*, 27(2):123–130, 2009.

[94] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[95] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[96] Charles W. Anderson and W. Thomas Miller. Neural networks for control. chapter A Challenging Set of Control Problems, pages 475–508. MIT Press, Cambridge, MA, USA, 1990.

Appendix A

Description of the environments

In this appendix, we will summarize all the details regarding all the environments used in the experiments of this work.

A.1 LQR

The Linear Quadratic Regulator (LQR) domain is the classical control theory task of stabilizing a linear system (with a linear feedback controller) under a quadratic cost objective function. The system dynamics are:

$$x(t+1) = Ax(t) + Bu(t).$$

In our experiments we used the identity matrix for A and B. The reward function is:

$$r(t) = -x(t)^T Q x(t) - u(t)^T R u(t).$$

The discount factor for every experiment using this environment is set to $\gamma = 0.9$. This environment has no absorbing states, but we set a horizon of 50 steps for each episode.

A.2 NLS

The NLS environment is a 2D nonlinear system defined in [93]. The system dynamics are:

$$\begin{aligned} x_1(t+1) &= x_1(t) - 0.2\, x_2(t+1) + \eta, \\ x_2(t+1) &= x_2(t) + \frac{1}{1+\exp(-u(t))} - 0.5 + \eta. \end{aligned}$$

where $\eta \sim \mathcal{N}(0, 0.02)$.

The reward function is:

$$r(t) = \begin{cases} 1 & \|x\| < 0.1 \\ 0 & \textit{otherwise} \end{cases}.$$

The trajectories have a horizon of 80 control steps. The discount factor used in the experiment involving this non-linear system is $\gamma = 0.95$.

A.3 Ship Steering

The ship steering problem consists in driving a ship towards a gate. Similar environments have been used both in control theory [96] and reinforcement learning [46]. The state dynamics for the environment is the following:

$$\begin{aligned} x(t+1) &= x(t) + v\sin(\theta(t))dt \\ y(t+1) &= y(t) + v\cos(\theta(t))dt \\ \theta(t+1) &= \theta(t) + \omega(t)dt \\ \omega(t+1) &= \omega(t) + (u - \omega(t))dt/T \end{aligned}$$

With $v = 3.0m/s$, $dt = 0.2s$, $T = 5.0$ and $u \in [-\frac{\pi}{12}, \frac{\pi}{12}]$ (radians/s). The transition model applies the above dynamics for n steps. The reward function is:

$$r(t) = \begin{cases} r_{\text{out}} & \textit{outOfBounds} \\ 0 & \textit{throughGate} \\ -1 & \textit{otherwise} \end{cases},$$

where *throughGate* is true if the ship crossed the gate in the last transition, and *outOfBounds* is true if the ship goes out of the environment bounds. If the ship crosses the gate or goes outside the field, the trajectory ends. The trajectory horizon is set to 5000 control steps. The discount factor used in the experiments on all variants of this environment is $\gamma = 0.99$.

Variable	Min	Max
x	0	150
y	0	150
θ	$-\pi$	π
$\dot{\theta}$	$-\frac{\pi}{12}$	$\frac{\pi}{12}$
u	$-\frac{\pi}{12}$	$\frac{\pi}{12}$

Table A.1: State and action ranges for small and difficult environment

Variable	Min	Max
x	0	1000
y	0	1000
θ	$-\pi$	π
$\dot{\theta}$	$-\frac{\pi}{12}$	$\frac{\pi}{12}$
u	$-\frac{\pi}{12}$	$\frac{\pi}{12}$

Table A.2: State and action ranges for big environment

Three different versions of this environment are used in this work: the small environment, the big environment (both used in Chapter 4), and the difficult environment (used in Chapter 5). Both the small and the difficult environment share the same state and action space, reported in Table A.1. The gate is a line connecting the two points $g_s = (100, 120)$ and $g_e = (120, 100)$. The difference between these two environments are the reward and the integration steps. In the small environment, we have $n = 3$ and $r_{\text{out}} = -100$, while in the difficult environment $n = 1$ and $r_{\text{out}} = -10000$.

The state bounds for the ship steering environment are reported in Table A.2. The integration steps and the out of bounds penalty for this version of the environment are the same as the small environments i.e., $n = 3$ and $r_{\text{out}} = -100$.

A.4 Segway

The Segway is a balancing wheeled robot, that can be described as an inverted pendulum mounted on a wheel, as shown in Appendix A.4. The objective of this environment is to drive the Segway towards the goal position while maintaining vertical the inverted pendulum as much as possible. This

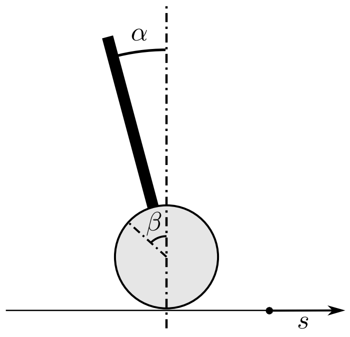


Figure A.1: The Segway environment and state variables

Parameter	value
M_r	$0.3 * 2$
Mp	2.55
I_p	$2.6 \cdot 10^{-2}$
I_r	$4.54 \cdot 10^{-4} * 2$
l	$13.8 \cdot 10{-}2$
r	$5.5 \cdot 10^{-2}$
dt	10^{-2}
g	9.81

Table A.3: Parameters used in the Segway environment

environment is very similar to the Segway described in [87], however differently from the previous work, we added the position control. The system dynamics are the following:

$$\begin{aligned}
\dot{\alpha} &= \omega_\alpha \\
\dot{\beta} &= \omega_\beta \\
\dot{s} &= -\omega_\beta r \\
\dot{\omega_\alpha} &= -\frac{h_2 l M_p r \omega_\alpha^2 \sin\alpha - g h_1 l M_p \sin\alpha + (h_2 + h_1) u}{h_1 h_3 - h_2^2} \\
\dot{\omega_\beta} &= \frac{h_3 l M_p r \omega_\alpha^2 \sin\alpha - g h_2 l M_p \sin\alpha + (h_3 + h_2) u}{h_2 h_3 - h_2^2} \\
h_1 &= (M_r + M_p) r^2 + I_r \\
h_2 &= M_p r l \cos(\alpha) \\
h_3 &= l^2 M_p + I_p.
\end{aligned}$$

The control action u is constrained such that the applied torque falls in the interval $[-5, 5]$. The discount factor used in the experiments involving this environment is $\gamma = 0.99$. The values for the parameters of the system can be found in Table A.3.

The transition model is obtained by integrating the system dynamics, applying constant torque action u, for $\Delta t = 0.02$. The initial state is $x_0 = [-1.0, \frac{\pi}{8}, 0, 0]$. The discount factor for every experiment using this environment is set to $\gamma = 0.9$. All the states where $\alpha \notin [-\frac{\pi}{2}, \frac{\pi}{2}]$ or $s \notin [-2, 2]$ are absorbing states. All trajectories that doesn't reach an absorbing state are cut using a horizon of 1500 control steps. The reward function is the following:

$$r(t) = \begin{cases} -10000 & \textit{fall} \\ -10000 & \textit{out} \\ -x(t)Qx(t)^T, & \textit{otherwise} \end{cases},$$

where $Q = diag([10, 3, 0.1, 0.1])$ is a diagonal weights matrix for the quadratic cost penalty, *fall* and *out* are true when the pendulum angle α and the position variable s go outside their bounds.

A.5 Prey-Predator

The prey-predator environment consists of two differential drive robots, where the objective of the learning agent, the predator, is to catch the other robot, the prey, inside a square environment with obstacles. Both robots are considered as points, with no footprint. The prey has a higher maximum angular velocity than the predator but they have the same turning radius ($0.6m$). The differential drive dynamics used are the following:

$$\begin{aligned} x(t+1) &= x(t) + \cos(\theta(t))v(t)dt \\ y(t+1) &= y(t) + \sin(\theta(t))v(t)dt \\ \theta(t+1) &= \theta(t) + \omega(t)dt. \end{aligned}$$

In our experiments we set $dt = 0.1$. State and action are bounded as described in Table A.4. The orientation is computed w.r.t. the x axis. The environment includes the following obstacles: the segment $(1, 3.48) - (5, 3.48)$ and the polygonal chain $(-3.0, -1.5) - (-3.0, 1.25) - (-1.48, 1.25) - (-1.48, -1.5)$. In Fig. A.2 a graphical representation of this environment is shown.

Variable	Min	Max
$x_{\text{prey, predator}}$	-5	5
$y_{\text{prey, predator}}$	-5	5
$\theta_{\text{prey, predator}}$	$-\pi$	π
v_{prey}	0	1.3
v_{predator}	0	1.0
ω_{prey}	-2.1666	2.1666
ω_{predator}	-1.666	1.666

Table A.4: State and action ranges for prey-predator environment

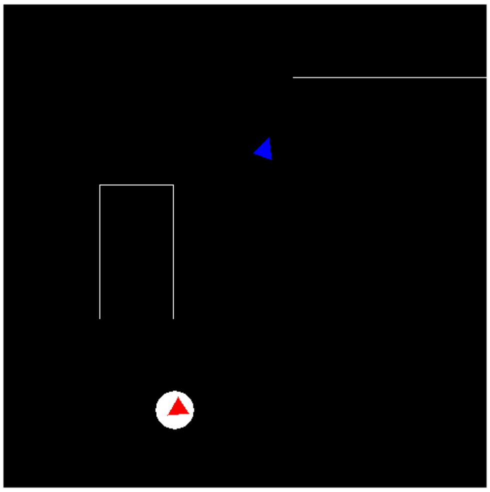


Figure A.2: The prey-predator environment. Predator is shown in red, prey in blue. The white area around the predator represents the catch radius.

The prey is considered captured if the predator distance is less than $0.4m$ and there is no obstacle between the two robots. If the prey has been captured, then the episode ends. All episodes are cut when they reach a horizon of 500 steps.

The evasion policy of the prey is a complex policy. The linear velocity is calculated as follows:

$$v_{\text{prey}}(t) = \begin{cases} 0 & \text{distance} > 3.0 \\ 1.3 & \text{distance} \leq 1.5 \\ 0.65 & \textit{otherwise} \end{cases}$$

To compute the angular velocity, first is computed the difference between the current prey orientation and the angle of attack the predator i.e., the angle w.r.t. the x axis of the segment connecting prey and predator. Then the value of the angular velocity is computed by multiplying the error by a proportional gain $k = \frac{1}{\pi}$. If a potential collision is detected within $1.5m$, the prey rotates in the direction of the biggest angle at maximum angular velocity. If following this rotation, the prey is closer than $0.9m$ to another obstacle, the rotation is inverted. This last behavior is implemented in order to maneuver properly near the corners, avoiding to get easily trapped.

The reward function for this environment is computed by multiplying by -1 the distance between the prey and the predator after applying the action i.e., it is computed using the next state. For every experiment using this environment, the discount factor is set to $\gamma = 0.99$.