Adversarial Inverse Reinforcement Learning with Changing Dynamics

BY

ANDREA TIRINZONI
B.S, Politecnico di Milano, Milan, Italy, 2015

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2017

Chicago, Illinois

Defense Committee:

Brian D. Ziebart, Chair and Advisor

Piotr J. Gmytrasiewicz

Marco D. Santambrogio, Politecnico di Milano

To my dear friend Pino

ACKNOWLEDGMENTS

I want to thank my advisor, professor Brian Ziebart, for giving me the opportunity to work on this project and supporting me during its development. Thanks to everyone working in the Purposeful Prediction Lab for helping me and spending time with me during the last semester. I also want to thank my Polimi advisor, professor Marcello Restelli, for providing his feedback on this work.

I want to thank my parents, Tiziana and Stefano. Without their support, confidence, and motivation, I would not be writing this document. Thanks to all my friends for sharing my happiness when I was admitted at UIC and motivating me during this whole year in Chicago.

Thanks to the Polimi-UIC students of Fall 2016, Marco, Luca, Alessandro, Davide, Giacomo and Eleonora, who shared with me this new experience and with whom I had an unforgettable time.

Finally, I want to thank Eleonora for her continuous support during the last semester, for her help in the review of this document and her precious suggestions.

AT

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

IRL	Inverse Reinforcement Learning
IOC	Inverse Optimal Control
MDP	Markov Decision Process
POMDP	Partially Observable Markov Decision Process
PWLC	Piece-Wise Linear and Convex
PBVI	Point-Based Value Iteration

SUMMARY

Most work on inverse reinforcement learning, the problem of recovering the unknown reward function being optimized by a decision-making agent, has focused on cases where optimal demonstrations are provided under single dynamics. We analyze the more general settings where the learner has access to sub-optimal demonstrations under several different dynamics. We argue that several problems, such as learning under covariate shift or risk aversion, can be modeled in this way.

We propose an adversarial formulation where the learner tries to imitate a constrained, worst-case estimate of the demonstrator's control policy. We adopt the method of Lagrange multipliers to remove the constraints and produce a convex optimization problem.

We prove that the constraints imposed by the multiple dynamics lead to an NP-Hard optimization subproblem, the computation of a deterministic policy maximizing the total expected reward from several different Markov decision processes. We propose a tractable approximation by reducing the latter to the optimal control of partially observable Markov decision processes.

We show the performance of our algorithm on two synthetic data problems. In the first one, we try to recover the reward function of a randomly generated Markov decision process, while in the second we try to rationalize a robot navigating through a grid and demonstrating goal-directed behavior.

CHAPTER 1

INTRODUCTION

Inverse reinforcement learning (IRL) [1][2][3] is the problem of recovering the unknown reward function being optimized by an agent in a decision-making context (modeled as a Markov decision process) given demonstrated behavior. This has two main motivations. First, neuroscience studies proved that reinforcement learning [4], the problem of learning the optimal way to behave in order to maximize a long-term utility, naturally occurs in the human brain and in certain animals [5] [6]. Although it is known that these subjects act to maximize some reward, the latter is generally unknown. Thus, it seems wise to estimate the optimized utility by observing the agent's behavior. For instance, reinforcement learning occurs in honey bees during the process of nectar acquisition [7]. However, honey bees might weigh nectar intake with some other factors, such as traveled distance or risk, in order to decide whether the acquisition is worth. Since specifying such weights is definitely a complicated problem, it makes much more sense to estimate them by observing the bees' behavior.

The second motivation behind IRL is the difficulty in creating artificial decision-making agents. The common approach is to design a reward function specifying the task to be solved and train the agent to behave so as to maximize such reward. However, this constitutes a non-trivial problem when the task to be specified is complicated. Consider for example the problem of learning how to drive a car. The driver might take into consideration several variables in order to decide what is the best thing to do, such as the speed limit on the current road, the

presence of other cars or pedestrians, and so on. Once again, weighing such variables so as to obtain the desired behavior is very difficult, while it is much easier to learn from, e.g., human driving demonstrations (this is called apprenticeship learning [3]).

In this document, we analyze and solve a particular IRL problem, which we describe in section 1.1. The motivations and resulting contributions of this work are detailed in sections 1.2 and 1.3, respectively. Finally, the document is outlined in section 1.4, while the mathematical notation we adopt is specified in section 1.5.

1.1 Problem Description

Most work on IRL has focused on problems where provided demonstrations are optimal (or nearly optimal). Furthermore, it is common to assume that demonstrations are generated under fixed dynamics. This allows efficient algorithms for recovering a reward function, not necessarily the real one, that makes the learner achieve performances comparable to those of the demonstrator, even when the former is acting under different dynamics. However, existing algorithms do not support the case where demonstrations are provided under different dynamics, and it is difficult to extend them to handle such situation.

We consider the more general settings where sub-optimal demonstrations are provided under different dynamics. We suppose the demonstrator is acting according to some policy that is optimal for a known environment (i.e., state transition dynamics) and that demonstrations are provided under a set of different known environments. We suppose the existence of a single reward function that can be represented as a linear combination of given features of the states. Our goal is to estimate a reward function that allows the learner to imitate the demonstrator

under its dynamics (those from which we are not provided trajectories), and to improve the expert's sub-optimal performance in the observed environments.

We adapt the adversarial formulation described in [8] to efficiently solve our generalized problem. Our formulation is still a zero-sum game where the learner tries to imitate a constrained, worst-case estimate of the demonstrator's control policy. However, the addition of more constraints, due to the demonstrations under different dynamics, leads to an NP-Hard optimization problem, that is, the computation of a deterministic policy maximizing the total expected reward from several processes. We propose a tractable approximation by reducing the latter to the optimal control of partially observable Markov decision processes [9][10].

1.2 Motivations

The problem we are trying to solve has three main different motivations. We highlight them in this section.

Imitation Learning under Covariate Shift

In machine learning, covariate shift [11] denotes a scenario where train data and test data have different distributions, but the concept that is learned (typically the probability of the target variable given the inputs) remains the same. When the distributions are known, a simple solution is to re-weight the train data by adopting importance sampling [12]. Consider, however, the case where an agent tries to imitate an expert (called imitation learning and better introduced in 2), and suppose the latter provides demonstrations under different dynamics than those for which it is optimal. If test data, used to evaluate the imitator, is provided under the latter dynamics, we obtain a covariate shift problem where importance sampling is not

applicable (the demonstrator's policy is unknown). We argue that our method can be used to deal with such case without re-weighting the data.

IRL under Risk Aversion

Risk aversion in the context of Markov decision processes [13] is the idea that maximizing the expected reward may not lead to the optimal behavior in many cases. Consider, for instance, a robot that is moving towards a certain goal in an environment with several pitfalls where the former could break. Maximizing the expectation could lead the robot to move very close to a pitfall in order to reach the goal as soon as possible. However, this is not taking into account the intrinsic variance in the environment, which, with very small probability, might bring the robot to a pitfall, leading to disastrous consequences. On the other hand, a risk averse behavior is such that the agent moves safely far away from the pitfalls.

The main idea to deal with risk aversion is to maximize different functions instead of the expectation. We argue that another approach is to change the agent's dynamics so that the desired risk-averse behavior is learned and, successively, executed in the real environment.

Consider now the case when the demonstrator in our IRL settings is risk-averse. This means that trajectories are generated in those environments where the risk-averse behavior is employed, but the corresponding policy is only optimal for the changed dynamics used to learn such behavior. Once again, our framework can be used to deal with such problem.

IRL under Robust Behavior

Robust control of Markov decision processes [14] is the problem of finding the best way to behave when state-transition dynamics are uncertain. The typical solution is to compute the optimal policy under the worst-case dynamics among those in certain uncertainty sets.

Suppose now the expert in our IRL settings is robust. This means it is executing such robust policy under different instances of the uncertain dynamics. We argue that our framework can be used to recover the reward function even in this situation.

1.3 Contributions

Our contribution is two-fold. First, we propose an approximate solution to the NP-Hard problem of computing the optimal deterministic policy maximizing the total reward from different processes. This a general problem, not necessarily only related to IRL, whose solution can be re-used in many contexts. To this end, we analyze and solve it from the most general perspective possible, so as to ease its re-usability.

Our second contribution is specifically related to IRL. We propose a framework for dealing in a principled way with the problem introduced in section 1.1 and motivated in section 1.2. Since existing algorithm are not suitable for our particular settings, we believe our work constitute a significant improvement.

1.4 Document Outline

The document is organized as follows. In chapter 2, we provide the mathematical background needed to understand this work. Focus is given to Markov decision processes (MDPs), the mathematical model that specifies our decision-making settings, partially observable Markov

decision processes (POMDPs), which we adopt to approximate the NP-Hard sub-problem of our formulation, and causally conditioned probability distributions, which compactly represent our stochastic processes. Furthermore, we introduce the reader to imitation learning, behavioral cloning and, most importantly, IRL. In chapter 3, we continue the description of IRL by providing an overview of the most important state-of-the-art algorithms (feature matching, maximum entropy and maximum margin planning). We also detail the adversarial approach we extend to define our formulation. Chapter 4 formally defines the problem we are trying to solve, while Chapter 5 presents our adversarial formulation and derives our learning algorithm. In chapter 6, we reduce the NP-Hard sub-problem to planning in partially observable environments. We propose a modified POMDP algorithm, analyze its performance and, finally, specify its application to our IRL formulation. In Chapter 7, we demonstrate the performance of our algorithm on two synthetic-data problems. In the first we randomly generate an MDP and try to recover its reward function, while in the second we do the same for a robot navigating through a grid. Finally, chapter 8 summarizes our work and describes some of the possible future extensions.

1.5 Mathematical Notation

In this document, we adopt the following mathematical notation:

- Variables are denoted as lower-case letters (x,y,...);
- Random variables are denoted as upper-case letters (X,Y,...);
- Vectors are denoted as lower-case bold letters ($\mathbf{x}$,$\mathbf{y}$,...);
- Random vectors are denoted as upper-case bold letters ($\mathbf{X}$,$\mathbf{Y}$,...).

CHAPTER 2

BACKGROUND

This chapter provides the necessary background to fully understand this work. Section 2.1 starts by describing MDPs, the mathematical tool to model environments where reinforcement learning and inverse reinforcement learning are applied. Then, section 2.2 provides an overview of POMDPs, which we employ in chapter 6 to solve an important sub-problem of our formulation. Since these two topics are well known and well described in the literature, we simply provide the mathematical foundations, while referring the reader to other sources for more detailed explanations. Section 2.3 quickly introduces directed information theory and causally conditioned probability distributions, which allow for a more concise representation of our stochastic processes. Finally, Section 2.4 describes the problem of imitation learning, focusing on the two main methodologies to solve it: behavioral cloning and inverse reinforcement learning.

2.1 Markov Decision Processes

This chapter provides a quick introduction to MDPs. Since in this document we restrict ourselves to finite-state, finite-horizon MDPs, we focus on such case. A description of the more general settings can be found in [15] and [4].

MDPs are discrete time stochastic control processes used to model complex decision-making problems under uncertainty. At each time of the process, the agent is in some state, takes some

action and observes a reward for taking that action in that particular state. The agent's task is to find the sequence of actions so as to maximize the long-term total reward. Definition 2.1.1 formalizes this concept.

Definition 2.1.1. *A Markov Decision Process (MDP) is a tuple* $< \mathcal{S}, \mathcal{A}, \tau, R, p_0 >$*, where:*

- $\mathcal{S}$ *is a finite set of states;*
- $\mathcal{A}$ *is a finite set of actions;*
- τ *are the state transition dynamics, where* $\tau(s_{t+1} \mid s_t, a_t)$ *is the probability of the next state being* s_{t+1} *given that we execute action* a_t *from state* s_t*;*
- R *is a reward function, where* $R(s_t)$ *specifies the reward that the agent receives for entering state* s_t[1]*;*
- p_0 *is a distribution of probability over initial states, where* $p_0(s)$ *is the probability that* s *is the state where the process starts.*

A control policy $\pi(a_t \mid s_t)$ is a probability distribution over actions given states. The process starts at time $t = 1$, where the first state is drawn from the initial distribution p_0. Then, the agent selects the first action a_1 according to its policy $\pi(a_1 \mid s_1)$ and makes a transition to another state as specified by the dynamics $\tau(s_2 \mid s_1, a_1)$. The agent successively selects the

---

[1] *Notice that the reward is usually specified as a function* $R(s, a)$ *of state and action, but in this document we consider it as a function of the state alone. The extension is trivial.*

second action, and so on until the process ends in state s_T. The goal is to find the policy $\pi^\star$ which maximizes the sum of expected rewards, that is:

$$\pi^\star = \underset{\pi}{argmax}\ \mathbb{E}[\sum_{t=1}^{T} R(S_t) \mid \tau, \pi] \tag{2.1}$$

It is possible to prove that the knowledge of the current state is sufficient for acting optimally in the future; that is, knowing the past history of state-action sequences does not add any further information. A process where this is the case is said to be Markovian. Furthermore, the optimal policy in an MDP is always deterministic, i.e., it can be specified as a mapping $\pi^\star(s_t)$ from states to actions returning the best action a_t for each state s_t.

For a particular policy π, the state value function represents the total expected reward that is achieved from each state s_t by following π:

$$V^\pi(s_t) = R(s_t) + \mathbb{E}[\sum_{i=t+1}^{T} R(S_i)|\tau, \pi] \tag{2.2}$$

while the state-action value function represents the total expected reward that is achieved by executing action a_t from state s_t and then following π:

$$Q^\pi(s_t, a_t) = R(s_t) + \mathbb{E}_{s_{t+1}\sim\tau(\cdot|s_t,a_t)}[\sum_{i=t+1}^{T} R(S_i)|\tau, \pi] \tag{2.3}$$

These two quantities are related by the so-called Bellman expectation equations [16], as specified in Theorem 2.1.1.

Theorem 2.1.1. *Let $M = < \mathcal{S}, \mathcal{A}, \tau, R, p_0 >$ be an MDP and $\pi(a_t \mid s_t)$ be a policy. Then we can compute the state value function $V^\pi(s_t)$ and the state-action value function $Q^\pi(s_t, a_t)$ for π by solving the following dynamic program:*

$$V^\pi(s_T) = R(s_T) \tag{2.4}$$

$$V^\pi(s_t) = \sum_{a_t} \pi(a_t \mid s_t) Q^\pi(s_t, a_t) \ \forall t = T-1, ..., 1 \tag{2.5}$$

$$Q^\pi(s_t, a_t) = R(s_t) + \sum_{s_{t+1}} \tau(s_{t+1} \mid s_t, a_t) V^\pi(s_{t+1}) \ \forall t = T-1, ..., 1 \tag{2.6}$$

The following theorem states Bellman's optimality equations [16], which represent one of the most important results in this field and allow the computation of the optimal policy $\pi^\star$ by means of dynamic programming.

Theorem 2.1.2. *Let $M = < \mathcal{S}, \mathcal{A}, \tau, R, p_0 >$ be an MDP. Then we can compute the optimal policy $\pi^\star(s_t)$ for M by solving the following dynamic program:*

$$V^{\pi^\star}(s_T) = R(s_T) \tag{2.7}$$

$$V^{\pi^\star}(s_t) = \underset{a_t}{max} \ Q^{\pi^\star}(s_t, a_t) \ \ \forall t = T-1, ..., 1 \tag{2.8}$$

$$\pi^\star(s_t) = \underset{a_t}{argmax} \ Q^{\pi^\star}(s_t, a_t) \ \ \forall t = T-1, ..., 1 \tag{2.9}$$

where $Q^{\pi^\star}$ is computed as specified in Equation 2.6.

2.2 Partially Observable Markov Decision Processes

This section quickly introduces POMDPs, focusing on the properties we use in this work. For a more detailed description, we refer the reader to [17].

POMDPs [9] [10] provide an extension of MDPs to partially observable environments, i.e., those where the agent cannot directly observe the state but only receives partial information about it. Definition 2.2.1 formalizes this new model.

Definition 2.2.1. *A partially observable Markov decision process (POMDP) is a tuple* $< \mathcal{S}, \mathcal{A}, \Omega, \tau, O, R, b_0, \gamma >$*,where:*

- $\mathcal{S}$ *is a finite set of states;*
- $\mathcal{A}$ *is a finite set of actions;*
- Ω *is a finite set of observations, where each observation gives partial information about the state the agent is into;*
- τ *are the state transition dynamics, where* $\tau(s_{t+1} \mid s_t, a_t)$ *is the probability of the next state being* s_{t+1} *given that we execute action* a_t *from state* s_t*;*
- O *are the conditional observation probabilities, where* $O(o_{t+1} \mid s_{t+1}, a_t)$ *is the probability of observing* o_{t+1} *after taking action* a_t *and transitioning to state* s_{t+1}*;*
- R *is the reward function, where* $R(s_t, a_t)$ *specifies the utility the agent obtains after taking action* a_t *from state* s_t*;*
- b_0 *is the initial state probability distribution (also called initial belief state);*

- $\gamma \in [0, 1]$ *is the discount factor and it is used to discount future rewards in infinite-horizon processes.*

The process starts at time $t = 1$, where the first state s_1 is drawn from b_0. However, the agent cannot directly observe such state but receives an observation o_1. Then, the agent selects an action a_1, transitions to the unobserved state s_2 according to $\tau(s_2 \mid s_1, a_1)$, receives an observation o_2 according to $O(o_2 \mid s_2, a_1)$ and finally obtains a reward $R(s_1, a_1)$. Then, the process is repeated (forever in case of infinite-horizon). The agent's goal is to select the sequence of action maximizing the total (discounted) reward over time.

In order to solve the problem, the agent keeps, at each time, a distribution over states $\mathcal{S}$ called belief state. A belief state is an $|\mathcal{S}|$-dimensional vector where each component represents the probability that the agent is in that particular state. Then, the partially observable MDP is reduced to a continuous fully observable MDP where the state space is $\mathcal{B}$, the space of all belief states (if we have $|\mathcal{S}|$ states, $\mathcal{B}$ is the $(|\mathcal{S}| - 1)$-dimensional simplex). In such MDP, the state-transition dynamics are:

$$\tau(\boldsymbol{b}_{t+1} \mid \boldsymbol{b}_t, a_t) = \sum_{o \in \Omega} Pr(\boldsymbol{b}_{t+1} \mid \boldsymbol{b}_t, a_t, o) Pr(o \mid a_t, \boldsymbol{b}_t) \tag{2.10}$$

and the reward function is:

$$R(\boldsymbol{b}_t, a_t) = \sum_{s \in \mathcal{S}} R(s, a_t) \boldsymbol{b}(s) \tag{2.11}$$

According to Bellman optimality equations, the value function for the optimal policy of the fully observable MDP can be written as:

$$V_t(\boldsymbol{b}_t) = \max_{a_t \in \mathcal{A}} \left[R(\boldsymbol{b}_t, a_t) + \gamma \sum_{\boldsymbol{b}_{t+1} \in \mathcal{B}} \tau(\boldsymbol{b}_{t+1} \mid \boldsymbol{b}_t, a_t) V_{t+1}(\boldsymbol{b}_{t+1}) \right] \tag{2.12}$$

This is a function of a continuous variable and cannot be computed by standard value iteration [4]. However, it turns out that such function is piece-wise linear and convex in $\boldsymbol{b}$ [9] and can be written as:

$$V_t(\boldsymbol{b}_t) = \max_{\boldsymbol{\alpha} \in \mathcal{V}_t} \boldsymbol{b}_t^\intercal \boldsymbol{\alpha} \tag{2.13}$$

where $\mathcal{V}_t$ is a set of vectors representing the normal directions to the hyperplanes describing the function (we call them alpha-vectors).

Such formulation allows for efficient algorithms to compute the value function (and the associated optimal policy). The most common approach, and the one we adopt in this document, is point-based value iteration (PBVI), which we describe in the next section.

2.2.1 Point-based Value Iteration

Point-based value iteration [18] approximates the optimal value function of a POMDP by considering only a restricted set of representative belief states $\mathcal{B}$ and computing one alpha-vector for each of them. Then, the value function is represented as the set $\mathcal{V}$ containing all alpha-vectors learned by the algorithm. The motivation behind PBVI is that most alpha-vectors in the full set exactly describing V are dominated by some others. This means that, when taking the maximum over actions, such vectors are never used and can be safely pruned. Since the

total number of alpha-vectors grows exponentially with the number of actions and observations, pruning is necessary to make the problem tractable. In PBVI, this is done implicitly by computing the dominating alpha-vector at each belief in $\mathcal{B}$. Thus, no dominated alpha-vector is ever added to $\mathcal{V}$.

The algorithm proceeds iteratively by alternating two main procedures:

- value backup: given the current belief set $\mathcal{B}$ and the current value function $\mathcal{V}$, computes an update $\mathcal{V}'$ of $\mathcal{V}$ by applying some backup equations similar to the Bellman optimality equations (a modified version of Equation 2.12);
- belief expansion: adds new belief states to the current belief set $\mathcal{B}$. For each belief in $\mathcal{B}$, this is done by taking each action and simulating a step forward, thus leading to a new belief state. Then, only the farthest belief from the current set $\mathcal{B}$ is retained.

These two procedures are then repeated for a specified number of iterations. The more iterations are executed, the better is the value function approximation (i.e., the number of alpha-vectors that are computed grows with the number of iterations).

2.3 Directed Information Theory

In this document, we make use of causally conditioned probability distributions to represent our stochastic processes. Such distributions arise naturally from directed information theory [19], which associates a direction to the flow of information in a system.

Given two random vectors $\boldsymbol{Y}_{1:T}$ and $\boldsymbol{X}_{1:T}$, the causally conditioned probability distribution of $\boldsymbol{Y}$ given $\boldsymbol{X}$ is:

$$p(\boldsymbol{Y}_{1:T} \mid\mid \boldsymbol{X}_{1:T}) = \prod_{t=1}^{T} p(Y_t \mid \boldsymbol{X}_{1:t}, \boldsymbol{Y}_{1:t-1}) \tag{2.14}$$

Notice the difference with respect to the conditional probability of $\boldsymbol{Y}$ given $\boldsymbol{X}$ in the limited knowledge about the conditioning variable $\boldsymbol{X}$ available:

$$p(\boldsymbol{Y}_{1:T} \mid \boldsymbol{X}_{1:T}) = \prod_{t=1}^{T} p(Y_t \mid \boldsymbol{X}_{1:T}, \boldsymbol{Y}_{1:t-1}) \tag{2.15}$$

Using this notation, we can compactly represent our processes as:

$$\tau(\boldsymbol{S}_{1:T} \mid\mid \boldsymbol{A}_{1:T-1}) = \prod_{t=1}^{T} p(S_t \mid \boldsymbol{S}_{1:t-1}, \boldsymbol{A}_{1:t-1}) \overset{(M)}{=} \prod_{t=1}^{T} p(S_t \mid S_{t-1}, A_{t-1}) \tag{2.16}$$

$$\pi(\boldsymbol{A}_{1:T-1} \mid\mid \boldsymbol{S}_{1:T-1}) = \prod_{t=1}^{T-1} p(A_t \mid \boldsymbol{S}_{1:t}, \boldsymbol{A}_{1:t-1}) \overset{(M)}{=} \prod_{t=1}^{T} p(A_t \mid S_t) \tag{2.17}$$

where the last equalities (M) hold only under the Markovian assumption. The product of these two quantities represent the joint distribution of state-action sequences:

$$p(\boldsymbol{S}_{1:T}, \boldsymbol{A}_{1:T-1}) = \tau(\boldsymbol{S}_{1:T} \mid\mid \boldsymbol{A}_{1:T-1})\pi(\boldsymbol{A}_{1:T-1} \mid\mid \boldsymbol{S}_{1:T-1}) \tag{2.18}$$

and can be used to concisely write the total expected reward as:

$$\mathbb{E}[\sum_{t=1}^{T} R(s_t) \mid \tau, \pi] = \sum_{\boldsymbol{S}_{1:T}, \boldsymbol{A}_{1:T-1}} p(\boldsymbol{S}_{1:T}, \boldsymbol{A}_{1:T-1}) R(\boldsymbol{S}_{1:T}) \tag{2.19}$$

where $R(\boldsymbol{S}_{1:T})$ is the sum of rewards received along the state sequence $\boldsymbol{S}_{1:T}$.

2.4 Imitation Learning

In imitation learning, an agent tries to replicate another agent's behavior. The imitating agent is usually called "learner", while the imitated one is referred to as "expert" or "demonstrator". The environment for this context is generally modeled as a set of states in which the agents can be located, and a set of actions that they can take to perform state transitions. Then, the problem of imitation learning reduces to finding a policy, that is, a mapping from states to actions (or a distribution of probability over actions given states) that best approximates the expert's policy. The latter is known only through demonstrations, i.e., the expert shows how to behave (by taking actions in different states) and the learner has to find the policy that best reproduces the demonstrated state-action sequences.

The two most common approaches to imitation learning are behavioral cloning and inverse reinforcement learning, which are described in the next two sections.

2.4.1 Behavioral Cloning

Behavioral cloning [20] is the most common approach to imitation learning. The main problem is reduced to a supervised learning one, where a mapping from states, or features of such states, to actions is learned by adopting classification techniques. Such method has been successfully adopted in several different fields. Robotics is the most common, where excellent results have been achieved in a wide variety of tasks. Among the notable examples, Pomerleau [21] implemented an autonomous land vehicle that learns to follow a road given demonstrated trajectories (from camera images) and using a neural network. LeCun et al. [22] proposed

a system for learning obstacle avoidance from human demonstrations (again in the form of images) and using a convolutional neural network. Sammut et al. [23] designed an algorithm to learn how to fly an aircraft given human demonstrations from a simulation software and using decision trees.

Although behavioral cloning has proven to perform very well on some specific tasks, it provides poor results when the goal is to maximize a long-term utility. The main issue with behavioral cloning (and with the supervised techniques adopted) is that samples (typically state-action couples) are supposed to be independent. This is clearly not the case when sequential decision making is demonstrated. For example, the state the agent is into depends on the previous state and action. Thus, when the demonstrator's behavior aims at maximizing a long-term reward, behavioral cloning algorithms tend to generalize poorly and fail at reproducing the optimal performance.

2.4.2 Inverse Reinforcement Learning

The idea to tackle problems of imitation learning when the demonstrator is showing sequential decision-making behavior is to formalize them as MDPs. In this context, the expert is supposed to be optimizing an unknown long-term reward function, and imitation learning reduces to estimating such function. This problem is known as inverse reinforcement learning (IRL) or inverse optimal control (IOC) [1][2][3].

In optimal control and reinforcement learning [4], the agent is given a model of the environment and a reward function, and must generate optimal behavior by finding a policy that

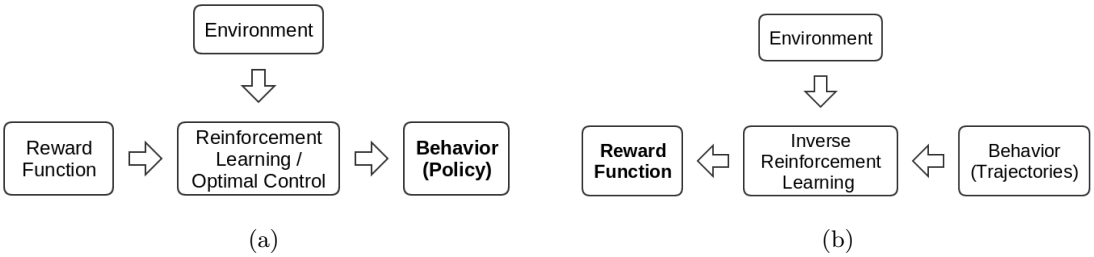


Figure 1: a) The reinforcement learning problem, b) The IRL problem.

maximizes the long-term reward[1]. This is shown in Figure 1(a). In IRL, on the other hand, the agent is given trajectories showing the expert's (optimal) policy together with a model of the environment, and must recover the reward function being optimized. This is shown in Figure 1(b).

Differently from behavioral cloning, IRL attempts to rationalize demonstrated sequential decisions by estimating the utility function that is being maximized by the expert. Since the whole field of reinforcement learning [4] is based on the idea that "the reward function is the most succinct, robust, and transferable definition of the task" [2], its recovery seems wiser than directly learning a mapping from states to actions. The estimated reward function can be successively used to learn the best control policy via classic reinforcement learning. Thus, the main advantage is that, once the reward function is recovered, it is easy to learn a policy that maximizes the expected reward over a long-term horizon, even when demonstrations are

[1]To be precise, in reinforcement learning the agent is not explicitly given these two quantities but has the ability to take actions in the environment and observe the resulting reward.

sub-optimal or the environment the learner is acting in is slightly different than that where the expert is acting in.

We can formalize the IRL problem as follows [1]. Given:

- a model of the environment the expert is acting in (i.e., state-transition dynamics τ), and
- a set of trajectories ζ_i, where each trajectory is a sequence $(s_1, a_1, s_2, ..., a_{T-1}, s_T)$ of states and actions generated by executing the expert's (optimal) policy $\pi^\star$ under τ,

estimate the reward function $R^\star$ being optimized by the expert. More specifically, the problem is reduced to estimating a reward function that makes the demonstrated behavior optimal (i.e., rationalizes such behavior). Formally, the estimated reward function R must satisfy:

$$\mathbb{E}\left[\sum_{t=1}^{T} R(s_t) \mid \tau, \pi^\star\right] \geq \mathbb{E}\left[\sum_{t=1}^{T} R(s_t) \mid \tau, \pi\right] \ \forall \pi \tag{2.20}$$

However, this formulation has several challenges. First, it constitutes an ill-posed problem since it is easy to prove that there exist many solutions (actually, infinitely many) [2]. As an example, a constant reward (e.g., a reward that is always zero) makes every policy optimal. Nevertheless, it is very unlikely that such function matches the one that is sought. Second, it is not possible to explicitly compute the left-hand side since the expert's policy $\pi^\star$ is not given but is demonstrated from sample trajectories. Furthermore, this formulation makes the assumption that the expert is optimal (i.e., $\pi^\star$ is optimal). When this is not the case, the problem becomes infeasible. Last, we can solve the inequalities only by enumerating all possible policies, which is computationally not practical.

Many algorithms have been proposed to tackle such difficulties. We defer the description of the most important ones to the next chapter.

CHAPTER 3

RELATED WORK

In the previous chapter, we introduced and formally defined inverse reinforcement learning, the problem of recovering the unknown reward function of an MDP, while describing its main challenges. This chapter presents some of the state-of-the-art IRL algorithms, focusing on how they tackle such complications. Section 3.1 describes feature matching, one of the first IRL algorithms and whose underlying assumptions represent the foundations of many successive works. Then, section 3.2 describes maximum entropy IRL, which provides a principled way of estimating the demonstrated policy, while section 3.3 presents maximum margin planning, which casts the main problem into a maximum margin one. Finally, section 3.4 introduces the adversarial approach that we extend to build our framework.

3.1 Feature Matching

Abbeel and Ng [3] represent rewards as linear combinations of given features of the states:

$$R(s) = \boldsymbol{w}^{\intercal}\boldsymbol{\phi}(s) \tag{3.1}$$

Given a policy π, they define the feature expectations of π as:

$$\boldsymbol{\mu}(\pi) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \boldsymbol{\phi}(s_t) \mid \pi\right] \tag{3.2}$$

Their main intuition is that, if the feature expectations of a policy π match those of the expert policy $\pi^\star$:

$$\|\boldsymbol{\mu}(\pi) - \boldsymbol{\mu}(\pi^\star)\| \leq \epsilon \tag{3.3}$$

then π is guaranteed to perform as well as $\pi^\star$ for all rewards with $\|\boldsymbol{w}\|_1 \leq 1$.

They propose an iterative procedure that looks for a policy satisfying the condition of Equation 3.3. The algorithm keeps a set of policies $\pi^{(i)}$ together with the corresponding feature expectations $\boldsymbol{\mu}^{(i)}$. At each iteration, the following quadratic program is solved to find the weights $\boldsymbol{w}$ that maximally separates the expert's feature expectations $\boldsymbol{\mu}_E$ from the ones in the above-mentioned set:

$$\max_{\boldsymbol{w}:\|\boldsymbol{w}\|_2 \leq 1} \min_i \boldsymbol{w}^\intercal(\boldsymbol{\mu}_E - \boldsymbol{\mu}^{(i)}) \tag{3.4}$$

Notice that this is equivalent to finding a maximum margin hyperplane separating $\boldsymbol{\mu}_E$ from all $\boldsymbol{\mu}^{(i)}$. Then, the optimal policy for the new weights is computed together with its feature expectations, and the algorithm is iterated until the two sets are separated by a margin less than ϵ. Finally, the output is one of the learned policies, if the demonstrator is optimal, or a mixture of some of them, if the demonstrator is sub-optimal.

Although this algorithm always achieves the feature matching property (which provides a way to solve the degeneracy problem of reward functions), it is not guaranteed to recover a reward that is similar to the true one.

3.2 Maximum Causal Entropy IRL

While Abbeel and Ng's algorithm [3] solves the degeneracy problem by matching the empirical sum of features, which leads to a (mixture) policy that achieves performances very close to those of the demonstrator even without recovering the true reward function, it introduces another ambiguity: many policies, or mixtures of policies, that match the feature expectations exist. No principled way to choose among them is proposed by the authors.

Ziebart et al. [24] [25] solve the above-mentioned ambiguity by employing the principle of maximum causal entropy. Their formulation seeks a probability distribution over actions given states that maximizes the causal entropy while matching the feature expectations:

$$\begin{aligned} \underset{\pi}{\operatorname{argmax}} \quad & H(\boldsymbol{A}_{1:T} \mid\mid \boldsymbol{S}_{1:T}) \\ \text{such that} \quad & \boldsymbol{\mu}(\pi) = \boldsymbol{\mu}_E \end{aligned} \tag{3.5}$$

where $H(\boldsymbol{A}_{1:T} \mid\mid \boldsymbol{S}_{1:T}) = \mathbb{E}[-\log \pi(\boldsymbol{A}_{1:T} \mid\mid \boldsymbol{S}_{1:T})]$ denotes the causal entropy of the distribution π [26]. The authors prove that solving this optimization problem reduces to minimizing the worst-case prediction log-loss and yields a stochastic policy of the form:

$$\pi(a_t \mid s_t) = e^{Q(s_t, a_t) - V(s_t)} \tag{3.6}$$

where:

$$Q(s_t, a_t) = \boldsymbol{w}^\intercal \boldsymbol{\phi}(s_t) + \mathbb{E}_{\tau(.|s_t, a_t)} \left[V(S_{t+1}) \right] \tag{3.7}$$

$$V(s_t) = \underset{a_t}{softmax}\ Q(s_t, a_t) \tag{3.8}$$

and the softmax function is defined as:

$$\underset{x}{softmax}\ f(x) = \log \sum_x e^{f(x)} \tag{3.9}$$

Then, the reward weights achieving such probability distribution can be computed by adopting a convex optimization procedure. The authors show how the resulting algorithm allows for efficient inference and successfully apply it to the problem of modeling route preferences and inferring the destination given partial trajectories.

3.3 Maximum Margin Planning

Ratliff et al. [27] propose a different approach to selecting a policy (and corresponding reward weights) that makes the expert better than all alternatives. They cast the problem into a maximum margin one, where the goal is to find a hyperplane separating the demonstrator's feature expectations from those of any other policy by a structured margin. The resulting formulation is the quadratic program:

$$\begin{aligned} \min_{\boldsymbol{w}} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2 + C\xi \\ \text{s.t.} \quad & \boldsymbol{w}^\intercal \boldsymbol{\mu}(\pi^\star) \geq \boldsymbol{w}^\intercal \boldsymbol{\mu}(\pi) + L(\pi^\star, \pi) - \xi \quad \forall \pi \end{aligned} \tag{3.10}$$

where L denotes a loss function comparing two policies and ξ are slack variables used to soften the margin (whose effect is controlled by constant C). The rationale behind the inclusion of a loss function into the maximum margin is that the latter should be larger for policies that are

very different than the demonstrated one. The authors allow the usage of any loss function, as far as it can be factored into state-action couples. Furthermore, they allow data from different MDPs to be used simultaneously.

One drawback of the quadratic program of Equation 3.10 is that it has a very large number of constraints. In order to make learning practical, the authors solve it using an approximate algorithm based on subgradient methods.

Differently from feature matching and maximum causal entropy IRL, this algorithm does not try to find a policy that achieves the same performance of the expert, but it directly estimates reward weights $\boldsymbol{w}$. The policy used for imitation can successively be computed by optimizing over the learned reward function.

3.4 Adversarial Inverse Optimal Control

Chen et al. [8] propose an adversarial framework for a more general imitation learning and IRL setting, that is, the case where learner and demonstrator are acting in different environments. They consider a demonstrator acting according to the (unknown) policy π under (known) dynamics τ, and a learner acting under different (known) dynamics $\hat{\tau}$. Then, the main idea is to find a policy $\hat{\pi}$ minimizing a loss function comparing learner and demonstrator's state sequences. Formally:

$$\underset{\hat{\pi}}{argmin}\ \mathbb{E}\left[\sum_{t=1}^{T} loss(S_t, \hat{S}_t) \mid \tau, \pi, \hat{\tau}, \hat{\pi}\right] \tag{3.11}$$

However, the demonstrator policy π is not known and the expectation cannot explicitly be computed. In order to provide an estimate of such policy, an adversary is introduced to find the policy $\check{\pi}$ that maximizes such imitative loss. To prevent the adversary from choosing very

bad policies, they consider the set of all stochastic policies that match the empirical sum of features of the demonstrated trajectories [3] and allow $\check{\pi}$ to be picked only from that set. Thus, the idea to deal with the feature-matching ambiguity is, in this case, to choose the worst case (i.e., loss maximizing) policy from the restricted set. The final formulation is a zero-sum game between the learner, looking for the best (stochastic) policy to minimize the loss, and the adversary, trying to maximize such loss by selecting another (stochastic) policy in a constrained way:

$$\min_{\hat{\pi}} \max_{\check{\pi} \in \Theta} \mathbb{E}\left[\sum_{t=1}^{T} loss(\hat{S}_t, \check{S}_t) \mid \tau, \check{\pi}, \hat{\tau}, \hat{\pi}\right] \tag{3.12}$$

where Θ is the set of all policies matching the expert feature expectations:

$$\check{\pi} \in \Theta \Leftrightarrow \mathbb{E}\left[\sum_{t=1}^{T} \boldsymbol{\phi}(\check{S}_t) \mid \tau, \check{\pi}\right] = \boldsymbol{\mu}_E \tag{3.13}$$

In order to solve such problem, they reduce the constrained zero-sum game to one that is parameterized by a vector of Lagrange multipliers and solve it using the double oracle method [28], while optimizing over such parameters using a simple convex optimization procedure [29].

This algorithm provides several advantages over existing methods, which represent the reason why we extend it to solve our generalized problem. First, any loss function can be used (as far as it can be decomposed over states). Second, it provides another principled way to solve the feature matching ambiguity: simply pick the worst case policy. This allows the algorithm to generalize well to new situations. Finally, the algorithm is proven to be Fisher consistent, i.e.,

the learned policy minimizes the loss under the assumption that the feature set is sufficiently rich.

CHAPTER 4

PROBLEM DEFINITION

This chapter formalizes the problem we are trying to solve. Section 4.1 describes the general domain we consider, while section 4.2 formulates the problem we are trying to solve.

4.1 Domain Description

Consider a domain where we have a finite set of states $\mathcal{S}$, a finite set of actions $\mathcal{A}$ that we can take in each state, a probability distribution $p_0(s)$ over initial states, and an unknown reward function $R^\star(s)$ specifying the utility we get after visiting state s. We suppose $R^\star$ can be written as a linear combination of given state features $\boldsymbol{\phi}$ [3]:

$$R^\star(s) = \boldsymbol{w}^\star \cdot \boldsymbol{\phi}(s) \tag{4.1}$$

where $\boldsymbol{\phi}$ and $\boldsymbol{w}^\star$ are K-dimensional vectors, and $\cdot$ denotes the inner product operator.

We consider only finite-horizon processes of length T time steps, thus our policies are non-stationary (i.e., time-variant). The extension to infinite-horizon (discounted) processes is trivial.

Suppose we have a demonstrator acting optimally in the above-mentioned domain according to some unknown deterministic policy $\pi(\boldsymbol{A}_{1:T-1} \mid\mid \boldsymbol{S}_{1:T-1})$ and under transition dynamics $\tau(\boldsymbol{S}_{1:T} \mid\mid \boldsymbol{A}_{1:T-1})$. Here the optimality of π is measured according to the unknown reward function $R^\star$. Furthermore, suppose there is a set of dynamics $\tau_1, \tau_2, ..., \tau_D$ for which we are provided demonstrations of π. This means that the demonstrator executes its policy π under

each dynamics, thus generating a set of sample trajectories $\zeta = \{\zeta_{i,j} \mid i = 1, ..., N, j = 1, ..., D\}$, where each trajectory is the sequence:

$$\zeta_{i,j} = (s_1, a_1, s_2, ..., a_{T-1}, s_T \mid \pi, \tau_j)_i \tag{4.2}$$

On each of these trajectories, the demonstrator obtains a sequence of rewards. Since we defined the reward function as a linear combination of state features, such sequence can be easily derived from the sequence of observed features:

$$\psi_{i,j} = (\boldsymbol{\phi}(s_1), \boldsymbol{\phi}(s_2), ..., \boldsymbol{\phi}(s_T) \mid \pi, \tau_j)_i \tag{4.3}$$

Of fundamental importance for our algorithm is the empirical sum of features observed on a certain trajectory:

$$\tilde{\boldsymbol{c}}_{i,j} = \sum_{\boldsymbol{\phi}(s) \in \psi_{i,j}} \boldsymbol{\phi}(s) \tag{4.4}$$

which allows us to estimate the expert's feature expectations under dynamics τ_j from data as:

$$\tilde{\boldsymbol{c}}_j = \frac{1}{N} \sum_{i=1}^{N} \tilde{\boldsymbol{c}}_{i,j} \tag{4.5}$$

Notice that, since the demonstrator is executing a policy that is optimal for τ under different dynamics, its behavior is likely to be sub-optimal on each τ_j.

4.2 Problem Formulation

Considering the above-mentioned assumptions, our goal is to find a reward function $R(s)$ that "explains" (or better, rationalizes) the demonstrator's behavior. More specifically, we would like the optimal policies that maximize the estimated reward R for each τ_j to perform at least as well as the demonstrator's control policy π when receiving the true reward $R^\star$. Formally:

$$\mathbb{E}[\sum_{t=1}^{T} R^\star(s_t) \mid \tau_j, \pi_j] \geq \mathbb{E}[\sum_{t=1}^{T} R^\star(s_t) \mid \tau_j, \pi] \quad \forall j = 1, ..., D \tag{4.6}$$

where π_j is the policy that maximizes the estimated reward R under dynamics τ_j. Furthermore, we would like the optimal policy that maximizes the estimated reward under τ to have performance close to the demonstrator, so as to imitate the latter.

Notice that we do not require our algorithm to perfectly recover the true reward function $R^\star$ (recall that the IRL problem is intrinsically ill-posed), but we are satisfied with one that allows us to perform as the demonstrator or, possibly, to improve its (sub-optimal) performance.

Since the reward function we consider in Equation 4.1 is represented as a linear combination of given features, the problem reduces to finding the best coefficients $\boldsymbol{w}$ of such combination. Again, we do not require that $\boldsymbol{w} = \boldsymbol{w}^\star$.

CHAPTER 5

ADVERSARIAL FORMULATION

This chapter describes the adversarial formulation we adopt to solve the problem defined in the previous chapter. Section 5.1 shows that our formulation takes the form of a constrained zero-sum game, while Section 5.2 employs results from convex optimization [29] to reduce the latter to a solvable unconstrained problem. Finally, Section 5.3 describes our learning algorithm in detail and analyzes its expected behavior.

5.1 Constrained Zero-Sum Game

We adapt the adversarial formulation described in [8] to handle our generalized case. The learner is still looking for a policy that minimizes some loss with respect to the demonstrator's (unknown) optimal policy π, and the adversary is still providing a constrained worst-case estimate of π. Moreover, we keep the assumption that the loss is represented as a function of the learner and adversary's states. No further restriction is needed. The only fundamental difference is that, since we now need to match features over multiple dynamics, we have more than one constraint specifying the set from which the adversary is allowed to pick its policy. The new formulation is formalized in Definition 5.1.1.

Definition 5.1.1. *The* ***adversarial inverse reinforcement learning with changing dynamics*** *formulation is a constrained zero-sum game where the learner chooses a stochastic*

policy $\hat{\pi}$ that minimizes a given loss function, whereas the adversary chooses a stochastic policy $\check{\pi}$ that maximizes such loss while matching the expected features on all demonstrated dynamics:

$$\min_{\hat{\pi}} \max_{\check{\pi} \in \Theta} \mathbb{E}\left[\sum_{t=1}^{T} loss(\hat{S}_t, \check{S}_t) \mid \tau, \hat{\pi}, \check{\pi}\right] \tag{5.1}$$

where Θ is the set of all stochastic policies matching the empirical expected sum of features (defined in Equation 4.5):

$$\check{\pi} \in \Theta \Leftrightarrow \mathbb{E}\left[\sum_{t=1}^{T} \boldsymbol{\phi}(\check{S}_t) \mid \tau_j, \check{\pi}\right] = \tilde{\boldsymbol{c}}_j \ \forall j = 1, ..., D \tag{5.2}$$

Another difference with respect to the previous formulation is that the expected loss is now computed under the demonstrator's "optimal" dynamics τ for both players instead of having the adversary acting under τ and the learner acting under its own dynamics. Notice that the extension to the latter case is trivial and does not affect the solution we present. Thus, we compute the expected loss only under τ for the sake of simplicity and conciseness.

5.2 Unconstrained Zero-Sum Game

As described in [8], we reduce the constrained zero-sum game of Definition 5.1.1 to an unconstrained zero-sum game by introducing Lagrange multipliers. The main difference is that, since we have multiple constraints, we need a different Lagrange multiplier vector for each of them. Theorem 5.2.1 formulates our new game.

Theorem 5.2.1. *The constrained zero-sum game of Definition 5.1.1 can be reduced to the following optimization problem:*

$$\min_{\boldsymbol{w}_1,\boldsymbol{w}_2,...,\boldsymbol{w}_D} \min_{\hat{\pi}} \max_{\check{\pi}} \mathbb{E}\left[\sum_{t=1}^{T} loss(\hat{S}_t, \check{S}_t) \mid \tau, \hat{\pi}, \check{\pi}\right] + \sum_{j=1}^{D} \boldsymbol{w}_j^{\intercal}\left(\mathbb{E}\left[\sum_{t=1}^{T} \boldsymbol{\phi}(\check{S}_t) \mid \tau_j, \check{\pi}\right] - \tilde{\boldsymbol{c}}_j\right) \quad (5.3)$$

Proof. The Lagrangian function [29] for the constrained maximization problem is:

$$L(\hat{\pi}, \check{\pi}, \boldsymbol{w}_1, \boldsymbol{w}_2, ..., \boldsymbol{w}_D) = \mathbb{E}\left[\sum_{t=1}^{T} loss(\hat{S}_t, \check{S}_t) \mid \tau, \hat{\pi}, \check{\pi}\right] + \sum_{j=1}^{D} \boldsymbol{w}_j^{\intercal}\left(\mathbb{E}\left[\sum_{t=1}^{T} \boldsymbol{\phi}(\check{S}_t) \mid \tau_j, \check{\pi}\right] - \tilde{\boldsymbol{c}}_j\right)$$

Thus, the game of Definition 5.1.1 can be rewritten as:

$$\min_{\hat{\pi}} \max_{\check{\pi}\in\Theta} \mathbb{E}\left[\sum_{t=1}^{T} loss(\hat{S}_t, \check{S}_t) \mid \tau, \hat{\pi}, \check{\pi}\right] = \min_{\hat{\pi}} \max_{\check{\pi}} \min_{w_1,w_2,...,w_D} L(\hat{\pi}, \check{\pi}, \boldsymbol{w}_1, \boldsymbol{w}_2, ..., \boldsymbol{w}_D)$$

Since strong Lagrangian duality and minimax duality hold [30], we are able to consider the innermost minimization as the outermost one, thus concluding the proof. □

5.2.1 Weight Sharing Relaxation

When the optimization problem of Theorem 5.2.1 is solved, the optimal Lagrange multipliers $\boldsymbol{w}_j$ represent the estimated weights of the unknown reward function for dynamics τ_j. Thus, our formulation recovers a possibly different reward function for each demonstrated dynamics. Since our domain considers only one fixed reward for all dynamics, we propose a relaxation that

takes this into account by sharing the same weight between constraints. The new formulation is:

$$\min_{\boldsymbol{w}} \min_{\hat{\pi}} \max_{\check{\pi}} \mathbb{E}\left[\sum_{t=1}^{T} loss(\hat{S}_t, \check{S}_t) \mid \tau, \hat{\pi}, \check{\pi}\right] + \boldsymbol{w}^{\intercal} \sum_{j=1}^{D} \left(\mathbb{E}\left[\sum_{t=1}^{T} \boldsymbol{\phi}(\check{S}_t) \mid \tau_j, \check{\pi}\right] - \tilde{\boldsymbol{c}}_j\right) \tag{5.4}$$

From now on, for the sake of simplicity, we consider only this relaxation. The mathematical results we present are easily extensible to the formulation of Theorem 5.2.1. In chapter 7, we compare the performances of the two.

5.3 Learning Algorithm

We now present the learning algorithm we employ to solve the optimization problem of Equation 5.4. Notice that the same algorithm can be trivially modified to solve the unrelaxed problem of Theorem 5.2.1.

We divide the optimization into two different parts: in the first one, we compute a Nash equilibrium of the inner zero-sum game of Equation 5.4 using the double oracle algorithm [28], while in the second one we optimize over Lagrange multipliers using a convex optimization procedure. The two parts are then combined, in a way such that the first one is used as a subroutine by the second, to solve the whole problem. The next two sections describe these two algorithms in detail.

5.3.1 Double Oracle

We consider finding a Nash equilibrium for the inner zero-sum game of Equation 5.4, that is, we want to solve the optimization problem:

$$\min_{\hat{\pi}} \max_{\check{\pi}} \mathbb{E}\left[\sum_{t=1}^{T} loss(\hat{S}_t, \check{S}_t) \mid \tau, \hat{\pi}, \check{\pi}\right] + \sum_{j=1}^{D} \mathbb{E}\left[\sum_{t=1}^{T} \boldsymbol{w}^{\intercal}\boldsymbol{\phi}(\check{S}_t) \mid \tau_j, \check{\pi}\right] \tag{5.5}$$

Notice that we can remove the empirical sum of feature terms since they are constant values and do not affect the equilibrium strategies.

As described in [8], we consider deterministic policies as the basic strategies of each player and we denote them by letter δ. Then, we represent stochastic policies as mixtures of deterministic ones. Since there are two different definitions of mixture policy, we specify the one we consider in Definition 5.3.1.

Definition 5.3.1. *Given a set $\mathcal{P}$ of N deterministic policies $\{\delta_1, \delta_2, ..., \delta_N\}$ and mixing coefficients $\alpha_1, \alpha_2, ..., \alpha_N$, such that $\alpha_i \geq 0$ and $\sum_{i=1}^{N} \alpha_i = 1$, the mixture policy π of $\mathcal{P}$ is the stochastic policy where, at the first time step, one of the N deterministic policies in $\mathcal{P}$ is chosen with probability given by α_i and then deterministically used throughout the whole process.*

Notice the difference between Definition 5.3.1 and the more general case where mixtures are fully stochastic policies obtained by mixing the deterministic ones at each time step (and not only at the beginning of the process). Although more restrictive, our definition is necessary to allow the computation of expectations under the mixture policy as a linear combination of the expectations under the deterministic policies (which is necessary when we want to use the

mixture policy to match the features). Notice also that there always exists a stochastic policy that achieves the same expectation as our mixture policy, thus we are able to match the features even when the demonstrator's policy is stochastic.

Since the payoff matrix that would result from these assumptions is exponential in the number of actions [8], we use the double oracle method to iteratively build the matrix and find a Nash equilibrium. The algorithm we employ is exactly the one described in [8], thus we do not specify it here. The only big difference is in the two subroutines to compute the best response of each player. We now show how this can be achieved.

Learner's Best Response

Given the adversary's mixed strategy $\check{\pi}$, the learner's best response is the deterministic policy $\hat{\delta}$ given by:

$$BR_{min}(\check{\pi}) = \underset{\hat{\delta}}{argmin}\ \mathbb{E}\left[\sum_{t=1}^{T} loss(\hat{S}_t, \check{S}_t) \mid \tau, \hat{\delta}, \check{\pi}\right] \tag{5.6}$$

This can be solved as a simple optimal control problem by considering an MDP with dynamics τ and reward:

$$R(s_t) = \mathbb{E}\left[-loss(s_t, \check{S}_t) \mid \tau, \check{\pi}\right] \tag{5.7}$$

Notice that the minus sign in front of the loss function is necessary since we are minimizing. Notice also that the feature-matching terms are not controlled by the learner's strategy, thus they are constant values and can be safely omitted. Value iteration can be used to efficiently solve this problem, as it is proposed in [8].

Adversary's Best Response

The computation of the adversary's best response, on the other hand, represents the main difficulty of this work. Given the learner's mixed strategy $\hat{\pi}$, the adversary's best response is the deterministic policy $\check{\delta}$ given by:

$$BR_{max}(\hat{\pi}) = \underset{\check{\delta}}{argmax}\ \mathbb{E}\left[\sum_{t=1}^{T} loss(\hat{S}_t, \check{S}_t) \mid \tau, \hat{\pi}, \check{\delta}\right] + \sum_{j=1}^{D} \mathbb{E}\left[\sum_{t=1}^{T} \boldsymbol{w}^{\intercal}\boldsymbol{\phi}(S_t) \mid \tau_j, \check{\delta}\right] \tag{5.8}$$

This is the problem of finding the optimal deterministic policy that maximizes the total expected reward over multiple MDPs (i.e., multiple different dynamics and reward functions) and is well-known to be NP-Hard [31]. Since the solution of such problem is a big part of this work and should be analyzed in a more general context, its description is deferred to the next chapter. At this point, the reader should simply suppose the existence of a procedure to tractably compute BR_{max}.

Given the two functions to compute best responses, the double oracle algorithm can be easily run to find a Nash equilibrium, that is, mixture policies $\hat{\pi}$ and $\check{\pi}$ solving the optimization problem of Equation 5.5.

5.3.2 Gradient Descent

The second main part of our learning algorithms is used to minimize the inner function of $\boldsymbol{w}$ in Equation 5.4. That is, we want to solve the optimization problem:

$$\underset{\boldsymbol{w}}{min}\ f(\boldsymbol{w}) \tag{5.9}$$

where the function f is defined as:

$$f(\boldsymbol{w}) = \min_{\hat{\pi}} \max_{\check{\pi}} \mathbb{E}\left[\sum_{t=1}^{T} loss(\hat{S}_t, \check{S}_t) \mid \tau, \hat{\pi}, \check{\pi}\right] + \boldsymbol{w}^{\intercal} \sum_{j=1}^{D} \left(\mathbb{E}\left[\sum_{t=1}^{T} \boldsymbol{\phi}(\check{S}_t) \mid \tau_j, \check{\pi}\right] - \tilde{\boldsymbol{c}}_j\right) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$$

Notice that we add a regularization term to the objective function. This is motivated experimentally by the fact that our algorithm tends to recover weights that are much larger in magnitude than the true ones. Thus, adding a small amount of regularization (controlled by parameter λ) helps to keep the magnitude similar to the real one.

Theorem 5.3.1 states a fundamental property of function f that allows us to find a simple algorithm to solve the problem of Equation 5.9.

Theorem 5.3.1. *The function $f(\boldsymbol{w})$ of Equation 5.9 is convex.*

Proof. We start by rewriting f in the more concise form:

$$f(\boldsymbol{w}) = \min_{\hat{\pi}} \max_{\check{\pi}} L(\hat{\pi}, \check{\pi}) + \boldsymbol{w}^{\intercal} \sum_{j=1}^{D} (F(\check{\pi}) - \tilde{\boldsymbol{c}}_j) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2 \tag{5.10}$$

where L and F are, respectively:

$$L(\hat{\pi}, \check{\pi}) = \mathbb{E}\left[\sum_{t=1}^{T} loss(\hat{S}_t, \check{S}_t) \mid \tau, \hat{\pi}, \check{\pi}\right] \tag{5.11}$$

$$F(\check{\pi}) = \mathbb{E}\left[\sum_{t=1}^{T} \boldsymbol{\phi}(\check{S}_t) \mid \tau_j, \check{\pi}\right] \tag{5.12}$$

In order to prove that f is convex, we need to show that [29]:

$$f(\theta \boldsymbol{w}_1 + (1-\theta)\boldsymbol{w_2}) \leq \theta f(\boldsymbol{w}_1) + (1-\theta) f(\boldsymbol{w}_2) \quad \forall \theta \in [0,1], \forall \boldsymbol{w}_1, \forall \boldsymbol{w}_2 \tag{5.13}$$

Since the regularization term is a well-known convex function, we prove the above-mentioned property only for the remaining part of f. Then, convexity follows from the fact that the sum of convex functions is also convex [29]. By expanding Equation 5.13 we obtain:

$$\begin{aligned}
\min_{\hat{\pi}} \max_{\check{\pi}} L(\hat{\pi}, \check{\pi}) + (\theta \boldsymbol{w}_1 + (1-\theta)\boldsymbol{w_2})^{\intercal} \sum_{j=1}^{D} (F(\check{\pi}) - \tilde{\boldsymbol{c}}_j) = \\
\min_{\hat{\pi}} \max_{\check{\pi}} \left\{ L(\hat{\pi}, \check{\pi}) + (\theta \boldsymbol{w}_1 + (1-\theta)\boldsymbol{w_2})^{\intercal} \sum_{j=1}^{D} F(\check{\pi}) \right\} - (\theta \boldsymbol{w}_1 + (1-\theta)\boldsymbol{w_2})^{\intercal} \sum_{j=1}^{D} \tilde{\boldsymbol{c}}_j \leq \\
\theta \min_{\hat{\pi}} \max_{\check{\pi}} L(\hat{\pi}, \check{\pi}) + \boldsymbol{w}_1^{\intercal} \sum_{j=1}^{D} (F(\check{\pi}) - \tilde{\boldsymbol{c}}_j) + (1-\theta) \min_{\hat{\pi}} \max_{\check{\pi}} L(\hat{\pi}, \check{\pi}) + \boldsymbol{w}_2^{\intercal} \sum_{j=1}^{D} (F(\check{\pi}) - \tilde{\boldsymbol{c}}_j) = \\
\min_{\hat{\pi}} \max_{\check{\pi}} \left\{ \theta L(\hat{\pi}, \check{\pi}) + \theta \boldsymbol{w}_1^{\intercal} \sum_{j=1}^{D} F(\check{\pi}) \right\} - \theta \boldsymbol{w}_1^{\intercal} \sum_{j=1}^{D} \tilde{\boldsymbol{c}}_j + \\
\min_{\hat{\pi}} \max_{\check{\pi}} \left\{ (1-\theta) L(\hat{\pi}, \check{\pi}) + (1-\theta) \boldsymbol{w}_2^{\intercal} \sum_{j=1}^{D} F(\check{\pi}) \right\} - (1-\theta) \boldsymbol{w}_2^{\intercal} \sum_{j=1}^{D} \tilde{\boldsymbol{c}}_j = \\
\min_{\hat{\pi}} \max_{\check{\pi}} \left\{ \theta L(\hat{\pi}, \check{\pi}) + \theta \boldsymbol{w}_1^{\intercal} \sum_{j=1}^{D} F(\check{\pi}) \right\} + \min_{\hat{\pi}} \max_{\check{\pi}} \left\{ (1-\theta) L(\hat{\pi}, \check{\pi}) + (1-\theta) \boldsymbol{w}_2^{\intercal} \sum_{j=1}^{D} F(\check{\pi}) \right\} - \\
(\theta \boldsymbol{w}_1 + (1-\theta)\boldsymbol{w_2})^{\intercal} \sum_{j=1}^{D} \tilde{\boldsymbol{c}}_j
\end{aligned}$$

Notice that the empirical sum of feature terms cancel out from both equations. By applying the triangle inequality of the max function, we can now write:

$$\min_{\hat{\pi}} \max_{\check{\pi}} L(\hat{\pi}, \check{\pi}) + (\theta \boldsymbol{w}_1 + (1-\theta)\boldsymbol{w_2})^\intercal \sum_{j=1}^{D} F(\check{\pi}) =$$

$$\min_{\hat{\pi}} \max_{\check{\pi}} \theta L(\hat{\pi}, \check{\pi}) + (1-\theta) L(\hat{\pi}, \check{\pi}) + \theta \boldsymbol{w}_1^\intercal \sum_{j=1}^{D} F(\check{\pi}) + (1-\theta)\boldsymbol{w}_2^\intercal \sum_{j=1}^{D} F(\check{\pi}) \leq$$

$$\min_{\hat{\pi}} \max_{\check{\pi}} \theta L(\hat{\pi}, \check{\pi}) + \theta \boldsymbol{w}_1^\intercal \sum_{j=1}^{D} F(\check{\pi}) + \min_{\hat{\pi}} \max_{\check{\pi}} (1-\theta) L(\hat{\pi}, \check{\pi}) + (1-\theta)\boldsymbol{w}_2^\intercal \sum_{j=1}^{D} F(\check{\pi})$$

which concludes the proof. □

Given the result of Theorem 5.3.1, we can minimize the function $f(\boldsymbol{w})$ using a convex optimization procedure similar to that described in [8]. We employ a batch gradient descent algorithm that uses double oracle to compute the gradient of f with respect to $\boldsymbol{w}$. This is shown in more detail in Table I. Notice that batch gradient descent computes, at each iteration, the average gradient over the whole dataset. In our case, the empirical feature term depends on the data and does not depend on the current weight vector, thus it can be pre-computed to speed up the algorithm. The expected feature term, on the other hand, depends on the current weight vector (through the policy $\check{\pi}$ computed by double oracle) but does not depend on the data, thus it can be computed only once at each iteration (and not once for every trajectory).

In case we want to use the unrelaxed formulation where we have D Lagrange multipliers, we only need to consider an extended weight vector $\boldsymbol{w} = (\boldsymbol{w}_1, \boldsymbol{w}_2, ..., \boldsymbol{w}_D)$. Then, the gradient of f with respect to the extended vector $\boldsymbol{w}$ is simply the vector containing the gradients with

TABLE I: BATCH GRADIENT DESCENT FOR ADVERSARIAL IRL

Batch Gradient Descent
Inputs: learning rate η, convergence threshold ϵ, regularization weight λ **Outputs:** weights $\boldsymbol{w}$
Initialize weights: $\boldsymbol{w} \leftarrow \mathcal{N}(0,1)$ **while** $\frac{1}{K}\|\nabla_{\boldsymbol{w}} f(\boldsymbol{w})\|_1 > \epsilon$ $(\hat{\pi}^\star, \check{\pi}^\star) \leftarrow doubleOracle(\boldsymbol{w})$ $\nabla_{\boldsymbol{w}} f(\boldsymbol{w}) \leftarrow \sum_{j=1}^{D} \mathbb{E}\left[\sum_{t=1}^{T} \boldsymbol{\phi}(S_t) \mid \check{\pi}^\star, \tau_j\right] - \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{D} \tilde{\boldsymbol{c}}_{i,j} + \lambda \boldsymbol{w}$ $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} f(\boldsymbol{w})$ **end while** **return** $\boldsymbol{w}$

respect to the single Lagrange multipliers $\nabla_{\boldsymbol{w}} f(\boldsymbol{w}) = (\nabla_{\boldsymbol{w}_1} f(\boldsymbol{w}_1), \nabla_{\boldsymbol{w}_2} f(\boldsymbol{w}_2), ..., \nabla_{\boldsymbol{w}_D} f(\boldsymbol{w}_D))$.

Each single gradient $\nabla_{\boldsymbol{w}_j} f$ in $\nabla_{\boldsymbol{w}} f(\boldsymbol{w})$ is computed as:

$$\nabla_{\boldsymbol{w}_j} f(\boldsymbol{w}_j) = \mathbb{E}\left[\sum_{t=1}^{T} \boldsymbol{\phi}(S_t) \mid \check{\pi}^\star, \tau_j\right] - \frac{1}{N}\sum_{i=1}^{N} \tilde{\boldsymbol{c}}_{i,j} + \lambda \boldsymbol{w}_j \tag{5.14}$$

No further modification is needed to solve the unrelaxed formulation of the problem.

CHAPTER 6

MULTIPLE-MDP OPTIMIZATION

This chapter presents the main theoretical result of this thesis. As mentioned in Section 5.3.1, a difficult sub-problem arising from our adversarial formulation is the computation of a deterministic policy maximizing the total expected reward over multiple Markov decision processes. Since the solution to such problem is not related to inverse reinforcement learning and can be re-used in any context, we analyze it separately. We start by formally defining the problem we are trying to solve in Section 6.1. Then, we present how we are able to reduce its solution to the optimal control of a partially observable Markov decision process [9] in Section 6.2, and we propose a modified version of point-based value iteration [18] that solves such reduced problem in Section 6.3. Finally, Section 6.4 shows how we can use this result to tractably find the adversary's best response in our adversarial framework.

6.1 Problem Definition

Suppose we are given a set of D Markov decision processes $\mathcal{M} = \{\mathcal{M}_1, \mathcal{M}_2, ..., \mathcal{M}_D\}$, all defined over state space $\mathcal{S}$ and action space $\mathcal{A}$. Each MDP $\mathcal{M}_j \in \mathcal{M}$ is a tuple $< \mathcal{S}, \mathcal{A}, \tau_j, R_j, p_0 >$, where $\tau_j(\boldsymbol{S}_{1:T} \mid\mid \boldsymbol{A}_{1:T-1})$ are the state-transition dynamics, $R_j(S_t, A_t, S_{t+1})$ is the reward function, and $p_0(S)$ is the initial state distribution. Our goal is to find a deterministic policy π that

maximizes the total expected reward from all MDPs in $\mathcal{M}$. Formally, we want to solve the optimization problem:

$$\underset{\pi}{argmax} \sum_{j=1}^{D} \mathbb{E}\left[\sum_{t=1}^{T-1} R_j(S_t, A_t, S_{t+1}) \mid \tau_j, \pi\right] \tag{6.1}$$

Unfortunately, such problem is NP-Hard [31]. It is easily possible to show that the policy achieving the maximum is non-Markovian, i.e., it depends on the whole state-action sequence and not only on the current state. The next section proposes a tractable approximation that makes the policy Markovian by introducing a new continuous variable.

6.2 Approximate Dynamic Programming

The optimization problem of Equation 6.1 is not practical to solve using classic dynamic programming since the optimal policy is non-Markovian. Therefore, we propose an approximate dynamic program that makes the optimal policy Markovian by incorporating knowledge of the state-action sequences into a new continuous variable. We call the latter "belief state", similarly to POMDPs. Our main result is given in Theorem 6.2.1.

Theorem 6.2.1. *The optimization problem of Equation 6.1 can be solved by considering the following dynamic program. We define a "belief state" vector b incorporating knowledge of the transition probabilities from all MDPs in $\mathcal{M}$ as:*

$$\boldsymbol{b}_t \stackrel{def}{=} \frac{1}{\sum_{j=1}^{D} \tau_j(\boldsymbol{s}_{1:t} \mid\mid \boldsymbol{a}_{1:t-1})} \begin{bmatrix} \tau_1(\boldsymbol{s}_{1:t} \mid\mid \boldsymbol{a}_{1:t-1}) \\ \tau_2(\boldsymbol{s}_{1:t} \mid\mid \boldsymbol{a}_{1:t-1}) \\ \vdots \\ \tau_D(\boldsymbol{s}_{1:t} \mid\mid \boldsymbol{a}_{1:t-1}) \end{bmatrix} \tag{6.2}$$

Then, the state-action value and state value functions are, respectively:

$$Q(s_t, a_t, \boldsymbol{b}_t) = \sum_{j=1}^{D} \boldsymbol{b}_{t,j} \sum_{s_{t+1}} \tau_j(s_{t+1} \mid s_t, a_t) \left[R_j(s_t, a_t, s_{t+1}) + V(s_{t+1}, \boldsymbol{b}_{t+1})\right] \tag{6.3}$$

$$V(s_t, \boldsymbol{b}_t) = \max_{a_t} Q(s_t, a_t, \boldsymbol{b}_t) \tag{6.4}$$

The next belief state $\boldsymbol{b}_{t+1}$ can be computed from $\boldsymbol{b}_t$, given s_t, a_t and s_{t+1}, as:

$$\boldsymbol{b}_{t+1} = \frac{\boldsymbol{b}_t}{\sum_{j=1}^{D} \boldsymbol{b}_{t,j} \tau_j(s_{t+1} \mid s_t, a_t)} \odot \begin{bmatrix} \tau_1(s_{t+1} \mid s_t, a_t) \\ \tau_2(s_{t+1} \mid s_t, a_t) \\ \vdots \\ \tau_D(s_{t+1} \mid s_t, a_t) \end{bmatrix} \tag{6.5}$$

where the symbol $\odot$ denotes the point-wise (or element-wise) product of the two vectors. Finally, the optimal deterministic policy is:

$$\pi^\star(s_t, \boldsymbol{b}_t) = \underset{a_t}{argmax}\ Q(s_t, a_t, \boldsymbol{b}_t) \tag{6.6}$$

Proof. Suppose we reach time step $T-1$ after observing state sequence $\boldsymbol{s}_{1:T-1}$ and action sequence $\boldsymbol{a}_{1:T-2}$. It only remains to pick the best last action a_{T-1}, which leads to the final state s_T. The contribution of this last decision to the total expected reward is:

$$\begin{aligned}
\sum_{j=1}^{D} \mathbb{E}\left[R_j(S_{T-1}, A_{T-1}, S_T \mid \tau_j, \pi\right] = \\
\sum_{j=1}^{D} \tau_j(\boldsymbol{s}_{1:T-1} \mid\mid \boldsymbol{a}_{1:T-2})\pi(\boldsymbol{a}_{1:T-2} \mid\mid \boldsymbol{s}_{1:T-2}) \\
\sum_{a_{T-1}} \pi(a_{T-1} \mid \boldsymbol{a}_{1:T-2}, \boldsymbol{s}_{1:T-1}) \sum_{s_T} \tau_j(s_T \mid s_{T-1}, a_{T-1}) R_j(s_{T-1}, a_{T-1}, s_T) = \\
\sum_{a_{T-1}} \pi(\boldsymbol{a}_{1:T-1} \mid\mid \boldsymbol{s}_{1:T-1}) \sum_{j=1}^{D} \tau_j(\boldsymbol{s}_{1:T-1} \mid\mid \boldsymbol{a}_{1:T-2}) \sum_{s_T} \tau_j(s_T \mid s_{T-1}, a_{T-1}) R_j(s_{T-1}, a_{T-1}, s_T)
\end{aligned}$$

We want to find the best action a_{T-1} which maximizes this expectation, namely:

$$\begin{aligned}
\underset{a_{T-1}}{argmax} \sum_{j=1}^{D} \tau_j(\boldsymbol{s}_{1:T-1} \mid\mid \boldsymbol{a}_{1:T-2}) \sum_{s_T} \tau_j(s_T \mid s_{T-1}, a_{T-1}) R_j(s_{T-1}, a_{T-1}, s_T) = \\
\underset{a_{T-1}}{argmax} \frac{\sum_{j=1}^{D} \tau_j(\boldsymbol{s}_{1:T-1} \mid\mid \boldsymbol{a}_{1:T-2}) \sum_{s_T} \tau_j(s_T \mid s_{T-1}, a_{T-1}) R_j(s_{T-1}, a_{T-1}, s_T)}{\sum_{j=1}^{D} \tau_j(\boldsymbol{s}_{1:T-1} \mid\mid \boldsymbol{a}_{1:T-2})} = \\
\underset{a_{T-1}}{argmax} \sum_{j=1}^{D} \boldsymbol{b}_{T-1,j} \sum_{s_T} \tau_j(s_T \mid s_{T-1}, a_{T-1}) R_j(s_{T-1}, a_{T-1}, s_T)
\end{aligned} \tag{6.7}$$

where we define the belief state $\boldsymbol{b}_t$ as:

$$\boldsymbol{b}_t \stackrel{def}{=} \frac{1}{\sum_{j=1}^{D} \tau_j(\boldsymbol{s}_{1:t} \mid\mid \boldsymbol{a}_{1:t-1})} \begin{bmatrix} \tau_1(\boldsymbol{s}_{1:t} \mid\mid \boldsymbol{a}_{1:t-1}) \\ \tau_2(\boldsymbol{s}_{1:t} \mid\mid \boldsymbol{a}_{1:t-1}) \\ \vdots \\ \tau_D(\boldsymbol{s}_{1:t} \mid\mid \boldsymbol{a}_{1:t-1}) \end{bmatrix} = \frac{1}{\sum_{j=1}^{D} \prod_{i=1}^{t} \tau_j(s_i \mid s_{i-1}, a_{i-1})} \begin{bmatrix} \prod_{i=1}^{t} \tau_1(s_i \mid s_{i-1}, a_{i-1}) \\ \prod_{i=1}^{t} \tau_2(s_i \mid s_{i-1}, a_{i-1}) \\ \vdots \\ \prod_{i=1}^{t} \tau_D(s_i \mid s_{i-1}, a_{i-1}) \end{bmatrix} \tag{6.8}$$

Notice that the last equality holds since we are considering only Markovian dynamics. Furthermore, the dependence on the whole state-action sequence has now been removed: the maximization over actions depends only on the current state and belief state. We now define the state-action value function at time step T-1 as the part of Equation 6.7 that is being maximized:

$$Q(s_{T-1}, a_{T-1}, \boldsymbol{b}_{T-1}) = \sum_{j=1}^{D} \boldsymbol{b}_{T-1,j} \sum_{s_T} \tau_j(s_T \mid s_{T-1}, a_{T-1}) R_j(s_{T-1}, a_{T-1}, s_T) \tag{6.9}$$

and the state value function at time step T-1 as:

$$V(s_{T-1}, \boldsymbol{b}_{T-1}) = \max_{a_{T-1}} Q(s_{T-1}, a_{T-1}, \boldsymbol{b}_{T-1}) \tag{6.10}$$

Finally, the computation of the optimal action at time step T-1 can be reformulated in terms of the state-action value function as:

$$\pi^{\star}(s_{T-1}, \boldsymbol{b}_{T-1}) = \underset{a_{T-1}}{argmax}\ Q(s_{T-1}, a_{T-1}, \boldsymbol{b}_{T-1}) \tag{6.11}$$

We now prove the update rule for the belief state. If the current belief state is $\boldsymbol{b}_t$, we take action a_t from state s_t and we end up in state s_{t+1}, the i-th component of the next belief state is:

$$\boldsymbol{b}_{t+1,i} = \frac{\boldsymbol{b}_{t,i}\tau_i(s_{t+1} \mid s_t, a_t)}{\sum_{j=1}^{D} \boldsymbol{b}_{t,j}\tau_j(s_{t+1} \mid s_t, a_t)} \tag{6.12}$$

To see this, we substitute the definition of $\boldsymbol{b}_t$ in Equation 6.12:

$$\begin{aligned}\boldsymbol{b}_{t+1,i} = \frac{\frac{\prod_{k=1}^{t}\tau_i(s_k|s_{k-1},a_{k-1})}{\sum_{l=1}^{D}\prod_{k=1}^{t}\tau_l(s_k|s_{k-1},a_{k-1})}\tau_i(s_{t+1} \mid s_t, a_t)}{\sum_{j=1}^{D}\frac{\prod_{k=1}^{t}\tau_j(s_k|s_{k-1},a_{k-1})}{\sum_{l=1}^{D}\prod_{k=1}^{t}\tau_l(s_k|s_{k-1},a_{k-1})}\tau_j(s_{t+1} \mid s_t, a_t)} &= \\ \frac{\prod_{k=1}^{t}\tau_i(s_k \mid s_{k-1}, a_{k-1})\tau_i(s_{t+1} \mid s_t, a_t)}{\sum_{j=1}^{D}\prod_{k=1}^{t}\tau_j(s_k \mid s_{k-1}, a_{k-1})\tau_j(s_{t+1} \mid s_t, a_t)} &= \\ \frac{\prod_{k=1}^{t+1}\tau_i(s_k \mid s_{k-1}, a_{k-1})}{\sum_{j=1}^{D}\prod_{k=1}^{t+1}\tau_j(s_k \mid s_{k-1}, a_{k-1})}&\end{aligned} \tag{6.13}$$

and we get that the last equation is exactly the i-th component of $\boldsymbol{b}_{t+1}$. We now move to time step T-2. The state-action value function can be easily modified to account for the immediate reward and the value of the next state:

$$Q(s_{T-2}, a_{T-2}, \boldsymbol{b}_{T-2}) = \sum_{j=1}^{D}\boldsymbol{b}_{T-2,j}\sum_{s_{T-1}}\tau_j(s_{T-1} \mid s_{T-2}, a_{T-2})\left[R_j(s_{T-2}, a_{T-2}, s_{T-1}) + V(s_{T-1}, \boldsymbol{b}_{T-1})\right]$$

where the state value function is again:

$$V(s_{T-2}, \boldsymbol{b}_{T-2}) = \max_{a_{T-2}} Q(s_{T-2}, a_{T-2}, \boldsymbol{b}_{T-2}) \tag{6.14}$$

and the optimal action is:

$$\pi^{\star}(s_{T-2}, \boldsymbol{b}_{T-2}) = \operatorname*{argmax}_{a_{T-2}} Q(s_{T-2}, a_{T-2}, \boldsymbol{b}_{T-2}) \tag{6.15}$$

We can now continue this procedure backward up to the first time step, thus concluding the proof. □

Theorem 6.2.1 defines a dynamic program that can be used to solve the optimization problem of Equation 6.1. However, its solution is still not trivial. In order to make everything Markovian, we introduce a continuous variable that prevents us from using any tabular representation of the policy and value functions. The simplest solution is belief discretization: for a D-dimensional belief state vector, we partition the space $\mathbb{R}^D$ into a finite set of hypercubes and discretize the belief over such partition. Then, we are able to solve the dynamic program of Theorem 6.2.1 using a tabular representation. However, the number of discretized belief states necessary to get a satisfactory approximation grows exponentially with the belief dimension D. Thus, this algorithm would be practical only for low-dimension beliefs (at most 3-dimensional). The next section analyzes some properties of this formulation that allow a much more efficient solution.

6.2.1 Dynamic Program Properties

We analyze some properties of the dynamic program introduced in Theorem 6.2.1, focusing on the relation to POMDPs.

Let us start by commenting on the meaning of the belief state. Although the name is derived in analogy to POMDPs, there is no relation between the two definitions. In a POMDP, the belief state is an $|\mathcal{S}|$-dimensional vector whose i-th component represents the probability of the process being in the i-th state. Thus, the belief state vector belongs to the $(|\mathcal{S}|-1)$-dimensional simplex. In our case, the belief state is a D-dimensional vector whose i-th component represents the (normalized) probability of the state-action sequence observed up to the current time under the i-th dynamics. Since our belief is normalized, it belongs to the (D-1)-dimensional simplex.

The other main difference with respect to POMDPs is in the value function V. In our case, V is a function of both the state s_t and belief state $\boldsymbol{b}_t$, and represents the expected future reward that we get starting from s_t, executing the optimal policy $\pi^\star$ and given that the state-transition probabilities are encoded into $\boldsymbol{b}_t$. This is possible since there is no relation between our state and what we call a belief state. In a POMDP, V is a function of the belief state $\boldsymbol{b}$ alone, and represents the expected future reward the agents obtains considering that the unknown current state is distributed according to $\boldsymbol{b}$.

Although many differences exist, our value function has a property that allows us to solve the dynamic program of Theorem 6.2.1 using (modified) POMDP algorithms. Such property is given in Theorem 6.2.2.

Theorem 6.2.2. *The value function defined in Equation 6.4 is piece-wise linear and convex (PWLC) in the belief state* $\boldsymbol{b}_t$.

Proof. We prove the theorem by induction. By replacing Q with its definition, we can write the value function at time step T-1 as:

$$V(s_{T-1}, \boldsymbol{b}_{T-1}) = \max_{a_{T-1}} \sum_{j=1}^{D} \boldsymbol{b}_{T-1,j} \sum_{s_T} \tau_j(s_T \mid s_{T-1}, a_{T-1}) R_j(s_{T-1}, a_{T-1}, s_T) =$$

$$\max_{a_{T-1}} \boldsymbol{b}_{T-1}^{\intercal} \begin{bmatrix} \sum_{s_T} \tau_1(s_T \mid s_{T-1}, a_{T-1}) R_1(s_{T-1}, a_{T-1}, s_T) \\ \sum_{s_T} \tau_2(s_T \mid s_{T-1}, a_{T-1}) R_2(s_{T-1}, a_{T-1}, s_T) \\ \vdots \\ \sum_{s_T} \tau_D(s_T \mid s_{T-1}, a_{T-1}) R_D(s_{T-1}, a_{T-1}, s_T) \end{bmatrix} \tag{6.16}$$

The function that is maximized in Equation 6.16 is clearly linear in $\boldsymbol{b}_{T-1}$ for any choice of state s_{T-1} and action a_{T-1}. Thus, $V(s_{T-1}, \boldsymbol{b}_{T-1})$ is the maximum over a set of linear functions, i.e., it is PWLC.

Suppose now the value function is PWLC at time step t+1. This means that it can be written as a maximization over a set of hyperplanes:

$$V(s_{t+1}, \boldsymbol{b}_{t+1}) = \max_{k} \boldsymbol{b}_{t+1}^{\intercal} \boldsymbol{\alpha}_k(s_{t+1}) \tag{6.17}$$

where $\boldsymbol{\alpha}_k(s_{t+1})$ are the normal directions to such hyperplanes (we call them alpha-vectors in the remaining, again in analogy to POMDPs). We need to prove that this implies that the state value function is PWLC at time step t. This can be written as:

$$V(s_t, \boldsymbol{b}_t) = \max_{a_t} \boldsymbol{b}_t^\intercal \begin{bmatrix} \sum_{s_{t+1}} \tau_1(s_{t+1} \mid s_t, a_t)\Big[R_1(s_t, a_t, s_{t+1}) + V(s_{t+1}, \boldsymbol{b}_{t+1})\Big] \\ \sum_{s_{t+1}} \tau_2(s_{t+1} \mid s_t, a_t)\Big[R_2(s_t, a_t, s_{t+1}) + V(s_{t+1}, \boldsymbol{b}_{t+1})\Big] \\ \vdots \\ \sum_{s_{t+1}} \tau_D(s_{t+1} \mid s_t, a_t)\Big[R_D(s_t, a_t, s_{t+1}) + V(s_{t+1}, \boldsymbol{b}_{t+1})\Big] \end{bmatrix} =$$
$$\max_{a_t} \sum_{s_{t+1}} \left\{ \boldsymbol{b}_t^\intercal \begin{bmatrix} \tau_1(s_{t+1} \mid s_t, a_t) R_1(s_t, a_t, s_{t+1}) \\ \tau_2(s_{t+1} \mid s_t, a_t) R_2(s_t, a_t, s_{t+1}) \\ \vdots \\ \tau_D(s_{t+1} \mid s_t, a_t) R_D(s_t, a_t, s_{t+1}) \end{bmatrix} + V(s_{t+1}, \boldsymbol{b}_{t+1}) \boldsymbol{b}_t^\intercal \begin{bmatrix} \tau_1(s_{t+1} \mid s_t, a_t) \\ \tau_2(s_{t+1} \mid s_t, a_t) \\ \vdots \\ \tau_D(s_{t+1} \mid s_t, a_t) \end{bmatrix} \right\} \tag{6.18}$$

By expanding the definition of $\boldsymbol{b}_{t+1}$ in the PWLC representation of $V(s_{t+1}, \boldsymbol{b}_{t+1})$ given in Equation 6.17, we get:

$$V(s_{t+1}, \boldsymbol{b}_{t+1}) = \frac{1}{\sum_{j=1}^{D} \boldsymbol{b}_{t,j} \tau_j(s_{t+1} \mid s_t, a_t)} \max_k \sum_{j=1}^{D} \boldsymbol{b}_{t,j} \tau_j(s_{t+1} \mid s_t, a_t) \boldsymbol{\alpha}_{k,j}(s_{t+1}) \tag{6.19}$$

If we now substitute this into Equation 6.18, we notice that the normalization term of Equation 6.19 and the last term of Equation 6.18 cancel out. Thus, we have:

$$
\begin{aligned}
V(s_t, \boldsymbol{b}_t) = \max_{a_t} \sum_{s_{t+1}} \left\{ \boldsymbol{b}_t^\intercal \begin{bmatrix} \tau_1(s_{t+1} \mid s_t, a_t) R_1(s_t, a_t, s_{t+1}) \\ \tau_2(s_{t+1} \mid s_t, a_t) R_2(s_t, a_t, s_{t+1}) \\ \vdots \\ \tau_D(s_{t+1} \mid s_t, a_t) R_D(s_t, a_t, s_{t+1}) \end{bmatrix} + \max_k \boldsymbol{b}_t^\intercal \begin{bmatrix} \tau_1(s_{t+1} \mid s_t, a_t) \boldsymbol{\alpha}_{k,1}(s_{t+1}) \\ \tau_2(s_{t+1} \mid s_t, a_t) \boldsymbol{\alpha} k, 2(s_{t+1}) \\ \vdots \\ \tau_D(s_{t+1} \mid s_t, a_t) \boldsymbol{\alpha} k, D(s_{t+1}) \end{bmatrix} \right\} = \\
\max_{a_t} \left\{ \boldsymbol{b}_t^\intercal \begin{bmatrix} \sum_{s_{t+1}} \tau_1(s_{t+1} \mid s_t, a_t) R_1(s_t, a_t, s_{t+1}) \\ \sum_{s_{t+1}} \tau_2(s_{t+1} \mid s_t, a_t) R_2(s_t, a_t, s_{t+1}) \\ \vdots \\ \sum_{s_{t+1}} \tau_D(s_{t+1} \mid s_t, a_t) R_D(s_t, a_t, s_{t+1}) \end{bmatrix} + \sum_{s_{t+1}} \max_k \boldsymbol{b}_t^\intercal \begin{bmatrix} \tau_1(s_{t+1} \mid s_t, a_t) \boldsymbol{\alpha} k, 1(s_{t+1}) \\ \tau_2(s_{t+1} \mid s_t, a_t) \boldsymbol{\alpha} k, 2(s_{t+1}) \\ \vdots \\ \tau_D(s_{t+1} \mid s_t, a_t) \boldsymbol{\alpha} k, D(s_{t+1}) \end{bmatrix} \right\}
\end{aligned}
\tag{6.20}
$$

We see that the function being maximized over actions is the sum of a linear term and a set of $|\mathcal{S}|$ PWLC functions, i.e., it is again a PWLC function of $\boldsymbol{b}_t$. Thus, the maximum over a set of PWLC functions is PWLC. This concludes the proof. □

Theorem 6.2.2 proves a fundamental property: as for POMDPs, our value function is PWLC. This means that the alpha-vectors, the normal directions to the hyperplanes describing the function, are sufficient to fully represent it. Algorithms for solving POMDPs rely on this fact to efficiently approximate the value function and, thus, the optimal policy. Since the PWLC property of V is the only requirement we have to apply such algorithms, we have reduced the solution of our problem to that of a partially observable domain.

Many algorithms have been proposed in the POMDP literature. In this work, we adopt point-based procedures, which approximate the value function by storing a finite set of repre-

sentative belief states and computing one alpha-vector for each of them. Since, as we proved at the beginning of this section, there are a few differences between our dynamic program and that of a POMDP, we need to extend such algorithms to solve our case. The next section shows how we can do this for point-based value iteration [18].

6.3 Modified Point-Based Value Iteration

In the previous section, we proved that the value function defined in our dynamic program is piece-wise linear and convex. This allows the reduction of our problem to the optimal control of a POMDP. We adopt point-based value iteration (PBVI) [18], one of the most common and efficient algorithms for solving POMDPs. However, the differences with respect to our formulation make the algorithm not suitable to be applied as it is. This section describes how we can easily adapt it to our specific problem.

We show the skeleton of our modified algorithm, which is very similar to the original one, in Table II, while we defer the description of the modified sub-routines to the next two sections.

TABLE II: MODIFIED POINT-BASED VALUE ITERATION

modifiedPBVI
Inputs: number of iterations N **Outputs:** value function $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_2, ..., \mathcal{V}_T\}$

Initialize belief set: $\mathcal{B} = \{\boldsymbol{b}_1\}$
for $i = 1$ to N
  $\mathcal{V}_T(s_T) = \emptyset \; \forall s_T \in \mathcal{S}$
  for $t = T - 1$ to 1
  $\mathcal{V}_t = modifiedValueBackup(\mathcal{V}_{t+1}, \mathcal{B})$
  end
  $\mathcal{B} = modifiedBeliefExpansion(\mathcal{B})$
end
return $\mathcal{V}$

Notice immediately two differences with respect to the original algorithms presented in [18]. First, the initial belief state we use to initialize $\mathcal{B}$ is not a distribution over states (as in POMDPs), but is defined as:

$$\boldsymbol{b}_1 = \frac{1}{\sum_{j=1}^{D} p_0(s_1)} \begin{bmatrix} p_0(s_1) \\ p_0(s_1) \\ \vdots \\ p_0(s_1) \end{bmatrix} = \begin{bmatrix} 1/D \\ 1/D \\ \vdots \\ 1/D \end{bmatrix} \tag{6.21}$$

where the last equality holds since we consider all MDPs to have the same initial state distribution. The second main difference is that, since we are considering a finite-horizon problem, our

value function is represented by a different set of alpha-vectors at each time step. Thus, we write $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_2, ..., \mathcal{V}_T\}$ and we run $T-1$ backups before we apply the modified belief expansion. Furthermore, since our value function depends on the state, we have a different set of alpha-vectors for each state and time step. Thus, we write each $\mathcal{V}_t$ as the set $\mathcal{V}_t = \{\mathcal{V}_t(s_t) \mid \forall s_t \in \mathcal{S}\}$. Notice that $\mathcal{V}_T(s_T) = \emptyset \; \forall s_T \in \mathcal{S}$ since there is no action to take at the last time step.

6.3.1 Modified Value Backup

Value backup updates the current value function given the current belief set $\mathcal{B}$ by computing, for each $\boldsymbol{b} \in \mathcal{B}$, the alpha-vector achieving the maximum over actions. Since our value function is very different than that of a POMDP, we need to adapt the original procedure. We use a notation similar to the one used in [18], so that the reader can easily understand the new formulation.

Our goal is to compute the alpha-vectors describing $V(s_t, \boldsymbol{b})$ for each $\boldsymbol{b} \in \mathcal{B}$. We denote the set containing such vectors as $\mathcal{V}_t(s_t)$. Notice that, from now on, we drop the time subscript from $\boldsymbol{b}$ since we are keeping a time-independent set of belief states. We consider the representation of $V(s_t, \boldsymbol{b})$ given in Equation 6.20, which most highlights the different terms forming the alpha-vectors. Then, we define the following quantities, generalizing from the original algorithm [18], for each state $s_t \in \mathcal{S}$:

- $\Gamma_t^{a_t,\star}$: the vector multiplying $\boldsymbol{b}$ in the first linear term of $V(s_t, \boldsymbol{b})$ for action a_t;
- $\Gamma_t^{a_t,s_{t+1}}$: a set of all vectors multiplying $\boldsymbol{b}$ in the inner maximization of $V(s_t, \boldsymbol{b})$ for state s_{t+1} and action a_t;

TABLE III: MODIFIED VALUE BACKUP

modifiedValueBackup
Inputs: value function $\mathcal{V}_{t+1}$, belief set $\mathcal{B}$ **Outputs:** updated value function $\mathcal{V}_t = \{\mathcal{V}_t(s_t) \mid \forall s_t \in \mathcal{S}\}$
foreach $s_t \in \mathcal{S}$ $\Gamma_t^{a_t,\star} = \sum_{s_{t+1}} \boldsymbol{\tau}_\bullet(s_{t+1} \mid s_t, a_t) \odot \boldsymbol{R}_\bullet(s_t, a_t, s_{t+1})$ $\Gamma_t^{a_t,s_{t+1}} = \left\{\boldsymbol{\tau}_\bullet(s_{t+1} \mid s_t, a_t) \odot \boldsymbol{\alpha}(s_{t+1}) \mid \forall \boldsymbol{\alpha} \in \mathcal{V}_{t+1}(s_{t+1})\right\}$ $\Gamma_t^{a_t,\boldsymbol{b}} = \Gamma_t^{a_t,\star} + \sum_{s_{t+1}} \underset{\boldsymbol{\alpha}\in\Gamma_t^{a_t,s_{t+1}}}{argmax}\ \boldsymbol{b}^\intercal \boldsymbol{\alpha}$ $\mathcal{V}_t(s_t) = \left\{\underset{\Gamma_t^{a_t,\boldsymbol{b}},\forall a_t}{argmax}\ \boldsymbol{b}^\intercal \Gamma_t^{a_t,\boldsymbol{b}} \mid \forall \boldsymbol{b} \in \mathcal{B}\right\}$ **end** **return** $\mathcal{V}_t$

- $\Gamma_t^{a_t,\boldsymbol{b}}$: the alpha-vector that is multiplied with $\boldsymbol{b}$ to compute the outer maximum of $V(s_t, \boldsymbol{b})$ for action a_t and belief $\boldsymbol{b} \in \mathcal{B}$.

In order to simplify the notation, we define $\boldsymbol{\tau}_\bullet(s_{t+1} \mid s_t, a_t)$ as the vector containing $\tau_j(s_{t+1} \mid s_t, a_t)$ for each $j = 1, ...D$. We do the same for the reward functions by defining $\boldsymbol{R}_\bullet(s_t, a_t, s_{t+1})$. Given such quantities, the modified algorithm is shown in Table III.

6.3.2 Modified Belief Expansion

Belief expansion updates the current belief set $\mathcal{B}$ by simulating every action and adding only the resulting belief state that is the farthest from $\mathcal{B}$ (if not already present). This is done for each $\boldsymbol{b} \in \mathcal{B}$, thus the current belief set is at most doubled at each expansion. However, our definition of belief state is very different from that of a POMDP. We cannot simply take an

TABLE IV: MODIFIED BELIEF EXPANSION

modifiedBeliefExpansion

Inputs: belief set $\mathcal{B}$
Outputs: expanded belief set $\mathcal{B}'$

Initialize expanded belief set: $\mathcal{B}' = \mathcal{B}$
foreach $\boldsymbol{b} \in \mathcal{B}$
  foreach $s \in \mathcal{S}$
    foreach $a \in \mathcal{A}$
      Sample s'_j according to $\tau_j(\bullet \mid s, a)$ $\forall j = 1, ..., D$
      for $j = 1$ to D
        $\boldsymbol{b}'_{s,a,j} = updateBelief(\boldsymbol{b}, s, a, s'_j)$
        $dist(\boldsymbol{b}'_{s,a,j}) = \min_{\boldsymbol{b}'' \in \mathcal{B}} \|\boldsymbol{b}'_{s,a,j} - \boldsymbol{b}''\|_1$
      end
    end
  end
  if $\underset{\boldsymbol{b}'_{s,a,j}}{argmax}\ dist(\boldsymbol{b}'_{s,a,j}) \notin \mathcal{B}'$
    $\mathcal{B}' = \mathcal{B}' \cup \left\{ \underset{\boldsymbol{b}'_{s,a,j}}{argmax}\ dist(\boldsymbol{b}'_{s,a,j}) \right\}$
  end
end
return $\mathcal{B}'$

action from a certain belief since we also need to know the state. Thus, we modify the original belief expansion to simulate every action from every state, compute the updated belief state according to Equation 6.5, and finally add the new belief that has the maximum distance from $\mathcal{B}$. We define the latter as the minimum L1 norm from a belief in $\mathcal{B}$. Our modified algorithm is shown in Table IV. Notice that we write $updateBelief(\boldsymbol{b}, s, a, s')$ to concisely represent the belief update performed by Equation 6.5.

6.3.3 Performance Analysis

We analyze the performance of our modified algorithm by considering the effect of pruning on the number of alpha-vectors that are computed by the algorithm. We start by computing the total number of (unpruned) alpha-vectors describing the value function. This is given in theorem 6.3.1.

Theorem 6.3.1. *The total number of alpha-vectors in $\mathcal{V}$ describing the value function V is:*

$$|\mathcal{S}| \sum_{t=1}^{T-1} |\mathcal{A}|^{\frac{1-|\mathcal{S}|^{T-t}}{1-|\mathcal{S}|}} \tag{6.22}$$

Proof. In order to prove the theorem, we generalize the procedure for computing the exact set of alpha-vectors for a POMDP, described in [18], to do the same for our specific case. The generalization is trivial: for every state, we compute the sets $\Gamma_t^{a_t,\star}$ and $\Gamma_t^{a_t,s_{t+1}}$, as described in the previous section, and we take their cross-sum for each action and time step.

At time step $T-1$, for each state in $\mathcal{S}$, we have exactly $|\mathcal{A}|$ alpha-vectors. To see this, simply notice that that the set $\Gamma_{T-1}^{a_{T-1},s_T}$ is empty for every state s_{T-1}. Thus, the total number of alpha-vectors at time $T-1$ is $|\mathcal{S}||\mathcal{A}|$. At time step $T-2$, $\Gamma_{T-2}^{a_{T-2},\star}$ contains exactly $|\mathcal{A}|$ vectors, while $\Gamma_{T-2}^{a_{T-2},s_{T-1}}$ contains $|\mathcal{A}||\mathcal{S}||\mathcal{A}|$ vectors, since for every action and state we add all the alpha-vectors at time $T-1$ for that particular state. Thus, taking the cross-sum over states for each action generates a total of $|\mathcal{A}|^{|\mathcal{S}|+1}$ vectors. Since we repeat this for each state, we get a total number of alpha-vectors at time step $T-2$ of $|\mathcal{S}||\mathcal{A}|^{|\mathcal{S}|+1}$. At time step $T-3$, $\Gamma_{T-3}^{a_{T-3},\star}$ contains again $|\mathcal{A}|$ vectors, while $\Gamma_{T-3}^{a_{T-3},s_{T-2}}$ contains $|\mathcal{A}||\mathcal{S}||\mathcal{A}|^{|\mathcal{S}|+1}$. Thus, the cross-

sum generates $|\mathcal{A}|^{|\mathcal{S}|^2+|\mathcal{S}|+1}$ vectors, and the total number at time step $T-3$ is, again, $|\mathcal{S}|$ times that quantity. By proceeding backward, it is easy to see that the number of alpha-vectors at time t is:

$$|\mathcal{S}||\mathcal{A}|^{\sum_{i=0}^{T-1-t}|\mathcal{S}|^i} = |\mathcal{S}||\mathcal{A}|^{\frac{1-|\mathcal{S}|^{T-t}}{1-|\mathcal{S}|}} \tag{6.23}$$

where we have applied the geometric series formula to reduce the exponent. Thus the total number of alpha-vectors is the sum over time:

$$|\mathcal{S}|\sum_{t=1}^{T-1}|\mathcal{A}|^{\frac{1-|\mathcal{S}|^{T-t}}{1-|\mathcal{S}|}} \tag{6.24}$$

This concludes the proof. □

As shown in Theorem 6.3.1, the number of alpha-vectors describing the value function is exponential in the number of states. Theorem 6.3.2 shows how our algorithm limits this number.

Theorem 6.3.2. *The number of alpha-vectors returned by modifiedPBVI after running for I iterations is at most:*

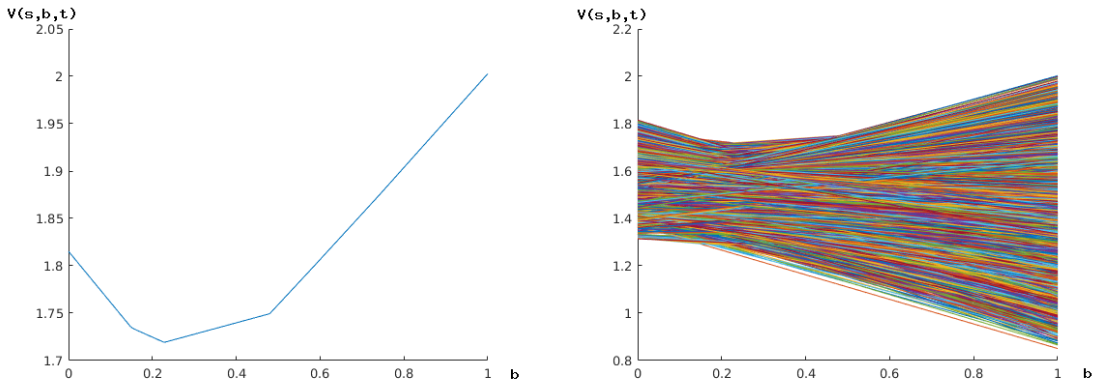
$$|\mathcal{S}|(T-1)2^{I-1} \tag{6.25}$$

Proof. We start by noticing that, if the current belief set contains $|\mathcal{B}|$ beliefs, the number of alpha-vectors computed by *modifiedValueBackup* is at most $|\mathcal{B}|$ for each state and time step. It could be less than that since we prune existing vectors. Thus, the total number of alpha-vectors computed by *modifiedValueBackup* when the belief set has cardinality $|\mathcal{B}|$ is at most $(T-1)|\mathcal{S}||\mathcal{B}|$. Since the belief set is initialized to contain only the initial belief state and at

most doubles at each expansion, its size after I iterations is at most 2^I. Again, it can be less than that since we do not add existing beliefs. At the I-th iteration we perform a value backup on a belief set whose cardinality is, at most, 2^{I-1} (backups are executed before expansions). Thus, the total number of alpha-vectors return cannot be larger than $|\mathcal{S}|(T-1)2^{I-1}$. $\square$

As proven in Theorem 6.3.2, the number of alpha-vectors computed by our generalized PBVI is no more exponential in the number of states. Although it is exponential in the number of iterations, we can freely limit this number. Furthermore, after a good representation of the value function has been found, pruning occurs very frequently, thus only a small number of vectors is added.

We compare *modifiedPBVI* to the modified version of exact alpha-vector computation on a toy problem with two randomly generated MDPs (i.e., the belief state belongs to the 1-dimensional simplex and can be represented by a single scalar). Figure 2 shows the true value function and the set of alpha-vectors composing it for a particular state s and time step t. These were computed by adopting the modified exact procedure. Although the shape of $V(s_t, b)$ is very simple, there were about 16000 alpha-vectors describing it. Figure 3 shows the hyperplanes computed by *modifiedPBVI* (the alpha vectors are their normal directions). We see that the value function can be perfectly represented by only 6 different vectors. All of them are computed by *modifiedPBVI*, without any dominated hyperplane being considered.



(a) (b)

Figure 2: a) The true value function for state s and time step t, b) All hyperplanes describing the true value function.

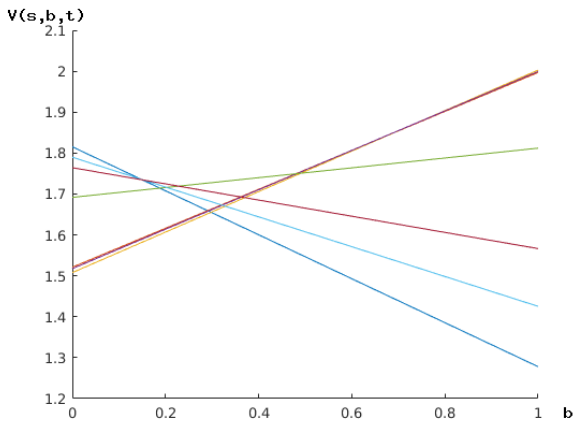


Figure 3: The 6 hyperplanes computed by *modifiedPBVI*.

6.4 Application to Adversarial IRL

We finally show how we can apply the algorithm presented in this chapter to our adversarial formulation. Recall that we need to compute the adversary's best response as:

$$BR_{max}(\hat{\pi}) = \underset{\check{\delta}}{argmax}\ \mathbb{E}\left[\sum_{t=1}^{T} loss(\hat{S}_t, \check{S}_t) \mid \tau, \hat{\pi}, \check{\delta}\right] + \sum_{j=1}^{D} \mathbb{E}\left[\sum_{t=1}^{T} \boldsymbol{w}^{\intercal}\boldsymbol{\phi}(\check{S}_t) \mid \tau_j, \check{\delta}\right] \tag{6.26}$$

It is easy to notice that this problem can be reduced to a multiple-MDP optimization. In fact, we are looking for the optimal deterministic policy $\check{\delta}$ that maximizes the total expected reward from $D+1$ MDPs. The first of such MDPs has dynamics τ and reward defined by:

$$R(s_t) = \mathbb{E}\left[loss(\hat{S}_t, s_t) \mid \tau, \hat{\pi}\right] \tag{6.27}$$

The remaining D MDPs have dynamics τ_j and reward given by the Lagrangian potential terms:

$$R(s_t) = \boldsymbol{w}^{\intercal}\boldsymbol{\phi}(s_t) \tag{6.28}$$

The only thing we are missing is to generalize the reward from a function of the state alone to a function of state,action and next state (as we adopt in this chapter). If we have reward function R and dynamics τ, this is easily done by considering:

$$R(s_t, a_t, s_{t+1}) = \begin{cases} R(s_t) & \text{if } t < T-1 \\ R(s_t) + \tau(s_{t+1} \mid s_t, a_t)R(s_{t+1}) & \text{if } t = T-1 \end{cases} \tag{6.29}$$

All we need to do now is to convert our reward functions as specified in Equation 6.29 and apply $modifiedPBVI$ to get the optimal value function. Then, we can simply obtain the optimal policy for each state s_t and belief $\boldsymbol{b}$ by taking the action associated with the hyperplane achieving the maximum at $\boldsymbol{b}$.

CHAPTER 7

EXPERIMENTAL RESULTS

This chapter shows the experimental results obtained by our adversarial formulation on synthetic experiments where data is randomly generated. We start by specifying the settings for all our experiments in section 7.1. In Section 7.2, we try to recover the reward function of a randomly generated Markov decision process where demonstrations are provided under perturbed dynamics, while in Section 7.3 we try to rationalize an agent navigating through a grid and providing demonstrations from slightly different environments. We show that our algorithm, even though we have no theoretical guarantee, very frequently satisfies the property of Equation 4.6, that is, the learner is able to improve the demonstrator's performance. We discuss and summarize these results in section 7.4.

7.1 Experiment Settings

We start by specifying the parameters considered in our experiments. We generate demonstrations under at most 3 different dynamics. Thus, we run from 7 to 10 PBVI iterations (depending on the particular problem) to compute the adversary's best response, which are enough to well approximate the latter. We consider a decaying learning rate that allows the gradient descent algorithm to converge, and we add a small amount of regularization in all our experiments (from 10^{-4} to 10^{-2}). Since double oracle is generally slow to converge, we consider a threshold for adding strategies to the payoff matrix. Thus, a player's best response is added

to its strategy set only if it provides an improvement to its expected utility greater than the threshold. This allows the double oracle algorithm to converge faster. We adopt the zero-one loss measure in the first experiment, while we adopt the squared loss in the second.

One of the challenges in the implementation is the computation of expectations under the adversary's policy $\check{\pi}$, which is a function of the belief state. This requires a sum over all possible belief states, whose number is exponential in the number of time steps (it roughly corresponds to the number of different trajectories). Thus, such expectations are not practical to compute exactly. In order to provide an approximation, we use Monte Carlo sampling algorithms [32] to estimate them from sample trajectories. In our experiments, we generate from 50000 to 100000 trajectories to approximate the probability $\check{p}_{\tau_j}(S)$ of being in state S when acting according to $\check{\pi}$ and under τ_j, which can successively be adopted to compute all expectations we need in our formulation.

A consequence of the above-mentioned approximations is that our algorithm is no more guaranteed to converge to the same solution when its inputs do not change. In fact, at each iteration of gradient descent, the gradient and objective function are computed by means of the double oracle algorithm, which is no more guaranteed to exactly achieve a Nash equilibrium, due to the threshold we introduce, but only a close approximation. Furthermore, expectations under the adversary's policy $\check{\pi}$ are not computed exactly but only approximated from samples. This motivates us to run our algorithm several times under the same conditions in order to be able to draw conclusions regarding its performance.

7.2 Random MDP

In our first experiment, we randomly generate a Markov decision process and we try to recover its reward function. We start by generating the optimal (Markovian) transition dynamics $\tau(\boldsymbol{S}_{1:T} \mid\mid \boldsymbol{A}_{1:T-1})$ and the reward function in the form specified by Equation 4.1, where we draw the weights $\boldsymbol{w}^\star$ from the K-dimensional Gaussian distribution

$$\boldsymbol{w}^\star \sim \mathcal{N}_K(\underline{\boldsymbol{0}}, \boldsymbol{I}) \tag{7.1}$$

and the features from the uniform distribution:

$$\boldsymbol{\phi}(s) \sim \mathcal{U}_K(0,1) \ \forall s \in \mathcal{S} \tag{7.2}$$

Furthermore, we consider a uniform distribution over initial states. Then, we generate D dynamics by perturbing τ with Gaussian noise, that is, we consider:

$$\tau_j(s_{t+1} \mid s_t, a_t) = \tau(s_{t+1} \mid s_t, a_t) + \mathcal{N}(0, \sigma^2) \ \forall s_{t+1}, s_t, a_t \ \forall j = 1, ..., D \tag{7.3}$$

where σ is the noise standard deviation. Notice that we have to normalize the perturbed dynamics to obtain another probability distribution. Next, we compute the optimal policy $\pi(\boldsymbol{A}_{1:T-1} \mid\mid \boldsymbol{S}_{1:T-1})$ for acting under τ and maximizing the generated reward function. We use such policy to generate N demonstrations under each of the D dynamics, which our algorithm takes as input to estimate the reward function.

We consider an MDP with 20 states, 5 actions and 10 time steps. We show the performance of our algorithm for different number of demonstrations (N), number of features (K), and noise standard deviation (σ). In all cases, we compare the relaxed formulation (weight sharing) of Equation 5.4 to the unrelaxed one (Theorem 5.2.1).

In order to measure the performance of a policy π under certain dynamics τ, we consider the interval $[\bar{r}, \bar{R}]$, where $\bar{r}$ is the worst-case expected reward (i.e., the expectation achieved by a policy minimizing the reward) and $\bar{R}$ is the best-case expected reward (i.e., that achieved by a policy maximizing the reward). Since the expected reward achieved by any policy under τ lies in this interval, we define the performance of π as the percentage of the latter that is achieved. For instance, π achieves 0% performance when the expected reward is $\bar{r}$, while it achieves 50% when the expected reward is $\frac{\bar{r}+\bar{R}}{2}$, and so on.

In order to measure the performance of our algorithm, we compute the policy maximizing the estimated reward under each dynamics τ_j. Then, we compute the above-mentioned performance measure to obtain the percentage of the true reward achieved by such policies.

7.2.1 Optimal Demonstrations

We start by analyzing the simplest case where demonstrations are provided only under the optimal dynamics τ. This is the most common case in IRL, where the demonstrator is optimal and generates trajectories under single dynamics. Notice that, in this case, there is no difference between the relaxed and unrelaxed formulations since we have a single constraint (i.e., a single Lagrange multiplier). We consider 5 features and 100 demonstrations (actually, a much lower value is sufficient to obtain good performance, as it is shown next).

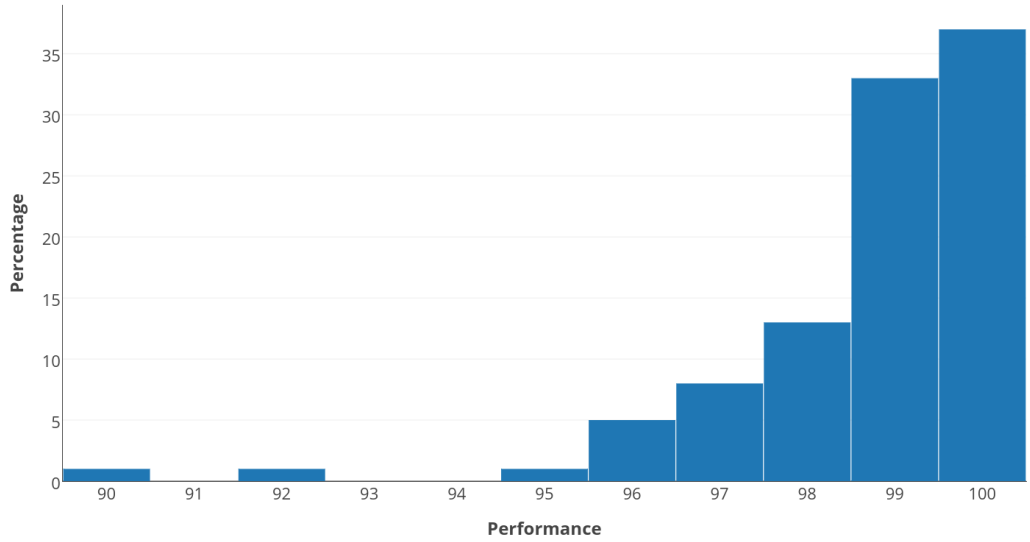


Figure 4: Distribution of performances obtained by multiple runs on single-dynamic optimal demonstrations.

Figure 4 shows a histogram of the performances achieved by our algorithm on 100 different datasets. We can see that every run obtains at least 90% of the maximum expected reward. Furthermore, 70% of the runs obtain more that 99% performance, which constitutes an excellent result. Notice that, in this case, we cannot improve the demonstrator performance since the latter is optimal (i.e., it always achieves 100%). Thus, we are satisfied with performance close to 100% (which our algorithm always obtains in this experiment).

7.2.2 Sub-optimal Demonstrations

We now consider the case where demonstrations are sub-optimal and under changing dynamics. We generate 3 sub-optimal dynamics by perturbing the optimal ones (as specified

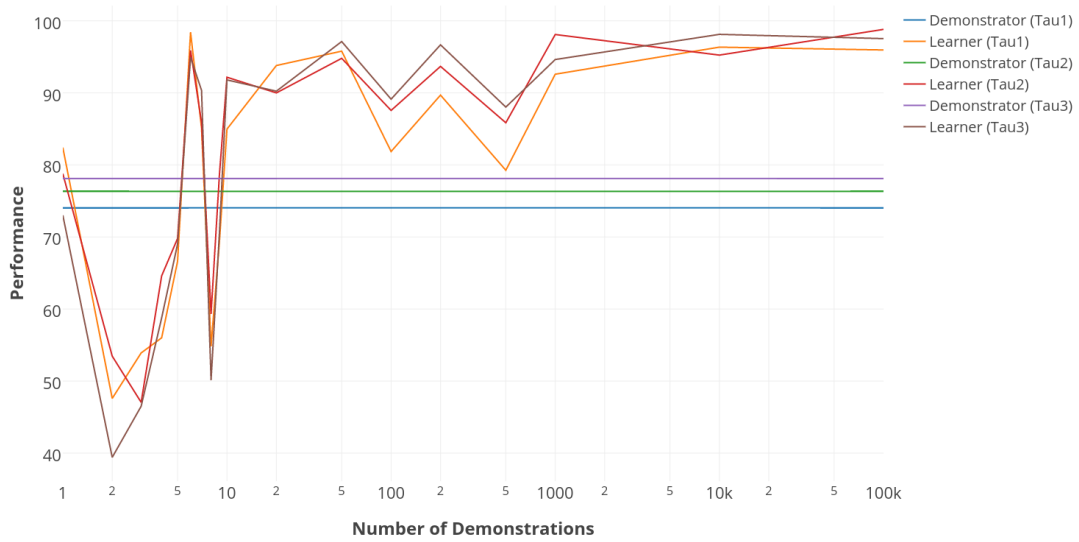


Figure 5: Learner and demonstrator's performance with weight sharing for different numbers of demonstrations.

above) and run our algorithm several times by varying the number of demonstrations, the number of features, and the noise standard deviation.

Number of demonstrations

Figure 5 compares the performances achieved by the learner on the 3 dynamics to those of the demonstrator. We see that, when we have a small number of demonstrations (say less than 10), the performance of our algorithm are subject to a high variance (due to the poor estimate of feature expectations) and are most likely worse that those of the demonstrator. As the number of demonstrations increases, our algorithm is, in this case, always able to perform better that the demonstrator on all sub-optimal dynamics. This means that the recovered reward function

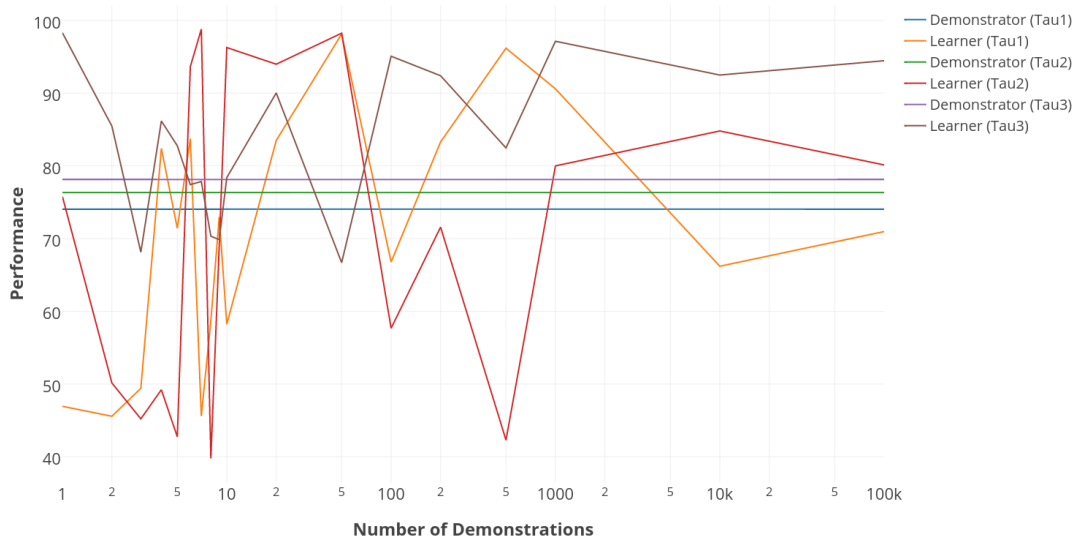


Figure 6: Learner and demonstrator's performance without weight sharing for different numbers of demonstrations.

is really rationalizing the expert's behavior and, when used to train the learner, allows us to improve its sub-optimal performance.

Figure 6 shows the same experiment without weight sharing. We see that now the algorithm is subject to higher variability and the learner's performance is not always better than the demonstrator's as the number of samples increases. We know that the more sub-optimal is the demonstrator, the more likely it is for the learner to achieve poor performance. Our intuition is that weight sharing could "average out" the sub-optimality of the different dynamics, thus leading to better results. However, the fact that in this experiment we are considering different

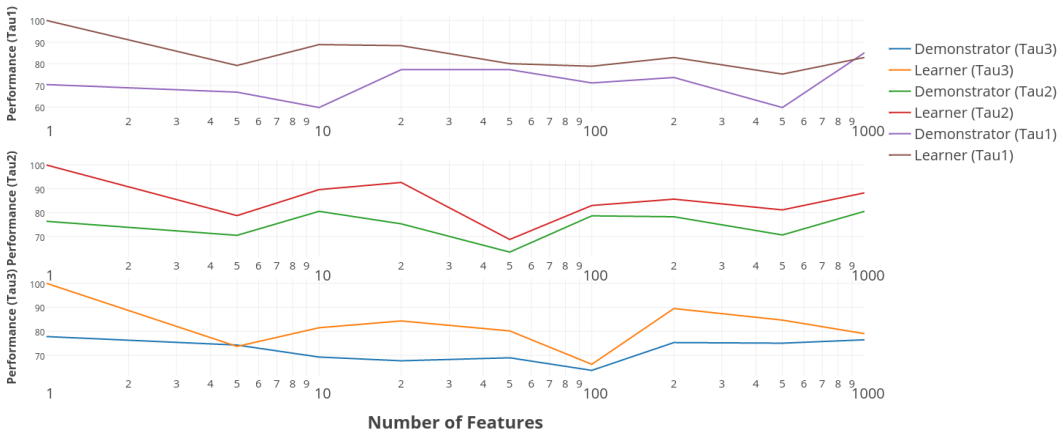


Figure 7: Learner and demonstrator's performance with weight sharing for different numbers of features.

weights can be seen as if we were independently treating each dynamics. Thus, we intuitively expect to get worse results on those dynamics that are more sub-optimal (e.g., τ_1 in Figure 6).

Number of features

We repeat the experiment several times with 3 dynamics, 1000 demonstrations and different numbers of features. Figure 7 shows the result. We see that, due to the high number of demonstrations, the learner always improves the demonstrator's performance. We also notice that the number of features has no particular impact on the algorithm. This is again expected since we are randomly generating the feature vector. Thus, adding one feature should not provide any further information.

When we do not adopt weight sharing, the learner frequently performs worse than the demonstrator, as can be noticed from Figure 8.

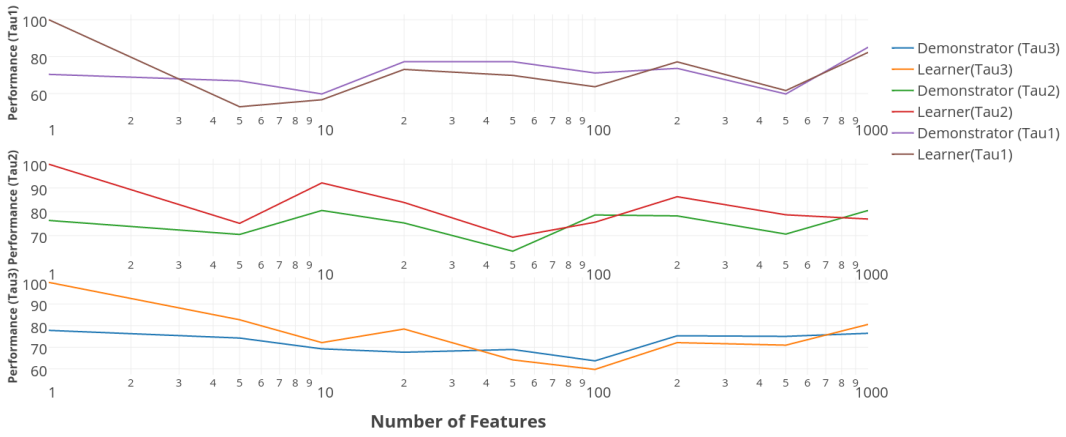


Figure 8: Learner and demonstrator's performance without weight sharing for different numbers of features.

Noise

Finally, we analyze the behavior of our algorithm as the demonstrations' sub-optimality changes. We consider, once again, 3 dynamics, 1000 demonstrations and 5 features. However, we vary the standard deviation of the Gaussian noise in the sub-optimal dynamics. Figure 9 shows the result. We see that, when the noise is very small, the demonstrator remains optimal and the learner achieves almost 100% reward. However, when the noise becomes larger, the demonstrator becomes always more sub-optimal (not monotonically due to the stochasticity in how we generate dynamics) and the learner is always achieving better performance. Again, this is not the case when weight sharing is not adopted. We do not show a plot for the latter case.

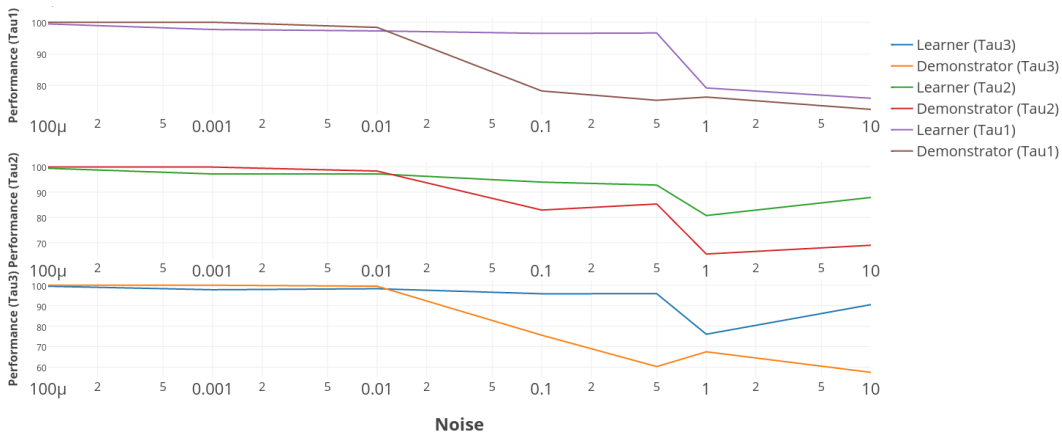


Figure 9: Learner and demonstrator's performance with weight sharing for different values of the noise standard deviation.

7.3 Grid World

Our second experiment aims at showing how the algorithm works on a simple artificial intelligence task. We start by specifying such task in section 7.3.1. Then, we presented the results we obtain under different conditions in the last four sections.

7.3.1 Task Definition

Suppose we have an agent (e.g., a robot) that is moving through a 5x5 grid, as shown in Figure 10(a). Each cell is a state of our environment, and the agent can take four possible actions, namely UP, RIGHT, DOWN and LEFT. We suppose dynamics are stochastic: every time the agent attempts to move to a certain cell, there is always a small probability (which we call noise ϵ) that it falls into one of the adjacent cells instead. As an example, if the agent is

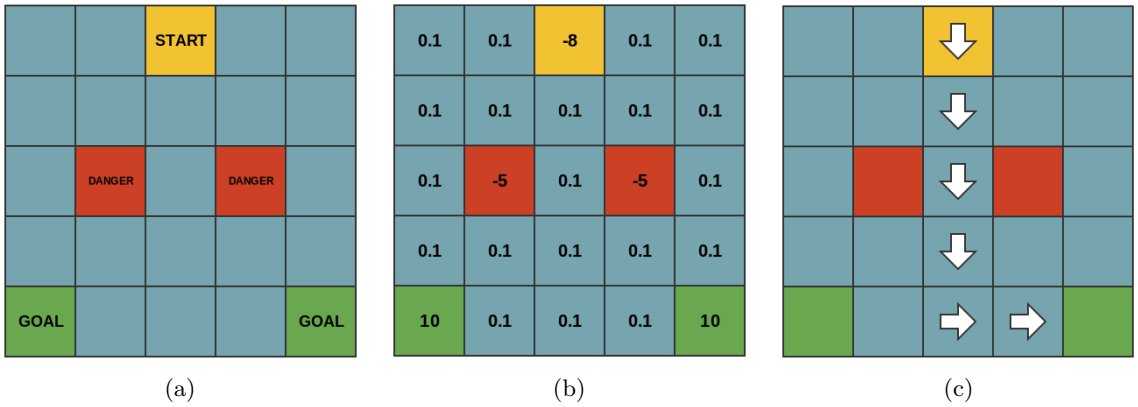


Figure 10: a) The Grid World environment, b) The reward the agent obtains from each state, c) The optimal sequence of actions.

in state $(3,3)$[1] and takes action RIGHT, the next state is $(3,4)$ with probability $1-\epsilon$ or either of $(2,3),(4,3),(3,2)$, each with probability $\frac{\epsilon}{3}$. The robot starts in the cell denoted by START and runs for 7 time step. The task is to reach one of the cells denoted by GOAL. However, the problem is not simple for two reasons. First, the stochasticity in the dynamics could bring the agent into a different cell than the target one, thus making it impossible to reach a goal state in the allowed number of time steps. Second, there are two cells (those denoted by DANGER) that the agent wants to avoid since they might have harmful consequences. Thus, not all paths to the goal states should be considered equally.

[1]We adopt the standard matrix indexing.

In order to specify the above-mentioned task, we need to build a suitable reward function. We start by defining a binary feature vector for each state s as:

$$\boldsymbol{\phi}(s) = \begin{bmatrix} START \\ NORMAL \\ DANGER \\ GOAL \end{bmatrix} \tag{7.4}$$

where each component is either one, if that state has the corresponding property, or zero in the opposite case. For instance, state $(1, 3)$ has feature vector $[1\ 0\ 0\ 0\ ]^T$, since it is the starting state, while state $(3, 3)$ has feature vector $[0\ 1\ 0\ 0\ ]^T$ since it is a normal state. Next, we specify the reward weight to be the vector:

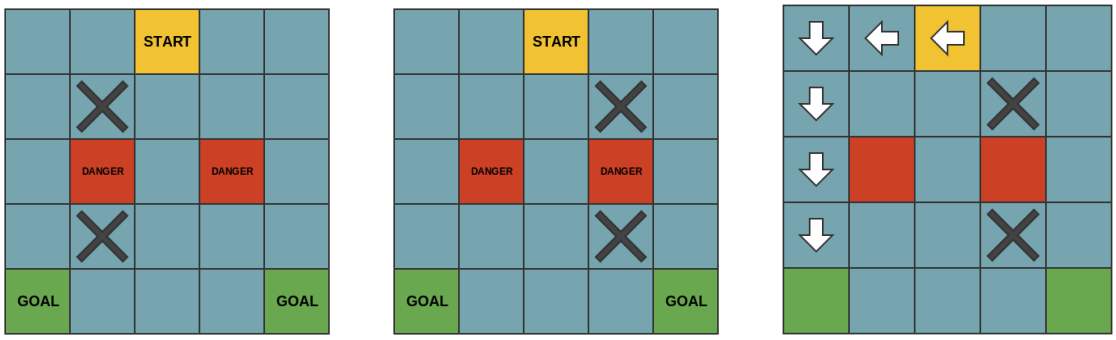
$$\boldsymbol{w}^\star = \begin{bmatrix} -8 \\ 0.1 \\ -5 \\ 10 \end{bmatrix} \tag{7.5}$$

The linear combination of the two leads to the reward function shown in Figure 10(b). We can see that, as expected, the agent obtains the maximum reward from the goal state, while it obtains a negative value from the dangerous zone. We associate a negative reward to the initial state as well since we do not want the agent to go back to the beginning, while any other state provides a small positive reward.

Now that we defined our environment, we specify the IRL settings. We first train our demonstrator to be optimal under the above-mentioned dynamics (we call them τ). This leads to a policy that applies the sequence of actions shown in Figure 10(c). Notice that taking two times LEFT instead of RIGHT as the last two actions leads to the same expected reward. Moreover, notice that the paths reaching the goals by moving through the rightmost (or leftmost) cells have a slightly lower expected reward. This is because the path going through the middle cell has a higher probability of reaching at least one of the two goal states (due to the stochasticity of τ).

Next, we generate demonstrations on two slightly different environments, described by dynamics τ_1 (Figure 11(a)) and τ_2 (Figure 11(b)), respectively. The only difference between the optimal and sub-optimal dynamics is the presence of a new kind of states, those denoted by a cross. Every time the agent moves to one of such states, it is deterministically brought back to the initial state. If this happens, it would be a disastrous situation: the agent will get the negative reward for entering the starting state and it will not be able to reach any goal state anymore (time steps are limited). Due to the stochasticity in the dynamics, the optimal path for τ is now very risky since the agent could accidentally fall into one of those states even when it does not intend to. Thus, the optimal policy for τ is no more optimal for τ_1 and τ_2. Figure 11(c) shows the new optimal path for τ_2. We can see that the agent is now trying to stay as far away from the cross states as possible.

Given demonstrations from the above-mentioned dynamics, our goal is two-fold. First, we want to recover a reward function that sufficiently explains the demonstrator's task, that is,



(a) (b) (c)

Figure 11: a) The first sub-optimal dynamics, b) The second sub-optimal dynamics, c) The optimal path for dynamics τ_2.

we want our estimated weights to be large for the GOAL feature component and small for the DANGER feature component. In this way, we can rationalize the demonstrated behavior as trying to reach a goal state by avoiding the dangerous states. Second, we want to use the estimated reward function to learn how to behave optimally under the demonstrated dynamics. This means that we want our agent to learn to avoid the cross states as much as possible, thus improving the demonstrator's policy.

7.3.2 Asymptotic Performance

We first analyze the asymptotic behavior of our algorithm, i.e., that achieved when infinite many trajectories are shown. In such case, the empirical expected sum of features averaged over the whole dataset converges to the true expected sum of features (the one computed under the demonstrator's unknown policy π). Thus, we consider the latter as the input and we run

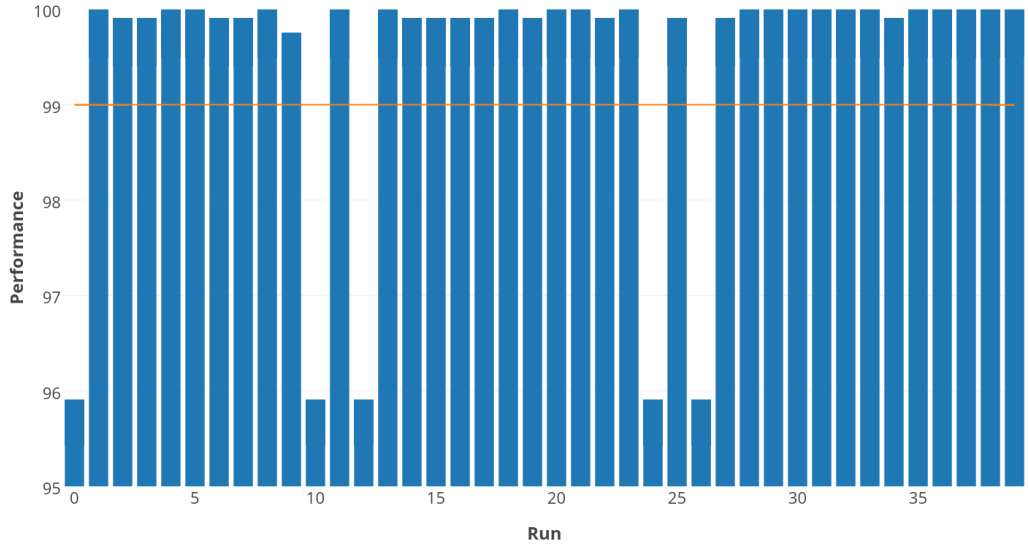


Figure 12: Learner's asymptotic performance on τ_1 and τ_2 for each run. The orange line specifies the optimality threshold.

multiple times the algorithm. The performances achieved by each run are shown in Figure 12. Notice that, due to the symmetry between τ_1 and τ_2, the learner achieves the same performance under both dynamics, thus we show only one. We can see that, besides the five runs where performances are low, the learner always obtains more that 99% of the maximum expected reward. In this example, a score above that threshold is achieved when the policy that safely stays as far away as possible from the cross states is learned. Since we motivate the occasional low-score runs to the stochasticity of our algorithm, we conclude that the asymptotic behavior is correct, that is, the unique global minimum of our objective function leads to a correct reward function.

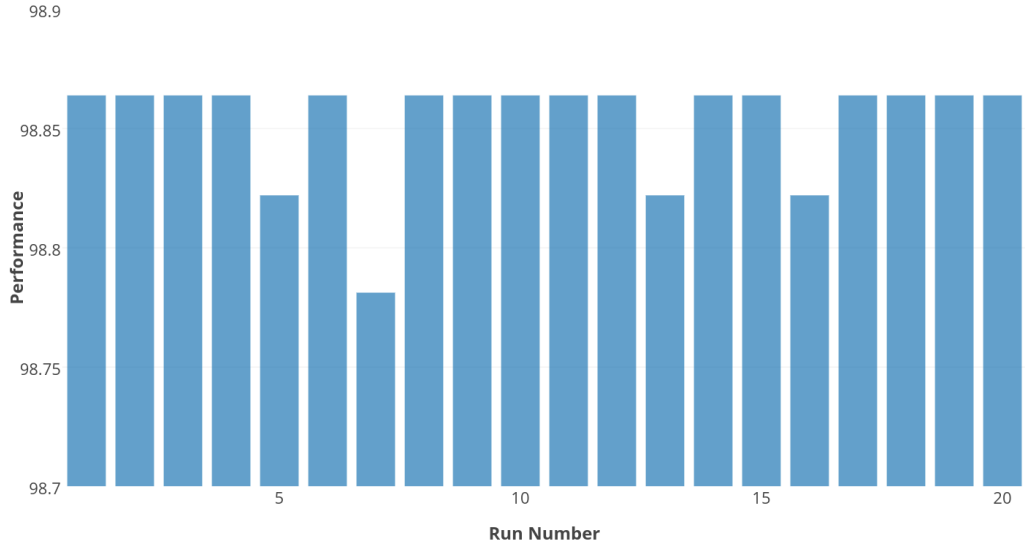


Figure 13: Learner's performance on τ_1 and τ_2 given a single optimal trajectory.

7.3.3 Single Optimal Trajectory

We now focus on a simple case where our algorithm fails at computing the optimal solution. Suppose a single demonstration is given showing the trajectory of Figure 10(c), i.e., the optimal path under dynamics τ. This means that we consider the sum of features [1 5 0 1] as the only input. Figure 13 shows the result of several runs under this condition. Although performances are still high, no run achieves more than 99%, that is, no run converges to reward weights leading to the desired safe policy. These is due to the fact that the feature sum of the optimal path is very different than the true expected feature sum under the demonstrated dynamics (that considered in the previous section). Thus, our algorithm estimates a reward function

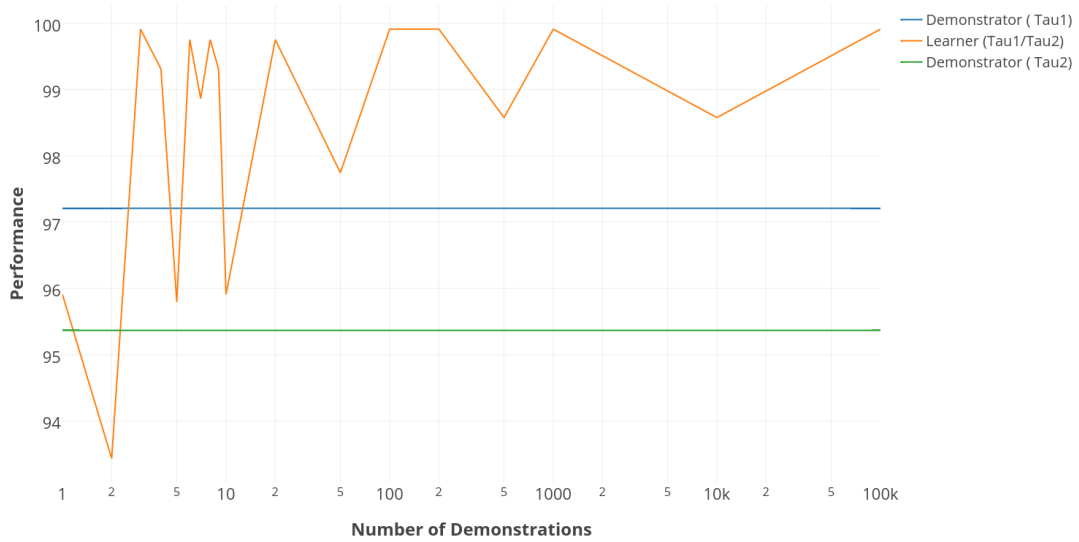


Figure 14: Demonstrator and learner's performance as the number of demonstrations varies.

that leads to a policy following such demonstrated path, which is sub-optimal under τ_1 and τ_2. This example shows the importance of having enough trajectories to capture the expert's sub-optimality under the new environments.

7.3.4 Multiple Demonstrations

We now investigate the behavior of our algorithm as the number of demonstrations changes. As already proved, the algorithm very frequently achieves perfect performance when infinite many trajectories are provided (i.e., the empirical sum of features matches the true expected sum of features).

Figure 14 compares the demonstrator and learner's performance as the number of demonstrations increases. Notice that there is only one curve for the learner since it always achieves

the same performance on both environments (due to their symmetry). Once again, we can see that the final result is subject to high variance when the number of demonstrations is low. However, the learner clearly outperforms the demonstrator when the latter is sufficiently increased. As already mentioned, a score greater than 99 is achieved when the policy that safely stays as far away as possible from the cross states is learned. In this example, this is often the case. Overall, the learner's performance is always very close to the demonstrator's, and most of the time even better.

We now take a look at the learned reward weights for the experiment under consideration and compare them to the true ones. Intuitively, if the estimated reward leads to a nearly optimal policy (where optimality is evaluated on the true reward function), the estimated weights' components should be, at least, in an order similar to the true ones'. Thus, the true weight vector and the estimated one should be almost parallel (i.e., they should have similar direction). This motivates a comparison based on the cosine similarity, which computes the cosine of the angle between two vectors. The result for the current experiment is shown in Figure 15. We can see that the cosine similarity is always greater than 0.6, which means that the estimated vectors are always in a direction similar to the true vector's (when the similarity takes value 1, the vectors are exactly parallel). Thus, we conclude that, at least for this experiment, our algorithm always provides a good estimate of the true reward function.

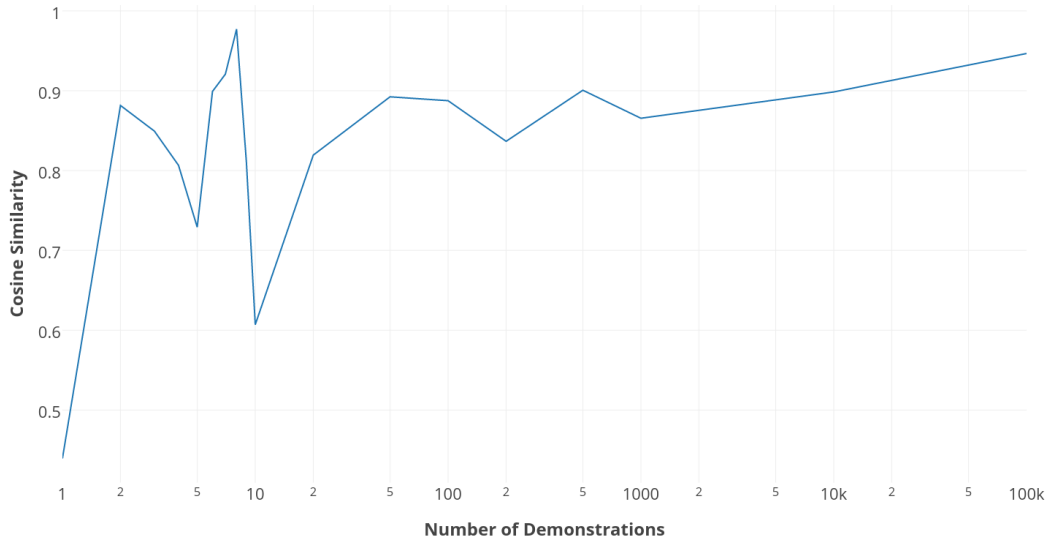
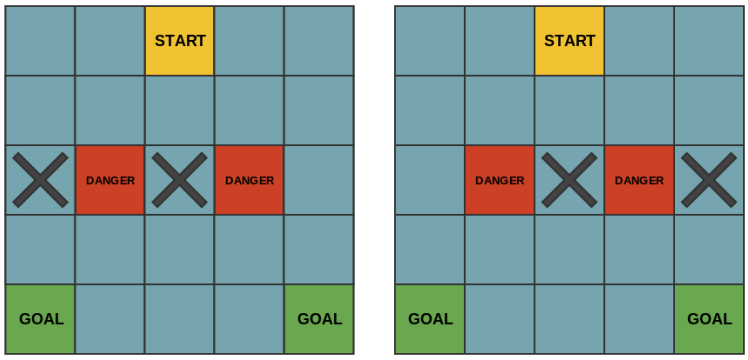


Figure 15: Cosine similarity between the true reward weights and the learned weights.

7.3.5 Highly Sub-optimal Demonstrations

Finally, we analyze a case where highly sub-optimal demonstrations are provided. We consider trajectories under two different environments than those considered above. These are shown in figure 16(a) and 16(b).

We can see that, in both environments, the optimal path followed by the expert, together with one of the alternative paths, is now blocked by a cross state. This means that most of the trajectories will not reach the goal state, which will be demonstrated in only a small number of cases. Thus, we expect our algorithm to infer that such state provides a low reward, giving a small weight to the corresponding feature. Figure 17 shows the performances achieved by 20



(a) (b)

Figure 16: a) The first highly sub-optimal dynamics, b) The second highly sub-optimal dynamics.

different runs. We can see that in only 4 runs the algorithm learns an optimal reward function. Notice however that the overall performances are rather high. This is due to the fact that the estimated weights lead the agent to learn a policy that stays in the normal states as much as possible, thus achieving a positive reward. Furthermore, the demonstrator achieves 88% of the total reward in both environments, thus the learner is always able to improve its performance.

To verify that the estimated reward is indeed very different than the desired one, we plot the cosine similarity between the true and learned weights for each run (Figure 18). We can see that, in many cases, the cosine similarity is negative, i.e., the two vectors have an angle greater than 90 degrees. In our case, this is due to the fact the fourth component of the estimated vector (the goal feature weight) has a different sign than the true one. As expected, our algorithm gives a negative reward to the goal state, and the learned policy tries to avoid such state as

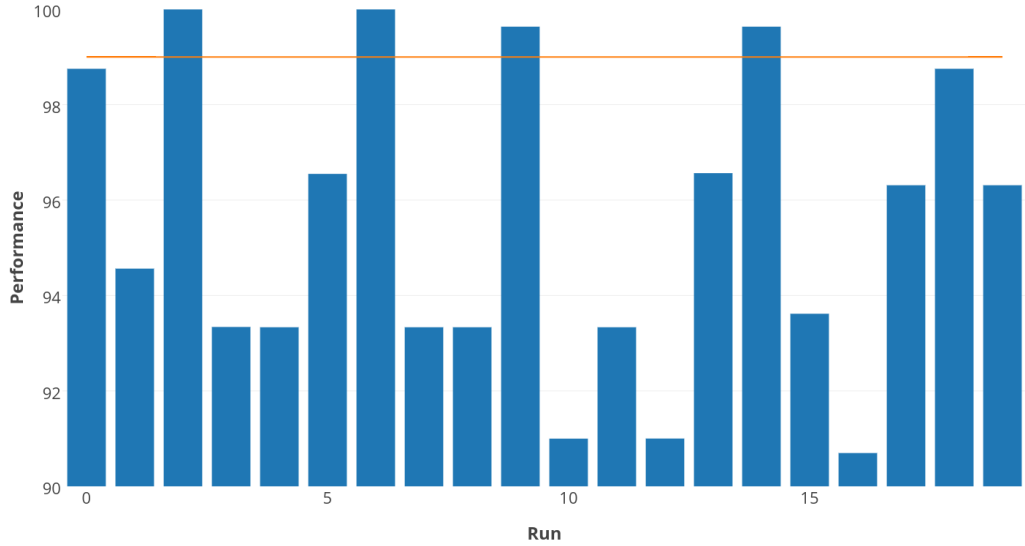


Figure 17: Learner's performance on highly sub-optimal dynamics. The orange line shows the optimality threshold.

much as possible. However, we conclude that, even in this complicated task, our algorithm behaves rather well.

7.4 Discussion

In both experiments, under the assumption that we have sufficient demonstrations, rich features, and nearly-optimal trajectories, our algorithm is able to recover a reward function that rationalizes the expert's behavior. In particular, the optimal policy for maximizing the estimated reward achieves performances very close to the demonstrator's when the latter is providing optimal trajectories. Furthermore, when only sub-optimal trajectories are observed, the learner is generally able to improve the expert's behavior.

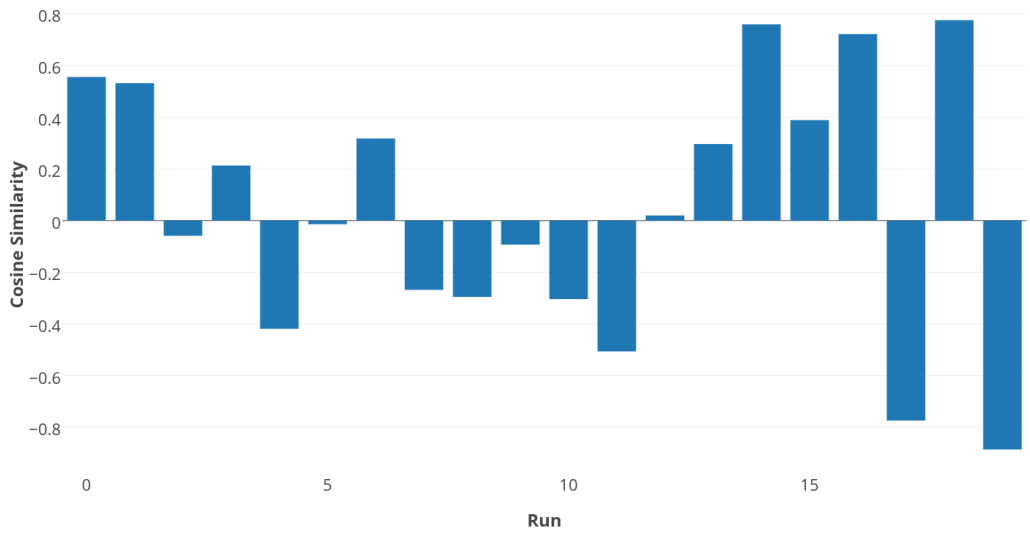


Figure 18: Cosine similarity between true and learned weights on highly sub-optimal dynamics.

The main bottleneck of our algorithm is time complexity. This is mostly due to the unpredictable behavior of double oracle. In fact, the latter iteratively builds the payoff matrix to find a Nash equilibrium. However, there is no guarantee on the number of iterations until convergence, and each iteration runs both point-based value iteration, to find the adversary's best response, and Monte Carlo sampling, to approximate the adversary's state probabilities. Furthermore, double oracle is executed once for each iteration of gradient descent. All these factors combined lead to a very time-consuming procedure which allows the algorithm to solve only rather small problems. Nevertheless, the parameters we adopt in our experiments are much bigger than those needed to achieve good approximations, thus we can always limit their values to solve larger problems.

CHAPTER 8

CONCLUSION AND FUTURE WORK

In this thesis, we analyzed inverse reinforcement learning with changing dynamics. This is the case where sub-optimal trajectories are demonstrated under several dynamics, generally different than those for which the expert is optimal. To our knowledge, no existing algorithm is able to solve such problem. We motivated the need for an efficient solution by providing examples of possible IRL scenarios, from learning under covariate shift to demonstrated risk-averse or robust behavior.

Leveraging on existing methods, we formulated the problem in an adversarial manner; we considered a zero-sum game between a learner attempting to minimize a loss with respect to the unknown expert's policy and an adversary attempting to provide a loss-maximizing estimate of such policy. Furthermore, we constrained the adversary to pick only policies that match the expert's feature expectations under each dynamics. We reduced the constrained zero-sum game to a free one by introducing Lagrange multipliers, and we solved it by employing a simple convex optimization procedure.

We proved that, in order to solve our formulation, we need to compute the deterministic policy maximizing the total reward from different MDPs, which is an NP-Hard problem. We proposed a tractable approximation by reducing the latter to the optimal control of POMDPs. We defined an approximate dynamic program by introducing a continuous variable incorporating knowledge of the state-transition probabilities from each MDP. We proved that the resulting

value function is piece-wise linear and convex and we proposed a modified version of point-based value iteration to approximate it.

We showed the performance of our algorithm on two synthetic data experiments. In the first one, we tried to recover the unknown reward function of a randomly generated MDP. We proved that the algorithm is able to achieve a performance very close to that of the demonstrator when the latter is generating optimal trajectories, and to improve the expert's performance when the latter is sub-optimal under the demonstrated dynamics. In the second experiment, we analyzed an agent navigating through a grid and trying to reach certain goals while avoiding dangerous zones. We provided demonstrations in environments with new dangerous zones and we proved that the recovered reward function is such that, rather frequently, the inferred optimal policy makes the learner avoid such new zones. We concluded that, in both experiments, the algorithm is able to estimate a reward function that rationalizes the expert's behavior. However, we indicated that its temporal complexity allows only rather small problems to be solved.

As a future work, we intend to adopt our algorithm to solve real-world problems. In particular, we will first reduce the time requirements by implementing more efficient point-based and Monte Carlo algorithms. Then, we will try to formulate the problem in such a way that double oracle is not required to compute a solution. We argue that such new formulation would alleviate the time complexity bottleneck, thus allowing our algorithm to solve large-scale problems. Finally, we will focus on real-world tasks, especially on inverse reinforcement learning under risk aversion.

CITED LITERATURE

1. Russell, S.: Learning agents for uncertain environments. In Proceedings of the eleventh annual conference on Computational learning theory, pages 101–103. ACM, 1998.

2. Ng, A. Y. and Russell, S. J.: Algorithms for inverse reinforcement learning. In Proceedings of the Seventeenth International Conference on Machine Learning, ICML '00, pages 663–670, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

3. Abbeel, P. and Ng, A. Y.: Apprenticeship learning via inverse reinforcement learning. In Proceedings of the Twenty-first International Conference on Machine Learning, ICML '04, pages 1–, New York, NY, USA, 2004. ACM.

4. Sutton, R. S. and Barto, A. G.: Introduction to Reinforcement Learning. Cambridge, MA, USA, MIT Press, 1st edition, 1998.

5. Niv, Y.: Reinforcement learning in the brain. Journal of Mathematical Psychology, 53(3):139–154, 2009.

6. Watkins, C. J. C. H.: Learning from Delayed Rewards. Doctoral dissertation, King's College, Cambridge, UK, May 1989.

7. Montague, P. R., Dayan, P., Person, C., and Sejnowski, T. J.: Bee foraging in uncertain environments using predictive hebbian learning. Nature, 377(6551):725, 1995.

8. Chen, X., Monfort, M., Ziebart, B. D., and Carr, P.: Adversarial inverse optimal control for general imitation learning losses and embodiment transfer. In Proceedings of the Thirty-Second Conference on Uncertainty in Artificial Intelligence, UAI'16, pages 102–111, Arlington, Virginia, United States, 2016. AUAI Press.

9. Sondik, E. J.: The optimal control of partially observable markov processes over the infinite horizon: Discounted costs. Oper. Res., 26(2):282–304, April 1978.

10. Cassandra, A. R., Kaelbling, L. P., and Littman, M. L.: Acting optimally in partially observable stochastic domains. 1994.

CITED LITERATURE (continued)

11. Zadrozny, B.: Learning and evaluating classifiers under sample selection bias. In Proceedings of the twenty-first international conference on Machine learning, page 114. ACM, 2004.

12. Shimodaira, H.: Improving predictive inference under covariate shift by weighting the log-likelihood function. Journal of statistical planning and inference, 90(2):227–244, 2000.

13. Ruszczyński, A.: Risk-averse dynamic programming for markov decision processes. Mathematical programming, 125(2):235–261, 2010.

14. Wiesemann, W., Kuhn, D., and Rustem, B.: Robust markov decision processes. Mathematics of Operations Research, 38(1):153–183, 2013.

15. Puterman, M. L.: Markov decision processes: discrete stochastic dynamic programming. John Wiley & Sons, 2014.

16. Bellman, R.: Dynamic Programming. Princeton, NJ, USA, Princeton University Press, 1 edition, 1957.

17. Sigaud, O. and Buffet, O.: Markov Decision Processes in Artificial Intelligence. Wiley-IEEE Press, 2010.

18. Pineau, J., Gordon, G., and Thrun, S.: Point-based value iteration: An anytime algorithm for pomdps. In Proceedings of the 18th International Joint Conference on Artificial Intelligence, IJCAI'03, pages 1025–1030, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

19. Massey, J.: Causality, feedback and directed information. Citeseer.

20. Bain, M. and Sammut, C.: A framework for behavioural cloning. 2001.

21. Pomerleau, D. A.: Alvinn, an autonomous land vehicle in a neural network. Technical report.

22. LeCun, Y., Muller, U., Ben, J., Cosatto, E., and Flepp, B.: Off-road obstacle avoidance through end-to-end learning.

CITED LITERATURE (continued)

23. Sammut, C., Hurst, S., Kedzier, D., and Michie, D.: Learning to fly. In In Proceedings of the Ninth International Conference on Machine Learning, pages 385–393. Morgan Kaufmann, 1992.

24. Ziebart, B. D., Maas, A. L., Bagnell, J. A., and Dey, A. K.: Maximum entropy inverse reinforcement learning. 2008.

25. Ziebart, B. D., Bagnell, J. A., and Dey, A. K.: Modeling interaction via the principle of maximum causal entropy. 2010.

26. Kramer, G.: Directed information for channels with feedback. Doctoral dissertation, University of Manitoba, Canada, 1998.

27. Ratliff, N. D., Bagnell, J. A., and Zinkevich, M. A.: Maximum margin planning. In Proceedings of the 23rd International Conference on Machine Learning, ICML '06, pages 729–736, New York, NY, USA, 2006. ACM.

28. McMahan, H. B., Gordon, G. J., and Blum, A.: Planning in the presence of cost functions controlled by an adversary. In Proceedings of the Twentieth International Conference on International Conference on Machine Learning, ICML'03, pages 536–543. AAAI Press, 2003.

29. Boyd, S. and Vandenberghe, L.: Convex Optimization. New York, NY, USA, Cambridge University Press, 2004.

30. Asif, K., Xing, W., Behpour, S., and Ziebart, B. D.: Adversarial cost-sensitive classification. In Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence, UAI'15, pages 92–101, Arlington, Virginia, United States, 2015. AUAI Press.

31. Vlassis, N., Littman, M. L., and Barber, D.: On the computational complexity of stochastic controller optimization in pomdps. ACM Transactions on Computation Theory (TOCT), 4(4):12, 2012.

32. Hastings, W. K.: Monte carlo sampling methods using markov chains and their applications. Biometrika, 57(1):97–109, 1970.

33. Atkeson, C. G. and Schaal, S.: Robot learning from demonstration.

VITA

Andrea Tirinzoni

Personal Data

Address:	809 South Damen Avenue, 60612 IL, Chicago
Phone:	+1 (312) 730 8322
email:	andrea.tirinzoni@gmail.com
Date of Birth:	July 18TH, 1993
Nationality:	Italian

Education

Today — Mar 2016
Master of Science in Computer Science,
University of Illinois at Chicago, Chicago
Gpa: 4.0/4

Today — Oct 2015
Master of Science in Computer Science and Engineering,
Politecnico di Milano, Milan
Gpa: 29.88/30

Jul 2015
Bachelor's Degree in Computer Engineering,
Politecnico di Milano, Milan
110/110 *summa cum laude*

Research Experience

Today — Jan 2017
Research Assistant
Purposeful Prediction Laboratory, University of Illinois at Chicago
Master thesis research on Adversarial Inverse Reinforcement Learning.
Research on Robust Control of Markov Decision Processes.
Advisor: Prof. Brian Ziebart.

VITA (continued)

Projects

Sep 2016 – Dec 2016

Nanoscribe 3D Printer Advanced Alignment System

Development of a computer vision application to automatically align the Nanoscribe 3D Printer using camera snapshots.

Advisor: Prof. Daniela Radakovic.

Sep 2016 – Dec 2016

Analysis of software bugs using GitHub APIs

Development of a machine learning tool to find patterns in bug corrections using git patches and GitHub metadata files.

Advisor: Prof. Mark Grechanik.

June 2016 – Mar 2016

Apache Spark Monitoring

Development of a module for extending Apache Spark's monitoring capabilities.

Advisor: Prof. Marco D. Santambrogio.

Scholarships

Aug 2016	Italian scholarship for the best UIC student in computer science
May 2016	Scholarship for engineering students with high GPA
Mar 2016	Exemption from tuition fees for high academic performance
Mar 2015	Exemption from tuition fees for high academic performance
May 2014	Scholarship for engineering students with high GPA
Mar 2014	Exemption from tuition fees for high academic performance

Languages

Italian:	Mothertongue
English:	Fluent (TOEFL - Score: 100/120)

Computer Skills

Basic Knowledge:	ASP.NET,Processing,LUA,Lisp,Torch,VHDL
Intermediate Knowledge:	C++,R,PHP,Python,Javascript,LaTeX,Bash
Advanced Knowledge:	Java,Scala,C#,C,Matlab,HTML,SQL