Nonparametric Inverse Reinforcement Learning and Approximate Optimal Control with Temporal Logic Tasks

by

Siddharthan Rajasekaran

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Master's

in

Robotics Engineering

in the

Graduate Division

of the

Worcester Polytechnic Institute

Committee in charge:

Professor Jie Fu, Chair
Professor Jane Li
Professor Carlo Pinciroli

Summer 2017

The dissertation of Siddharthan Rajasekaran, titled Nonparametric Inverse Reinforcement Learning and Approximate Optimal Control with Temporal Logic Tasks, is approved:

Chair _____ Date _____

_____ Date _____

_____ Date _____

Worcester Polytechnic Institute

Abstract

Nonparametric Inverse Reinforcement Learning and Approximate Optimal Control with Temporal Logic Tasks

by

Siddharthan Rajasekaran

Master's in Robotics Engineering

Worcester Polytechnic Institute

This thesis focuses on two key problems in reinforcement learning: How to design reward functions to obtain intended behaviors in autonomous systems using the learning-based control? Given complex mission specification, how to shape the reward function to achieve fast convergence and reduce sample complexity while learning the optimal policy?

To answer these questions, the first part of this thesis investigates inverse reinforcement learning (IRL) method with a purpose of learning a reward function from expert demonstrations. However, existing algorithms often assume that the expert demonstrations are generated by the same reward function. Such an assumption may be invalid as one may need to aggregate data from multiple experts to obtain a sufficient set of demonstrations.

In the first and the major part of the thesis, we develop a novel method, called Nonparametric Behavior Clustering IRL. This algorithm allows one to simultaneously cluster behaviors while learning their reward functions from demonstrations that are generated from more than one expert/behavior. Our approach is built upon the expectation-maximization formulation and non-parametric clustering in the IRL setting. We apply the algorithm to learn, from driving demonstrations, multiple driver behaviors (e.g., aggressive vs. evasive driving behaviors).

In the second task, we study whether reinforcement learning can be used to generate complex behaviors specified in formal logic Linear Temporal Logic (LTL). Such LTL tasks may specify temporally extended goals, safety, surveillance, and reactive behaviors in a dynamic environment. We introduce reward shaping under LTL constraints to improve the rate of convergence in learning the optimal and probably correct policies. Our approach exploits the relation between reward shaping and actor-critic methods for speeding up the convergence and, as a consequence, reducing training samples. We integrate compositional reasoning in formal methods with actor-critic reinforcement learning algorithms to initialize a heuristic value function for reward shaping. This initialization can direct the agent towards efficient planning subject to more complex behavior specifications in LTL. The investigation takes the initial step to integrate machine learning with formal methods and contributes to building highly autonomous and self-adaptive robots under complex missions.

Contents

List of Figures

Acknowledgments

All of the work described in this thesis was done in collaboration with my advisor, Jie Fu, who has continually pointed me in the right direction and provided inspiration to do the best work I could. I would also like to thank my co-advisor, Jane Li, for numerous discussions, guidance, and brainstorming sessions. I want to especially thank Jinwei Zhang for generating the datasets required to test my algorithms.

I want to thank Vishal Arya for the endless hours of discussions on the connection between animal psychology and reinforcement learning. I thank my sister and my parents for giving me the freedom to explore, and for creating the foundation to build my dreams.

Chapter 1

Introduction

1.1 Introduction

In this chapter, we give an informal introduction to Reinforcement Learning(RL) and Inverse Reinforcement Learning(IRL). Both RL and IRL, devise methods to allow autonomous agents to learn complex behaviors. In the following sections, we describe informally the problems each of these methods face and how we might address them.

1.2 Reinforcement Learning

Introduction to Reinforcement Learning

One of the fundamental problems in Optimal Control and Artificial Intelligence is solving the stochastic sequential decision-making problem. Driving a car is an example of controlling a stochastic system where we cannot predict the future exactly. This unpredictability is either due to unavoidable inaccuracies in the model or the model being inherently stochastic. Often probability is used to capture such inaccuracies to probabilistically predict the future and plan our present control actions to put the system in the desired path far in the future. Probabilistic modeling makes it possible to deal with real world problems but introduces a new set of challenges due to the exponential growth of possible paths the car can take due to the stochastic transitions.

To tackle this problem, we need a policy which is a mapping from any state we could land in, to the actions. This is a feedback mechanism where the state of the system is fed back as the input to the policy. This is essential because executing a plan without feedback and without considering the stochasticity in the model is likely to fail as the system may not evolve as intended. The exact state of the system can only be determined by the environment which makes it hard to precompute a sequence of control inputs without knowing what the environment would choose. One can then ask how it is possible to pre-compute a policy rather than just actions. A policy lets us evaluate every possibility in the future and hence

lets us have a plan irrespective of what the environment chooses. For each of its choices, we compute a counter strategy using our policy. If we enumerate every possible path the system might take, computing this policy becomes intractable and a challenging task.

Bellman made this problem tractable using Dynamic Programming (DP) which avoids having to deal with the exponentially growing number paths [3]. We defer from formally defining Markov Decision Processes (MDP) which is the framework used in DP to the next chapter and give a high-level overview of the formalism here. In an MDP, the system occupies a state and is allowed to take an action at each decision epochs. The agent of the system is an entity that has control over these actions. At each decision epoch, the agent chooses an action first and based on this action the environment samples a new state that the system would occupy and emits a reward. The objective of dynamic programming is to maximize the expected reward accrued over the paths followed by the system. Exact dynamic programming algorithms, which are explained in the following section, quickly become intractable. There exists a class of methods which perform Approximate Dynamic Programming (ADP) otherwise known as Reinforcement Learning (RL). Hence RL is nothing but a class of algorithms that find the optimal policy in an MDP by making crucial approximations. We will discuss in detail these approximations in the following chapters.

Learning in RL is harder than other types of learning like supervised or unsupervised. In supervised or unsupervised learning, we learn the underlying statistics of the generative model from sampled data. In RL, we have control over decision variables which dictate where we have to sample the next data. Also, as a result of this decision, we change the underlying state we are in where the statistics of the data which we are learning changes. Moreover, in RL, we have the added objective of maximizing the rewards by learning complex behaviors. Many a time the optimal behavior that accrues the maximum reward over time is complex and remains unspecified. Often It is required to guide the agent towards these behaviors to solve complex tasks. In this thesis, we propose a method to combine an alternative specification using Linear Temporal Logic(LTL) [30] with RL and thus guide the agent to solve the task at hand.

1.3 Inverse Reinforcement Learning

Reinforcement Learning (RL) refers to a class of learning tasks where the agent learns the associative mapping from its current state to actions using feedback signals from the environment. RL algorithms facilitate extraction of complex behaviors from simple reward specifications. Though the reward is a primary signal emitted by the environment, it is often specified by the user and not naturally perceivable by an autonomous agent. For instance, humans naturally associate pleasure to good (positive reward) and pain to bad (negative reward). On the other hand, all methods in RL need a specification of the reward for the agent to maximize (to know what is good and what is bad) since they are not inherent in the environment. Thus the reward signal can be formulated to engineer specific behavior. The unsaid problem in such a specification is that it is often unclear as to what

reward function would force the required behavior.

One can always specify a sparse reward signal by giving the agent a large positive reward only when it succeeds at the required task. Sparse rewards, however, make the learning harder since the agent now has to rely solely on random exploration to luckily happen upon this reward. In case of driving a car, a rather simple task for humans, one can give a high positive reward when the agent drives the car from point A to point B without colliding with other cars while following the traffic rules. Such a specification will theoretically make the agent converge at the required behavior. However, practically, it would be almost impossible for the agent to occur at this reward signal even once by exploring randomly. Also, the costs incurred during such a learning process is huge, and one might want to start with an acceptable behavior instead of a random one.

To address the sparse reward problem, there have been methods like reward shaping that motivate the agent towards promising actions that eventually lead to the actual reward [25]. However, this encompasses the original problem of rich reward specification since the desired behavior is often complex and giving an auxiliary reward for following this behavior requires some domain knowledge in the first place.

Inverse Reinforcement Learning (IRL) on the other hand automates the process of searching for the reward function. It takes as input demonstrations from the expert (which are valid examples of the required behavior) and learns a reward function which would result in a similar behavior when maximized. Learning the reward function from the demonstrations, however, is an ill-defined problem in that there can exist a large number of reward functions that could result in the same behavior. Maximum Entropy Inverse Reinforcement Learning (MaxEnt IRL) tackles this problem by associating the variance in the expert behavior to unintentional sub-optimality while not giving up on maximizing the reward accrued.

1.4 Key Contributions

Learning rewards from multiple behaviors in demonstrations

As mentioned in the previous section, MaxEnt IRL associates variance in the expert behavior by associating their variance to unintentional suboptimality. We posit that there can exist situations wherein all the variances in demonstration need not originate from the sub-optimality of a single behavior and could arise from more than one underlying behavior. We propose a novel solution to tackle this problem by combining Expectation Maximization and Inverse Reinforcement Learning algorithms.

Guiding RL based on Linear Temporal Logic (LTL) Specifications

In these methods, we sample data from the system by actually executing actions and receiving outcomes. Sampling based methods alleviate the modeling costs and the computation costs, that is, they do not need the complete model of the system and also tackle the curse of

dimensionality by making crucial approximations. These approximations handle the curse of dimensionality by trading off exactness for speed and sampling. The downside of this trade-off is that these methods now rely on exploring the entire state space to locate rewards and thus learn a behavior which maximizes the accrued reward. In fact, one can guarantee that the optimal behavior can be learned only if the agent has occurred at that behavior by chance [17]. In this thesis, we exploit the structure offered by a particular class of approximate methods called actor-critic to guide the agent towards a sparsely defined goal using Linear Temporal Logic (LTL) specifications.

Actor-Critic Method

In this section, we give a high-level overview of Actor-Critic methods and how we might exploit the structure in them to guide the agent towards a goal far into the future. In this thesis, we utilize a rather important connection between reward shaping and actor-critic methods. As the name suggests, the actor critic methods use actors to make decisions and critics to evaluate those decisions. One can view the critic as a means to change the convergence characteristics of the algorithm. Our method, to increase the rate of convergence, is highly inspired from reward shaping [23] where an auxiliary reward is given to encourage the agent to take promising actions. We propose to induce similar effects in actor-critic algorithms using value function initialization. We try to answer the question, "How can we formalize a good initialization of the critic to nudge the agent towards completion of tasks?".

Alternative High-Level Specifications Using Linear Temporal Logic (LTL)

Linear Temporal Logic (LTL) allows us to specify tasks where the agent has to satisfy some temporal constraints. They can be viewed as an alternative to reward functions for task specification. In fact, for temporal constraints, it is hard to specify a reward only as a function of the states.

For instance, imagine that our agent interacts with a simple grid world where it has four directions to choose from at every state. We might want to abstract away a high-level sequential task using reaching a set of regions in order. For instance, imagine we have a task of retrieving a key from a shelf and then using the key to open a safe elsewhere to get the document and then bring it to a target location. For this task, several sequential ordering is important, the agent cannot get the document without retrieving the key, and it probably is useless to go to the target location without the document and so on. Such sequential ordering of tasks can be better captured using Linear Temporal Logic (LTL) which uses a formal specification or a set of rules which evaluate to being true only when the task is completed successfully.

Reinforcement Learning cannot solely rely on "luckily" occurring at task completion based on random exploration. This becomes extremely unlikely when there is an LTL specification to be satisfied. In formal controller synthesis, they come up with strategies that directly maximize the probability of satisfying a specification while in RL we maximize the

reward. To close this gap, we propose to initialize value functions (critics) based on heuristics derived from LTL specifications.

1.5 Outline

The key contributions of the thesis are in Chapter.3 and Chapter.4. In Chapter.2, we introduce RL and IRL formally and introduce and interpret some of the seminal algorithms in these topics.

In the next chapter, we introduce the mathematical notation and framework for Markov Decision Processes (MDP), Reinforcement Learning and Inverse Reinforcement Learning. It introduces basic RL algorithms like value iteration and policy iteration used in the inner loop of the IRL problem. It also summarizes all the IRL algorithms in the literature into a general structure. We then give a broad overview of RL methods and explain how we might use actor-critic for shaping. In Chapter.3, we elaborate the theory behind our method and how we combine Expectation Maximization with MaxEnt IRL. It also includes the experimental results where the algorithm is tested on different types of demonstrations from simple grid-world and continuous car driving simulator tasks. In the last Chapter.4, we introduce a heuristic value initialization for actor critic methods and our results demonstrate its effectiveness on a simple grid world task.

Chapter 2

Preliminaries and Definitions

In this chapter, we formally introduce Markov Decision Processes (MDP), Reinforcement Learning (RL) and Inverse Reinforcement Learning (IRL). We will also explain the issues in these methods and a more formal introduction to our primary contributions.

2.1 Markov Decision Processes (MDP)

Definitions

Markov Decision Processes (MDP)

Markov Decision Processes (MDP) is a mathematical abstraction for making sequential decisions in the presence of uncertainty. An MDP is a tuple $(S, A, P(s'|s,a), E[r|s,a,s'], \gamma)$ where,

- S: The set of **states** that the system can occupy. This set can be discrete or continuous. In this thesis, we deal with discrete state MDPs
- A: The set of possible **actions** that the agent has control over at each decision epoch.
- $P(s'|s,a)$: The **state transition distribution** also referred to as **system dynamics**. This is the probability distribution over the states of the MDP we could land in given that we take an action a from state s. The environment samples the next state $s' \in S$ from this distribution which the system would occupy.
- $E[r|s,a,s']$: The real-valued **reward** function is the expected reward received by the agent given that it takes action a from state s and lands in state s'. Generally, we assume we have access directly to the expected reward itself and write it as $r(s,a,s')$
- γ: The **discount factor** which is in the range $(0, 1]$.

The Multi Stage Sequential Stochastic Process

Fig.2.1 shows the sequential stochastic process where we would like to maximize the accrued reward through possible actions. The system starts off at some initial state s_0, and the agent chooses an action $a_0 \in A$ which leads to the transition according to the state transition distribution. The system as a result lands in a state s_1 and receives a reward $r(s_0, a_0, s_1)$. In an MDP, we assume that the system is first order Markovian, i.e., the above-mentioned transition depends on only our current state and action. Note that we only have control over the actions. The exact transition is stochastic, and we have no control over the state in which we would land. In other words, we only choose which distribution we want the environment to sample from instead of choosing the exact resultant state through our actions. As a result of this transition, we reach the next decision epoch. This process repeats sequentially, and we get a series of rewards according to our decisions and transitions. The above process can also be viewed as a two player game where the agent acts first and tries to maximize the accrued reward, while the environment acts next by sampling the next state from the transition probability distribution (which is its fixed strategy).

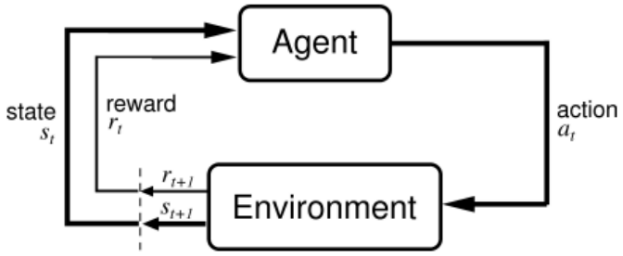


Figure 2.1: The agent-environment sequential interaction in reinforcement learning adapted from Sutton and Barto (1998) [33]

Returns

Consider the case of interacting with the system and observing a trajectory

$$\tau = \{(s_0, a_0, r_0), (s_1, a_2, r_2), \cdots\}$$

Since we are interested in the accrued rewards, there are many possible definitions of return such as the total return, mean return, fixed horizon return, discounted return, etc. In this thesis, we will exclusively use discounted return. In general, it is common to assume that the reward is independent of the state we land in and the discounted return G_t during the process at the decision epoch t is given by

$$G_t = r(s_{t+1}, a_{t+1}) + \gamma r(s_{t+2}, a_{t+2}) + \gamma^2 r(s_{t+3}, a_{t+3}) + \cdots$$

. Since the states visited along the trajectory are random variables by nature even if we follow a fixed set of actions, one can view the reward R_t received by the agent at time t as a random variable. The return at time t during one such interaction is given by,

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \tag{2.1}$$

One way to interpret discounted return is to think of it as giving lesser importance to the rewards obtained much farther away in the future. This is roughly equivalent to computing a total return for a receding horizon of $T = \frac{1}{(1-\gamma)}$ timesteps at every timestep. It is important to note that the trajectory that occurs during our interaction with the environment is a sample from the underlying probability distribution of the sequential process. The main idea of Approximate Dynamic Programming or Reinforcement Learning is to infer the statistics of the distribution using samples and thus solve an optimization problem defined in the following sections.

Policy

A policy is a plan or a strategy that the agent uses to take an action at any decision epochs that follow. It can be shown that a policy that depends only on the current state irrespective of the history can perform exactly as any history dependent policy [28]. Hence a policy $\pi : S \to \mathcal{P}(A)$ is nothing but a mapping from state to a probability distribution over the action space. It provides the agent a means to make a decision under any state the system might land in the future. Thus a policy is more than just an action. It is also common to only consider deterministic policies (the reasoning is given in the following section) which have the complete support of the probability distribution on a single action. We denote stochastic policy as a distribution over actions using $\pi(a|s)$ and deterministic policies as a mapping from states to a single action using $\pi(s)$.

Value Function

One can associate an expected return to a state s given that we start from that state and follow a policy π thereafter. This expected return, due to the Markovian nature of the MDP, depends only on the current state s and the future actions and not on the history. This is known as the value function $V : S \to \mathbb{R}$, a mapping from states to a real number, given by,

$$V^\pi(s) \triangleq \mathbb{E}_\pi\big\{G_t | s_t = s\big\} \tag{2.2}$$

where the expectation is taken with respect to the policy π and the system dynamics.

Occupancy of a Policy

Given a initial distribution ρ_0 from which the initial state s_0 is sampled, we can find the state occupancy, in other words, the probability that a given state is occupied by taking the actions according to our policy π.

$$D^\pi_{s,t+1} = \sum_{s',a'} P(s|s',a')\pi(a'|s')D^\pi_{s',t} \tag{2.3}$$

where $D^{\pi}_{s,t}$ is the state occupancy or probability that state s is occupied at time t under the policy π. We initialize $D_{s,0} = \rho_0$ and recursively compute the state occupancy using the above equation until any time t. We can also find the state-action occupancy using

$$D^{\pi}_{(s,a),t+1} = \pi(a|s) D^{\pi}_{s,t+1} \tag{2.4}$$

$$\text{otherwise} \quad D^{\pi}_{(s,a),t+1} = \sum_{s',a'} \pi(a|s) P(s|s',a') D^{\pi}_{(s',a'),t} \tag{2.5}$$

Note that the state occupancy is linear in the policy. That is, mixing two policies π_1 and π_2 using $\pi_m = \lambda\pi_1 + (1-\lambda)\pi_2$ (where $0 \leq \lambda \leq 1$) gives a state occupancy,

$$D^{\pi_m}_{(\cdot),t} = \lambda D^{\pi_1}_{(\cdot),t} + (1-\lambda) D^{\pi_2}_{(\cdot),t} \tag{2.6}$$

Steady state distribution of a Policy

When we propagate the system recursively using Eq.2.3 for a large number of steps, we reach the steady state distribution. That is, applying the update does not change the state occupancy i.e., $D^{\pi}_{s,t+1} = D^{\pi}_{s,t}$.

$$d^{\pi}_{s} = \lim_{n \to \infty} D^{\pi}_{s,t} \tag{2.7}$$

where d^{π}_{s} is the steady state distribution of the policy π.

Visitation frequency of a Policy

The state visitation frequency of a policy π is the total discounted number of times a particular state is occupied. Since we have only probabilistic occupancies, state-visitation is nothing but the cumulative probability mass that has visited a particular state.

$$D^{\pi}_{s} = \sum_{t} \gamma^t D^{\pi}_{s,t} \tag{2.8}$$

where D^{π}_{s} is the state visitation frequency

Similarly, the state-action visitation frequency is given by,

$$D^{\pi}_{(s,a)} = \sum_{t} \gamma^t D^{\pi}_{(s,a),t} \tag{2.9}$$

Feature Expectation

Often the value function, policy or the reward are parametrized in Reinforcement Learning using kernels, neural networks or a linear combination of a set of features[33]. A feature is any mapping f from a state s to a real number, i.e., $f : S \to \mathbb{R}$. A feature vector ϕ

is a vector of features[1]. Feature expectation $\mu(\pi)$ under a policy π refers to the vector of expected feature counts (the number of times a feature occurs).

$$\mu(\pi) = \sum_s D_s^\pi \phi(s) \tag{2.10}$$

From Eq.2.6, for the mixed policy $\pi_m = \lambda\pi_1 + (1-\lambda)\pi_2$ (where $0 \leq \lambda \leq 1$), we get a feature expectation

$$\mu(\pi_m) = \lambda\mu(\pi_1) + (1-\lambda)\mu(\pi_2) \tag{2.11}$$

The Optimization Problem

The return G_t at time t is a random variable due to the stochasticity either in the system dynamics or the policy. Hence, we would like to maximize the expected return with respect to the space of feasible policies.

$$\max_\pi \mathbb{E}_\pi\{G_t\} \tag{2.12}$$

where the expectation is taken both with respect to the stochasticity in policy (denoted in the subscript) and also the system dynamics. The above problem is an important subset of the Mathematical Programming called the Dynamic Programming. In this problem, there are implicit constraints in the above objective function which are dictated by the system dynamics and the nature of the multi-stage stochastic process. Solving this problem by search becomes intractable quickly given the exponentially complex nature of branching factor. For instance, leave alone the objective, the search space for a deterministic policy is $|A|^{|S|}$. Moreover, for continuous action/state spaces, the search space is infinite and we need to resort to other methods to solve the problem. One can bring in the concept of value function to write the objective in a more elegant form. For one, we notice that the return can be recursively written down as the following by popping out the first in the definition of the return $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$ as,

$$\mathbb{E}_\pi\{G_t\} = \mathbb{E}_\pi\{R_{t+1}\} + \mathbb{E}_\pi\{G_{t+1}\} \tag{2.13}$$

Given that we start from state s, the above equation can be rewritten using Eq.2.2 as

$$\mathbb{E}_\pi\{G_t|s_t = s\} = \mathbb{E}_\pi\{R_{t+1}|s_t = s\} + \mathbb{E}_\pi\{G_{t+1}|s_t = s\} \tag{2.14}$$

$$\therefore V^\pi(s) = \mathbb{E}_\pi\{R_{t+1}|s_t = s\} + \gamma \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)\mathbb{E}_\pi\big\{G_{t+1}|s_{t+1} = s'\big\} \tag{2.15}$$

$$= \sum_a \pi(a|s) r(s,a) + \gamma \sum_a \pi(a|s) \sum_{s'} P(s'|s,a) V^\pi(s') \tag{2.16}$$

$$V^\pi(s) = \sum_a \pi(a|s) \Big[r(s,a) + \gamma \sum_{s'} P(s'|s,a) V^\pi(s') \Big] \tag{2.17}$$

[1]This vector could be of any number of dimensions. Hence this is a vector valued function whose dimensions are given by the feature mappings f

The left side of Eq.2.17 is the promised cumulative pay off given that we start from state s and follow the policy π. The right side of the equation can be interpreted as getting an immediate reward of $\sum_a \pi(a|s)r(s,a)$ for our decision at time t then getting a future discounted reward that is promised from the state would land according to the transition distribution. Note that Eq.2.17 is a system of equations and there are as many equations as the number of states and all equations should hold simultaneously. The above can be written as a single linear equation in the vector of values for each state.

The solution to the optimization problem in Eq.2.12 is called the optimal policy π^*. Although the solution to this need not be unique, it can be shown that there must exist a deterministic policy that is optimal. This policy obeys the so called Bellman optimality equations.

$$V^*(s) = \max_a \left[r(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s') \right] \tag{2.18}$$

$$\text{and} \quad \pi^*(s) = \arg\max_a \left[r(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s') \right] \tag{2.19}$$

where the deterministic policy $\pi^* : S \to A$ is the only support of the distribution in action space. We will be using the notations $\pi(a|s)$ and $\pi(s)$ for stochastic and deterministic policy.

2.2 Dynamic Programming

Value Iteration

The set of equations in Eq.2.18 however are not linear. One way to solve these equations is to perform value iteration where we initialize the value to be some random function $V^{(0)}(s)$ and apply the Bellman update iteratively at all states. Practically, we can get the value within ϵ of the optimal value function with polynomial number of iterations (polynomial in $|\mathcal{S}|, |\mathcal{A}|$ and $\frac{1}{(1-\gamma)}$)

$$V^{(n+1)}(s) \coloneqq \max_a \left[r(s,a) + \gamma \sum_{s'} P(s'|s,a) V^{(n)}(s') \right] \tag{2.20}$$

where n is the iteration number.

Policy Iteration

Policy Evaluation

One of the main quantities used in arriving at the solution is the value of a policy. A value of a policy π is nothing but V^π as given by Eq.2.17. One can find this value by just replacing

the equation with an assignment operator similar to Eq.2.20 and iteratively repeating the update across all states.

$$V^{\pi}(s) \coloneqq \sum_{a} \pi(a|s)\Big[r(s,a) + \gamma \sum_{s'} P(s'|s,a)V^{\pi}(s')\Big] \tag{2.21}$$

Policy Update

Once we have the evaluated policy we update the current policy $\pi^{(n)}$ by being greedy with respect to the evaluated values.

$$\pi^{(n+1)}(s) = \max_{a} \sum_{s'} P(s'|s,a)V^{\pi^{(n)}(s')} \tag{2.22}$$

Policy iteration is said to converge when $\pi^{(n+1)} = \pi^{(n)}$ for all states.

2.3 Reinforcement Learning

In this section, we will give a brief overview of different methods used to perform reinforcement learning.

Monte-Carlo methods

Monte-Carlo methods directly sample trajectories by rolling out policies in the real system. After all, by the definition of value function (Eq.2.2), one can roll out the policy π several times until a terminal state is reached, then give an approximate estimate of value as,

$$V^{\pi}(s) \approx \hat{V}^{\pi} = \frac{1}{m}\sum_{i=1}^{m} G_t^i | s_t = s \tag{2.23}$$

where G_t^i is the return (defined in Eq.2.1) obtained at i^{th} roll out and m is the number of roll outs.

Let us have a current estimate of value function $\hat{V}$ and observe a new return starting at state s $G_s = [G_t|s_t = s]$. The incremental update to compute the new value V' can be written as,

$$\hat{V}'(s) = \underbrace{\hat{V}(s)}_{\text{old estimate}} + \frac{1}{m+1}\underbrace{(\overbrace{G_s}^{\text{target}} - \overbrace{\hat{V}(s)}^{\text{old estimate}})}_{\text{error}} \tag{2.24}$$

where G_s is the return obtained starting from state s. The equation above is the rolling average version of Eq.2.23. This is the general form of stochastic averaging equation where

we chase the stochastic target G_s by weighing each measurement by $\frac{1}{m+1}$. This means that when we have rolled out a large number of trajectories we only take very small proportion of error due to the new measurement (in fact, inversely proportional to the number of previous measurements). In the limit we would stop learning after a large number of roll outs. This can hinder us if the statistics of underlying quantity being estimated changes dynamically[2] Hence we replace the fraction and obtain,

$$\hat{V}'(s) = \hat{V}(s) + \alpha(G_s - \hat{V}(s)) \tag{2.25}$$

where α is now called the learning factor.

Temporal Difference (TD) Learning

The drawback in Monte-Carlo method is that we have to wait till an episode is complete to get an estimate G_s of the return. This wait forces us to perform only offline updates, i.e., perform the update in Eq.2.24 after an episode of the roll-out is completed. In temporal difference methods, instead of computing the target from the complete roll-out, we truncate the roll-out in n steps to get,

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{V}(s_{t+n+1}) \tag{2.26}$$

We use the above as target in the stochastic averaging equation to get the TD update. We write the update obtained by using $n = 0$ (one step return) as

$$\hat{V}'(s_t) = \hat{V}(s_t) + \alpha(\overbrace{R_t + \gamma \hat{V}(s_{t+1})}^{\text{target}} - \hat{V}(s_t)) \tag{2.27}$$

But with TD updates, we have a lower variance but higher bias target than compared to Monte Carlo methods. This allows us to learn effectively even with one step simulations in the real world. In fact, we will see in chapter.4 how we might exploit this bias to motivate the agent towards exploring in a promising direction.

Function Approximation in Reinforcement Learning

For large or continuous state spaces, it is not practical to maintain a value function or the policy as a look up table as given in the previous section. Hence we use function approximation for the value and policy functions. The methods in RL can then be classified into three different types based on what we approximate

- Value Based
- Policy Based
- Actor Critic (approximates both policy and value)

[2] In the following sections we will see that when we change the policy π based on our current estimate of the value, we need to weigh the latest measurements more.

Value Based Methods

The value function is then written as $V(s; w)$ where the weights can be the weights of linear combination of features, weights of neural network or any parameter of a function approximations. In this setup, we can perform what is called as fitted value iteration [4] which projects a value from regular value iteration step back to the space spanned by the function approximator. In case of least squared error formulation, the value update becomes,

$$w' = w + \alpha(R_t + \gamma \hat{V}(s_{t+1}) - V(s_t))\nabla_w V(s_t) \tag{2.28}$$

In case we have a small number of actions, it is enough to parametrize only the value. These are called critic only methods. If there are a large number of actions, one can also approximate the policy. In the following sections we will introduce the REINFORCE algorithm [35] that is central to policy based methods. We will also discuss how this leads to actor-critic methods.

Policy Optimization

In these methods, we parametrize the policy as $\pi(a|s; \theta)$, where the policy is stochastic and gives the probability of picking an action a based on the current state s, and the parameters θ. We can redefine the complete reinforcement algorithm as maximizing the accrued reward with respect to the parameters θ. We can define what is called the utility of the current policy as

$$U(\theta) = \sum_{\tau} P(\tau|\theta)R(\tau) \tag{2.29}$$

where τ is the trajectory obtained by following policy $\pi(\cdot; \theta)$. There can be different trajectories obtained by following the same policy because of stochasticity in the system, the policy, and the start state. The REINFORCE algorithm finds the gradient of the utility U with respect to the parameters θ as

$$\begin{aligned}
\nabla_\theta U(\theta) &= \sum_{\tau} \nabla_\theta P(\tau|\theta)R(\tau) && (2.30)\\
&= \sum_{\tau} P(\tau|\theta)\frac{\nabla_\theta P(\tau|\theta)}{P(\tau|\theta)}R(\tau) && (2.31)\\
&= \sum_{\tau} P(\tau|\theta)\nabla_\theta \ln P(\tau|\theta)R(\tau) && (2.32)\\
&= \mathbb{E}_\tau[\nabla_\theta \ln P(\tau|\theta)R(\tau)] && (2.33)
\end{aligned}$$

The above estimate can be approximated with m samples of the path rolled out in real system using the policy parameters θ.

$$\nabla_\theta U(\theta) \approx \hat{g} = \frac{1}{m} \sum_i \nabla_\theta \ln P(\tau^{(i)}|\theta) R(\tau^{(i)}) \tag{2.34}$$

$$\mathbb{E}[\hat{g}] = \nabla_\theta U(\theta) \tag{2.35}$$

where $\hat{g}$ is the approximation of gradient computed using samples.

Even with one roll-out ($m = 1$), we get an estimate of the gradient whose expectation has zero bias but has a very large variance. One way to get this derivative is to use finite differences which result in evolutionary strategy based methods [9, 32]. Here we evaluate the utility U at θ and $\theta + \Delta\theta$, where $\Delta\theta$ is some random noise and compute the gradient as

$$\nabla_\theta U(\theta) \approx \frac{\mathbb{E}[U(\theta + \Delta\theta)] - \mathbb{E}[U(\theta)]}{\Delta\theta} \tag{2.36}$$

However, in REINFORCE we can get the gradient (a direction along which we can improve the objective) only by behaving according to θ. This seems very unlikely since we cannot say anything about the neighboring policies without trying them out. The reason, however, is that, in REINFORCE, it is required that the policy is stochastic. Though it seems that we are rolling out a single parameter θ, we might, in fact, be sampling low probable actions whose probability would be increased if they turn out to be good (and hence learning). This is also reflected mathematically in Eq.2.31 which enforces $P(\tau|\theta) \neq 0$ (since in the denominator) which in turn forces $\pi(a|s) \neq 0$ for any state-action pair. The interpretation of this is that the policy has a support all over the action space and makes sure that we have some non-zero probability of trying out all actions.

The most important property of the approximate gradient of $U(\theta)$ is that it exists if the family of parameterized policy that we pick is differentiable and has no restriction on the reward function. Hence reward function can be discontinuous or unknown, or sample space of paths is a discrete set. This makes it ideal when we have LTL specifications which are known to be either satisfied or not. This nature of LTL makes the reward function notoriously sparse. However, for policy gradient, sparse rewards pose the problem of diminishing gradients in all directions. Another problem is the high variance of the estimates in Eq.2.34. We will see some methods to reduce the variance of this estimate. We propose to exploit the structure offered by variance reduction to get a non-zero signal using value function initialization.

Expanding the term $\nabla_\theta \ln P(\tau^{(i)}|\theta)$ in our gradient estimate gives,

$$\nabla_\theta \ln P(\tau^{(i)}|\theta) = \nabla_\theta \ln \Big[\prod_{t=0}^{T} P(s_{t+1}^{(i)}|s_t^{(i)}, a_t^{(i)}) \pi(a_t^{(i)}|s_t^{(i)};\theta) \Big] \tag{2.37}$$

$$= \nabla_\theta \Big[\sum_{t=0}^{T} \ln P(s_{t+1}^{(i)}|s_t^{(i)}, a_t^{(i)}) + \sum_{t=0}^{T} \ln \pi(a_t^{(i)}|s_t^{(i)};\theta) \Big] \tag{2.38}$$

$$= \nabla_\theta \sum_{t=0}^{T} \ln \pi(a_t^{(i)}|s_t^{(i)};\theta) \tag{2.39}$$

$$\implies \hat{g} = \frac{1}{m} \nabla_\theta \sum_i \sum_{t=0}^{T} \ln \pi(a_t^{(i)}|s_t^{(i)};\theta) R(\tau^{(i)}) \tag{2.40}$$

$$\text{and} \quad \nabla_\theta U(\theta) = \nabla_\theta \mathbb{E}_\tau \Big[\sum_{a,s \in \tau} \ln \pi(a|s;\theta) R(\tau) \Big] \tag{2.41}$$

We see that the policy gradient is independent of the dynamics of the system. Which means that we do not need to have the dynamics model of the system as long as we have access to roll the policy out.

Variance Reduction and Actor-Critic Methods

The approximate gradient $\hat{g} = \frac{1}{m} \sum_i \nabla_\theta \ln P(\tau^{(i)}|\theta) R(\tau^{(i)})$ takes a step towards positive rewards. Assume that our system has positive rewards only. Then, instead of decreasing the probability of smaller rewards, we basically increase the probabilities for all rewards. But the increase is proportional to the magnitude of this reward i.e., it is more for more promising actions and less otherwise. One way to actually decrease the probability of smaller rewards is to subtract a baseline b from the observed reward.

$$\hat{g} = \frac{1}{m} \sum_i \nabla_\theta \ln P(\tau^{(i)}|\theta)(R(\tau^{(i)}) - b) \tag{2.42}$$

$$b = \mathbb{E}[R(\tau)] \approx \frac{1}{m} \sum_{i=1}^{m} R(\tau^{(i)}) \tag{2.43}$$

It can be shown that adding this term b still has zero bias on our policy gradient estimate.

$$\mathbb{E}[\nabla_\theta \ln P(\tau|\theta) b] = \sum_\tau P(\tau|\theta) \nabla_\theta \ln P(\tau|\theta) b \tag{2.44}$$

$$= \nabla_\theta \Big(\sum_\tau P(\tau|\theta) b \Big) \tag{2.45}$$

$$= \nabla_\theta 1 \cdot b = 0 \tag{2.46}$$

This is true for any constant baseline b. In [15] they set $b = \frac{\sum_i (\nabla_\theta \ln P(\tau^{(i)}|\theta))^2 R(\tau^{(i)})}{\sum_i (\nabla_\theta \ln P(\tau^{(i)}|\theta))^2}$ for minimum variance which is a better choice than simply averaging the rewards in the roll outs. We can use the fact that future actions do not affect the past rewards to decrease the variance further.

$$\hat{g} = \frac{1}{m}\sum_{i=1}^{m}\Big(\sum_{t=0}^{T-1}\nabla_\theta \ln \pi(a_t^{(i)}|s_t^{(i)};\theta)\Big)\Big(\sum_{t=0}^{T-1} r(s_t^{(i)}, a_t^{(i)}) - b\Big) \tag{2.47}$$

$$= \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T-1}\left[\nabla_\theta \ln \pi(a_t^{(i)}|s_t^{(i)};\theta)\Big(\sum_{k=t}^{T-1} r(s_k^{(i)}, a_k^{(i)}) - b(s_t^{(i)})\Big)\right] \tag{2.48}$$

We are able to do this because we know ahead of time that (at any instance t) the following holds,

$$\lim_{m\to\infty}\sum_{i=1}^{m}\sum_{t=0}^{T-1}\left[\nabla_\theta \ln \pi(a_t^{(i)}|s_t^{(i)};\theta)\Big(\sum_{k=0}^{t-1} r(s_k^{(i)}, a_k^{(i)}) - b(s_t^{(i)})\Big)\right] = 0 \tag{2.49}$$

Since we know that the expectation is going to turn out to be zero, we can replace these terms with zero beforehand instead of allowing them to increase the variance of our estimate. Eq.2.48 now has a bias term dependent on the state. This can be replaced with the value function to yield the actor-critic method. We introduce discounting in the reward to get

$$\hat{g} = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T-1}\left[\nabla_\theta \ln \pi(a_t^{(i)}|s_t^{(i)};\theta)\Big(\sum_{k=t}^{T-1} \gamma^{t-k} r(s_k^{(i)}, a_k^{(i)}) - b(s_t^{(i)})\Big)\right] \tag{2.50}$$

$$= \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T-1}\left[\nabla_\theta \ln \pi(a_t^{(i)}|s_t^{(i)};\theta)\Big(r(s_t^{(i)}, a_t^{(i)}) + \gamma V(s_{t+1}^{(i)}) - V(s_t^{(i)})\Big)\right] \tag{2.51}$$

We will see how this is similar to reward shaping in Chapter.4. This similarity allows us to use value function initializations to direct the search in policy gradients,

2.4 Inverse Reinforcement Learning

It is indeed important for an autonomous agent to help humans complete a task. Often, agents programmed to complete a specific task lack the usability in varied applications. These variations may also arise in a particular task at hand. For instance, consider an agent which assists humans to build a mechanical system collaboratively. It is often necessary for the agent to understand the intent behind the motions of the human counterparts to help in coming up with strategies that aid humans without having to explicitly specify the exact commands of how to help/collaborate. Inverse Reinforcement Learning (IRL) provides a structure to this problem wherein the learned intent is more generalizable to new unseen situations.

Inverse Reinforcement Learning Algorithms

Stuart Russel first proposed the idea of learning the reward function given the demonstration in his book [31]. In the forward problem or reinforcement learning setting, one would be interested in finding the optimal policy π^* given the MDP i.e. the tuple (S, A, P, R, γ). In the inverse problem, we are given the MDP without the reward function, i.e., $MDP \setminus R$ but with an expert demonstration π^e. The posed problem is indeed inverse of the forward one - What is the reward function that could generate the policy π^e. This is equivalent to finding the intent behind the demonstrations rather than locally approximating the demonstrated trajectories itself as in behavioral cloning[5]. One of the main advantages of learning the reward functions is that the agent can solve the forward problem with the learned reward function to not only imitate the expert but can also predict the expert's motion. The latter turns out to be very important especially in case of collaboration between the expert and the agent. We have tried our best to summarize all IRL algorithms in the literature into the following steps,

1. Parametrize the reward function as $r(s, a, s'; w)$
2. Initialize w randomly
3. Solve the forward problem (reinforcement learning) problem for the current reward parameters to get the current optimal policy π^o
4. Compute the difference $\mathbf{e}$ between the expert's policy π^e and the current optimal policy π^o
5. If $\mathbf{e}$ is smaller than a threshold, stop. Or else update w in a direction that reduces $\mathbf{e}$ then go to step 3

The above is the basic structure of an IRL algorithm. One can get different IRL algorithms based on the way we parametrize the reward [21, 10] or the way we solve the forward control [2, 37, 10]. This is a structure that will help us in understanding MaxEnt IRL, the method this thesis mainly builds on.

Reinforcement Learning in Linear Reward Setting

In this section, we give a graphical intuition about what RL does when finding an optimal policy.

It is often common to use linear reward parametrization. For example, in the case of riding a bike, we can have a feature that measures the deviation of the bike from its frame of reference (i.e., upright with respect to the ground). Penalizing the agent by giving a negative weight for this feature would be sufficient since we let the RL algorithm come up

with the complex behavior that maximizes this reward. Assume we have a linear reward parametrization,

$$r(s) = w^T \phi(s) \tag{2.52}$$

where, r is the reward we get for reaching state s, w is the learnable weight vector, and ϕ is the vector of features. One of the dimensions of this feature vector ϕ in the bike example would be the angular deviation from the upright position.

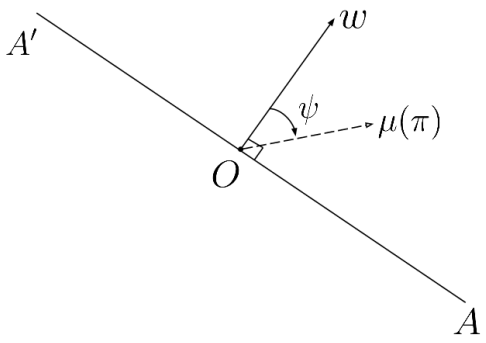


Figure 2.2: Cartoon showing the reward parameter vector w and the feature expectation $\mu(\pi)$ of a policy π.

Given a weight vector, the RL algorithm would maximize the total return from any state s_0.

$$\max_{\pi} G(s_0) \tag{2.53}$$

$$\text{where,} \quad G(s_0) = \mathbb{E}\Big[\sum_t \gamma^t r(s_t)|\pi\Big] \tag{2.54}$$

$$= \mathbb{E}\Big[\sum_t \gamma^t w^T \phi(s_t)|\pi\Big] \tag{2.55}$$

$$= w^T \mathbb{E}\Big[\sum_t \gamma^t \phi(s_t)|\pi\Big] = w^T \mu(\pi) \tag{2.56}$$

where, $\mu(\pi)$ is the discounted feature expectation of policy π (Eq.2.10). Intuitively, all Reinforcement Learning algorithms, to maximize $G(s_0)$, use policy to change the direction of the feature expectation $\mu(\pi)$. That is, if the feature expectation makes a smaller angle ψ with the weight vector (as shown in Fig.2.2), the agent receives bigger return (if each dimension in the weight is positive). In other words, if all elements in w are positive, RL algorithms answer the question "In what way can we reduce the angle between the vectors w and $\mu(\pi)$?".

Figure. 2.2 shows graphically the vectors w and $\mu(\pi)$ and the angle ψ between them that we try to minimize. One can claim that since RL minimizes ψ, $\mu(\pi)$ will be at least on the

same side of the hyperplane $A'OA$ as w [2]. If they were on the opposite side, then the RL algorithm would find a way to bring the dimension of feature expectation which lies on the other side of $A'OA$ to zero. This is a crucial insight used in Inverse Reinforcement Learning and is explained in the upcoming section.

SVM based IRL

In this section, we describe the method in [2]. We also discuss the shortcomings of the method when the demonstrations are suboptimal (which is often the case). This section has important notions in IRL and introduces feature expectation, policies mixing, etc. We also introduce Maximum Entropy IRL [37] which attempts to fix these shortcomings but induces an unforeseen problem in case of multiple behaviors in the demonstrations. We will also see how we might design an algorithm when there are multiple behaviors.

In [2], they use a linear reward parametrization and perform value iteration (in the inner loop in the IRL problem). One of the crucial insights exploited in SVM based IRL is that RL algorithms minimize the angle ψ between the weight vector w of the reward parametrization and the feature expectations $\mu(\pi)$. Another important concept is the ability to mix policies. Given two policies π_1 and π_2. We define a mixed policy π_m as,

$$\pi_m(s) = \lambda_1\pi_1(s) + \lambda_2\pi_2(s) \tag{2.57}$$

$$\text{s.t.} \quad \lambda_1, \lambda_2 > 0 \tag{2.58}$$

$$\lambda_1 + \lambda_2 = 1 \tag{2.59}$$

where λ_1 and λ_2 can be interpreted as the probability of choosing π_1 and π_2. Mixing of the policies allows us to behave in a such a way that we can get any feature expectation along the line joining $\mu(\pi_1)$ and $\mu(\pi_2)$ by varying the parameters λ_1 and λ_2. Afterall, the feature expectation is linear in policy and a mixed policy gives a mixed feature expectation with the same weights (Eq.2.11). This trick is used to achieve the feature expectation of the expert.

The SVM-IRL initializes a random weight vector $w^{(0)}$ and finds the corresponding optimal policy $\pi^{(0)}$ using value iteration as shown in Fig.2.3. Now, one can get the feature expectation $\mu(\pi^{(0)})$ by behaving according to π_0. However, we would like to have a feature expectation μ_E of the expert. Hence the method uses a weight vector $w^{(1)} = \frac{1}{\eta}(\mu_E - \mu(\pi^{(0)}))$. This forces the RL algorithm in the inner loop to find a policy $\pi^{(1)}$ that minimizes the angle between its feature expectation and $w^{(1)}$. As a result we get $\mu(\pi^{(1)})$ on the same side as μ_E (with respect to a hyper plane passing through $\pi^{(0)}$ and perpendicular to $w^{(1)}$). In an arbitrary n^{th} iteration, we can find the closest point in the convex hull formed by $\mu(\pi^{(0)}) \cdots \mu(\pi^{(n-1)})$ by solving a Linear Program. For instance, in Fig.2.3, the point $\bar{\mu}^{(2)}$ is the closest point (in

the convex hull formed by $\mu(\pi^{(0)})$, $\mu(\pi^{(1)})$, and $\mu(\pi^{(2)})$ to μ_E.

$$\min_{\lambda} ||\mu_E - \mu|| \tag{2.60}$$

$$s.t. \quad \mu = \sum_i \lambda_i \mu^{(i)} \tag{2.61}$$

$$\lambda_i \geq 0 \tag{2.62}$$

$$\sum_i \lambda_i = 0 \tag{2.63}$$

where $\mu^{(i)}$ is the feature expectation $\mu(\pi^{(i)})$ and λ_i are the probability of picking that particular policy. Let the feature expectation at this minimizing point be given by $\bar{\mu}^{(n)}$ at n^{th} iteration. The new weight $w^{(n)} = \mu_E - \bar{\mu}^{(n)}$ is the direction which the convex hull is lacking in representation. The fact that we end up closer to the expert's feature expectation μ_E at every iteration ensures that eventually μ_E lies inside the convex hull formed by $\mu(\pi^{(0)}) \cdots \mu(\pi^{(n)})$ for some n.

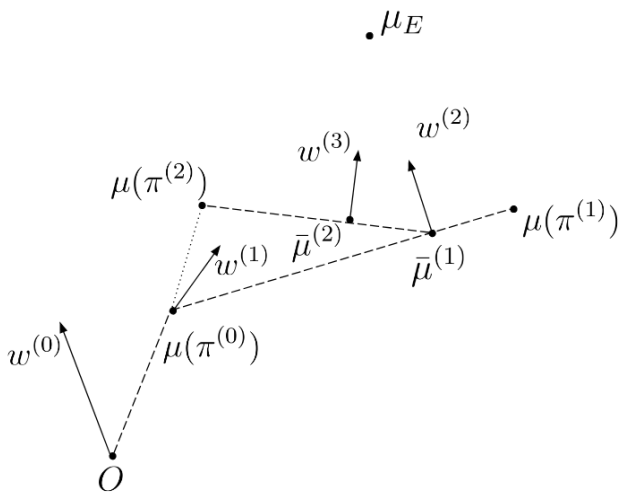


Figure 2.3: Cartoon showing iterations in SVM-IRL graphically. μ_E is the feature expectation of the expert. $w^{(0)}$ is the randomly initialized weights of the reward function. $\mu(\pi^{(0)})$ is the feature expectation after solving the RL problem with weights $w^{(0)}$. Note how each feature expectation tries minimize the angle between itself and the corresponding weight vector.

What Is Wrong With SVM IRL, and Its Current Fix?

The most important thing to note in SVM IRL is that we do not, in fact, learn the underlying reward function but rather the corner policies of the convex hull that allows us to match the feature expectation of the expert probabilistically. The main drawback of this is that

all the corners of the convex hull are the result of some optimal behavior. The expert, for instance, need not be optimal. This means that we solely rely on the mixed policy to match the expert. Although this mixture matches the feature expectation of the expert, it need not behave like the expert in reality [36]. The main reason behind this is that the expert does not switch between two optimal policies to execute a suboptimal policy. The expert is inherently suboptimal. This is shown in Fig.2.4 where it is seen that when expert chooses an action that results in Path 2, we probabilistically alternate between Path 1 and Path 3 to get a similar feature expectation but would never act like the expert. Also, there exist many policies that give the same return. It is not clear as to which policy should be preferred since we can only access the feature expectation of the expert through the structure of the IRL problem and not the exact trajectory itself [36].

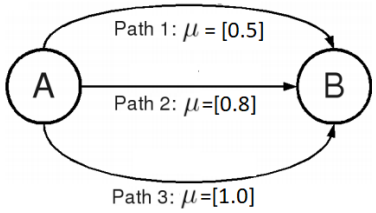


Figure 2.4: Cartoon (adapted from [36]) showing different paths available to go from A to B in an MDP. If the expert chooses Path 2 with feature expectation $\mu = 0.8$, then SVM-IRL chooses Path 1 with probability 0.4 and Path 2 with probability 0.6 to match the feature expectation.

This can be intuitively explained as the demonstrations being encoded into feature expectations. We are supposed to match the behavior of the expert only using the feature expectation and are not supposed to use the raw demonstrations (to avoid over fitting). In fact, the inverse mapping from a given feature expectation to behaviors is not essentially one to one. In [2], they use a forward control that picks one of the policies that results in the same feature expectation. This creates an unwarranted bias since the inverse mapping is an ill-defined problem and one must pick a uniform distribution over all these behaviors. Max-Ent IRL does exactly this and also picks suboptimal behaviors with probability proportional to their exponentiated return [36]. The main distinction between RL and IRL: In forward control, using such a randomized policy would mean that we are finding the most random policy instead of a deterministic one while not sacrificing the objective. In IRL, this means that the expert demonstrations are allowed to be similarly suboptimal while the algorithm would still find the underlying reward.

Chapter 3

Clustering Demonstrations using IRL

3.1 Introduction

Reinforcement Learning can be used to make an agent autonomously acquire complex behaviors using simple reward mechanisms. However, it can be hard to specify a reward function to obtain an intended behavior. Though theoretically, one can give a positive reward only when the task at hand is completed successfully (goal is reached), it proves to be hard for agents to explore randomly and occur at the goal by chance [1, 17, 23]. To tackle this, techniques like reward shaping are often used to direct the exploration to aid solving complex real-world problems [23]. Inverse Reinforcement Learning, on the other hand, provides a structured way of learning a reward function to obtain the required complex behavior using demonstrations from experts [2, 25, 36]. When Reinforcement Learning addresses the question "What possible behavior could maximize the cumulative reward?", Inverse Reinforcement Learning as the name suggests answers "What possible reward function could have made the expert choose these actions?". This is analogous to figuring out the intent behind the motions of the expert. Humans are proficient in finding the intent behind the motions of another individual. This is an essential mechanism to be automated since it can be used to predict and thus aid in collaborating with others. Once we have the reward functions learned from the demonstrations, we can use any conventional control or reinforcement learning algorithms [14, 19, 33] to solve the forward problem and thus achieve the desired behavior.

Among other IRL algorithms, MaxEnt-IRL [37] has received much attention as it addresses the ill defined problem of determining the best reward function that explains the expert's behavior. The problem is fundamentally ill-defined in that there can exist a range of reward functions that can explain the expert's behavior and also there can be a range of optimal behaviors given a single reward function. To do this, they posit that the expert prefers trajectories with a probability proportional to the exponentiated return along the trajectory. This associates the variance in expert's behavior to the sub-optimality involved in demonstrating. The inner loop of the MaxEnt IRL (forward problem) finds a policy that is most random while not giving up on maximizing the reward function being learned. This

is done so that we have a non-zero probability of behaving like the expert instead of choosing just a single optimal behavior which might have a high bias from the demonstrations. In our method, we use the insight that the variance in demonstrations may be due to more than one cluster of behavior each with a different reward function. These clusters may also hint at unmodeled reward feature which, if modeled, would better explain all the demonstrations. To solve this problem, we propose the non-parametric Behavior Clustering Inverse Reinforcement Learning Algorithm (abbreviated as non-parametric BCIRL).

One way to look at Inverse Reinforcement Learning algorithms is that they ultimately come up with generative models that best explain the variance in the demonstrations [10]. However, the learned reward function also helps in predicting the actions of agents in new unseen situations [37]. Therefore, it is important to reason about the different clusters of behaviors and learn the reward function of each cluster separately. In this thesis, we present a method that is interleaved with the existing IRL algorithm to simultaneously cluster while learning the reward function of each behavior separately while the underlying number of behaviors are not known.

3.2 Related Work and Overview

An entirely different approach to achieve desired behavior from demonstrations is by imitation learning [6, 7] using end-to-end learning. Here the actions taken during a demonstration are used as labels and states visited are used as corresponding inputs to train a nonlinear function approximator such as neural networks. These methods, however, do not generalize well to unseen situations and we need to use data aggregation based methods where a human is required in the loop to demonstrate for or label every unseen situation during training [8]. Inverse Reinforcement Learning algorithms learn the reward function using expert demonstrations to generalize well the expert's policy to states that have not been observed before. Another approach to learning from demonstrations is just to define the reward function so that we directly punish any deviations from the demonstrations [34]. These methods can work well in learning low-level motion primitives in dynamic systems, however, do not work in high-level decision-making problems since it is not straightforward to measure these deviations. Moreover, they require hand tuning the reward function to imitate expert's behavior in similar situations which can often be domain specific.

While MaxEnt IRL associates the variance in the expert behavior completely to their suboptimality [37], BCIRL associates large, consistent variances to multiple behaviors (when such a grouping/clustering can better explain the demonstrations). By learning from all demonstrations using MaxEnt IRL, one is likely to learn a behavior that best approximates the whole set of demonstrations in an average sense. That is, if we have examples of aggressive, evasive and tailgating behaviors in a single set of demonstrations without discrimination, the agent will learn a reward function that explains some proportion of the three behaviors. However, the resulting reward function need not result in any of the three behaviors but in an average behavior which is completely unintended.

Our method addresses this problem by creating new clusters each with a separate reward function. We propose to perform soft clustering on the demonstrated dataset. Instead of assuming that all demonstrations come from a single reward function, we rather have a probability indicating the likelihood that a particular demonstration comes from a particular cluster. We use the EM algorithm to associate each demonstration to a cluster while simultaneosly learning their reward functions in the inner loop.

3.3 Outline

The remaining sections are organized as follows: In section.3.4 we give a brief explanation of the MaxEnt IRL algorithm [37] which is be the algorithm that runs in our inner loop. Section.3.5 defines the problem statement. In Section.3.6 we first explain how the Expectation Maximization is used to perform an E-step using the probability that a demonstration belongs to a cluster and maximisation uses the same maximization objective as in MaxEnt IRL algorithm. We explain how we perform BCIRL in case we know the exact number of clusters (parametric BCIRL) and then extend this to cases where we do not have this information using non-parametric clustering. In the results section.3.7, we compare vanilla MaxEnt IRL to BCIRL and also parametric BCIRL to non-parametric BCIRL.

3.4 Preliminary and Notations

In this section, we give a high-level overview of the problem to show how MaxEntIRL is used in the inner loop of our method. We also briefly explain the soft value iteration (used in MaxEnt IRL) and the MaxEnt IRL algorithms.

Notations

An MDP is a tuple (S, A, P, R, γ), where S is the set of states, A is the set actions, P the transition probability, R is the rewards, and γ is the discount factor (defined in more detail in section.2.1). We denote the total number of underlying behaviors by m. The set of all reward parameters is denoted by $\Theta \equiv \{\theta_1, \theta_2, \cdots, \theta_m\}$ where each θ_j is the reward parameter vector of the j^{th} behavior. The dataset $\mathcal{D}$ we obtain from the user consists of n demonstrations. Each demonstration is a trajectory τ^i which is a sequence of state-action pairs visited along the trajectory i.e., $\tau^i = \{(s_0^i, a_0^i), (s_1^i, a_1^i) \cdots, (s_T^i, a_T^i)\}$ where the superscript i denotes the index of the demonstration and the subscript denotes the time index. Unless stated, τ without superscript is any possible trajectory in a given MDP and τ^i is the i^{th} demonstration in our dataset. We will use the terms behavior and cluster interchangeably[1]. We use c_j to denote the j^{th} cluster being learned. Through out the rest of the chapter, BCIRL without any prefix refers to the non-parametric version of behavior clustering IRL.

[1]In fact, we are clustering behaviors and each cluster corresponds to a single behavior

Overview of our problem

We are given a dataset $\mathcal{D}$ consisting of demonstrations generated by one or more agents with several different behaviors. We assume that there exists m consistent behaviors[2] and also that there exists at least more than one behavior (i.e., $m > 1$). In case of just one behavior, the algorithm turns into vanilla MaxEnt IRL. The BCIRL would indeed cluster all demonstrations into a single cluster in this case. We assume that we do not have explicit knowledge of which demonstration is attributed to which behavior. Our aim is to find the reward function of each behavior given these demonstrations using MaxEnt IRL. It is important to note that each behavior must be unique; otherwise, we would have no way to distinguish demonstrations from one another. In the end, we would find the reward function for each cluster of behavior that can best explain all the demonstrations in that cluster. We propose to use Expectation-Maximization using the insight that each behavior maximises their reward function independent of others[3]. Hence associating a demonstration to a behavior (expectation step) and then maximizing the likelihood of the demonstrations under the reward (maximisation step) recursively will converge.

Soft-Value Iteration

In section.2.2, we saw how methods like value iteration allowed us to perform dynamic programming efficiently. The original value iteration (Eq.2.20) finds one of the optimal policy which maximizes the return from every state. On the other hand, the soft version of value iteration finds the maximum entropy policy, i.e., the policy which is most random but still does not give up on maximizing the accrued reward (return). In this section, we define the soft-value iteration for an MDP. A complete derivation of this method can be found in [36]. In an MDP (S, A, P, R, γ), we perform the following updates iteratively to converge to the soft values.

$$V^{(n+1)}(s) = \ln\Big(\sum_{a} e^{r(s,a)+\gamma\sum_{s'} P(s'|s,a)V^{(n)}(s')}\Big) \tag{3.1}$$

where $V^{(n)}(s)$ is the value of state s in the n^{th} iteration. After convergence[4], the value at the last iteration n^* is the soft-value and the soft Q-values can be computed using the system model (transition probabilities).

$$V^{soft} \leftarrow V^{(n^*)} \tag{3.2}$$

$$\text{and} \qquad Q^{soft}(s,a) = r(s,a) + \gamma\sum_{s'} P(s'|s,a)V^{soft}(s') \tag{3.3}$$

[2]By consistent we mean that each behavior has little to no variance among all the demonstrations associated with it

[3]In general in Inverse Reinforcement Learning (IRL) it is common to assume that all the demonstration are independent of each other [2, 37]

[4]If the maximum difference between the value functions at any state in consecutive iterations is less than a threshold ϵ, it is considered converged

Maximum Entropy Inverse Reinforcement Learning

In this section, we first describe the Maximum Entropy IRL Algorithm [36]. According to the maximum causal entropy principle [36], the agent chooses policies with equal returns equally likely and policies with higher returns exponentially more likely [37]. It can be shown that this maximum entropy policy which stochastically mixes all these policies is

$$\pi(a|s) = \exp(Q^{soft}(s,a) - V^{soft}(s,a)) \tag{3.4}$$

where, Q^{soft} and V^{soft} are obtained from soft-value iteration [36]. Under maximum entropy policy, probability of a trajectory τ^i is given by,

$$P(\tau^i|\theta) = \frac{\exp(r_\theta(\tau^i))}{\sum_\tau \exp(r_\theta(\tau))} \tag{3.5}$$

where τ is any trajectory, $r_\theta(\tau)$ is the discounted return of the trajectory τ given the parameters θ of the reward function, and τ^i is the trajectory whose probability we want to find. We abbreviate the partition function $\sum_\tau \exp(r_\theta(\tau))$ as Z. The log likelihood of the demonstrations and the derivatives are given by

$$\mathcal{L}(\theta) = \sum_{\tau^i \in \mathcal{D}} \log P(\tau^i|\theta) \tag{3.6}$$

$$= \frac{1}{n} \sum_{\tau^i \in \mathcal{D}} r_\theta(\tau^i) - \log Z \tag{3.7}$$

$$\frac{\partial \mathcal{L}(\theta)}{\partial \theta} = \frac{1}{n} \sum_{\tau^i \in \mathcal{D}} \frac{\partial r_\theta}{\partial \theta}(\tau^i) - \frac{1}{Z} \sum_\tau \exp(r_\theta(\tau)) \frac{\partial r_\theta}{\partial \theta}(\tau) \tag{3.8}$$

$$= \frac{1}{n} \sum_{\tau^i \in \mathcal{D}} \frac{\partial r_\theta}{\partial \theta}(\tau^i) - \sum_\tau P(\tau|\theta) \frac{\partial r_\theta}{\partial \theta}(\tau) \tag{3.9}$$

$$= \frac{1}{n} \sum_{\tau^i \in \mathcal{D}} \frac{\partial r_\theta}{\partial \theta}(\tau^i) - \sum_{s,a} D(s,a|\theta) \frac{\partial r_\theta}{\partial \theta}(s,a) \tag{3.10}$$

where $D(s,a|\theta)$ is the state-action visitation frequency in the MDP given the maximum entropy policy from Eq.3.4 (computed according to the current reward parameters θ). With a slight abuse of notation $r_\theta(s,a)$ is the reward (and not the discounted return) emitted by the environment when leaving s with action a when using the parameters θ. One can use several parametrizations for the reward function depending on how non-linear the reward functions are in the state-action space. Levine *et al.* use Gaussian Process (GP) parametrization for the reward function to combine probabilistic reasoning about stochastic expert behavior (using IRL) with the ability to learn the reward as a nonlinear function of features (using GP). Finn *et al.* [11] use a deep neural network to completely eliminate the effort of the user

to design features and to learn reward directly as a function of the states and the parameters. In this thesis, we use a linear reward parametrization i.e., $r_\theta(s,a) = \theta^T \phi(s,a)$ where $\phi(s,a)$ is a set of vector features obtained when taking action a from state s. Often linear reward parametrization are enough to procure complex behaviours from the agent. For instance, in [1] they only use features such as the angle of the bicycle from the upright position (i.e., the frame of reference) and associate a negative reward with the angle accordingly. We will use $\phi(\tau)$ to represent the discounted feature expectation along the trajectory τ^i i.e., $\phi(\tau^i) = \sum_t \gamma^t \phi(s_t, a_t)$. It is to be noted that though we demonstrate our results on linear reward parametrization and driving task, the algorithms shall be easily extended to other parametrizations and domains.

$$\frac{\partial \mathcal{L}(\theta)}{\partial \theta} = \frac{1}{n} \sum_{\tau^i \in \mathcal{D}} \phi(\tau^i) - \sum_{s,a} D(s,a|\theta)\phi(s,a) = \bar{\phi}_{\mathcal{D}} - \bar{\phi}_\pi \tag{3.11}$$

where $\bar{\phi}_{\mathcal{D}}$ is the feature expectation of the expert demonstrations and $\bar{\phi}_\pi$ is the feature expectation of the steady state distribution under the maximum entropy policy π Eq.3.4.

The MaxEnt-IRL essentially initializes the parameters θ randomly and uses gradient descent to maximize the objective $\mathcal{L}(\theta)$. Intuitively, maximizing the likelihood in Eq.3.7 means that we are searching for the parameters θ of the reward function which maximizes the probability of seeing the demonstrations $\mathcal{D}$ while minimizing (to maintain a margin) the probability of observing any other trajectory[5]. In linear reward setting, we can interpret the gradient in Eq.3.11 as the difference between the feature expectation of expert and that of the current maximum entropy policy. Updating θ along this minimizes this difference since the objective function is just the inner product of the parameters and the difference i.e., $\mathcal{L}(\theta) = \theta^T(\bar{\phi}_{\mathcal{D}} - \bar{\phi}_\pi)$. Clearly this objective function is increased when the length of projection of θ along the difference $\bar{\phi}_{\mathcal{D}} - \bar{\phi}_\pi$ is increased. In SVM IRL (section.2.4), we use a new weight equal to this difference and try to reach the expert's feature expectation by mixing each policy. In MaxEnt IRL, we add the difference to the current weight to nudge it towards the missing components which we get closer to the expert's feature expectations.

3.5 Problem Statement

We are given an $MDP \setminus R$, that is, the tuple (S, A, P, γ) and the set of demonstrations $\mathcal{D}$ from the expert(s). Each demonstration in the dataset $\mathcal{D}$ is a trajectory τ which is a sequence of state-action pairs visited during that demonstration $\tau^i = \{(s_0^i, a_0^i), (s_1^i, a_1^i) \cdots, (s_T^i, a_T^i)\}$. We assume that each of these demonstration can arise probabilistically from several different behaviors c_j. In other words we perform soft clustering, that is, we use the probability $P(c_j|\tau^i)$ to assign a cluster c_j to the demonstration τ^i. We are also given the features $\phi(s,a)$ in which the reward is linear, i.e., $r_j(s,a) = \theta_j^T \phi(s,a)$ is the structure of the reward function

[5]Note that $\log Z = \log \sum_\tau \exp(r_\theta(\tau))$ sums over all the possible trajectories in the second term of Eq.3.7

of cluster c_j. We need to find the underlying distribution over clusters $P(c_j|\tau^i)$ for every trajectory τ^i and also the reward function parameters θ_j assigned with each of these clusters. We also have to learn the number m of clusters that could have possibly generated this data.

3.6 Non-parametric Behavior Clustering Inverse Reinforcement Learning

We first formulate the Expectation Maximization (EM) algorithm for a known number of behaviors (parametric). We then use non-parametric Bayesian clustering (which does not require the number of clusters as an input parameter) to extend it to more general case [12, 22] (Section.A.3).

The Generative Model

Here we discuss the assumed generative model that the environment or the underlying generative process uses to generate these demonstrations with various possible behaviors. We assume that there exists a probability distribution Ψ over the possible clusters (behaviors) $\{c_j : 1 \leq j \leq m\}$ in the demonstrations $\mathcal{D}$ to allow soft clustering. Each cluster c_j has a unique parameter θ_j for its reward function. First, a cluster c_j according to the multinomial distribution Ψ over the class variables are sampled, then the maximum entropy policy π_j according to the reward parameters θ_j is computed. The environment then samples a trajectory from the system according to this policy π_j and the system dynamics by rolling out. All these roll-outs are then added to the data set $\mathcal{D}$.

The Missing Data

In our problem, the data is generated from the distribution $P(\tau|\Theta, \Psi)$. Here, the observable data is the set of all demonstrations $\mathcal{D}$ and the missing data is the class assignment c_j (comes from j^{th} behavior) for each demonstration. Our objective is to maximize $P(\mathcal{D}|\Theta, \Psi)$. However this is a harder problem than maximizing $P(\mathcal{D}, c_j|\Theta, \Psi) = P(\mathcal{D}|c_j, \Theta, \Psi)P(c_j|\Theta, \Psi)$. This motivates us to formulate the problem of Multiple Behavior IRL problem in our setting as EM problem. We assume that the prior on class assignment is given by a distribution Ψ ($P(c_j|\Theta) = P(c_j) = \Psi(c_j)$), which is also one of the parameters to be learned[6].

EM formulation

The objective of our clustering IRL algorithm is

[6] A regular multinomial/catagorical parametrization - one parameter $\Psi(c_j)$ for each variable c_j

$$\max_{\Theta,\Psi} \ln P(\mathcal{D}|\Theta,\Psi) \tag{3.12}$$

Assuming that each demonstration was generated independent of the other, we can rewrite the objective as

$$\Theta_{opt}, \Psi_{opt} = \arg\max_{\Theta,\Psi} \sum_{i=1}^{n} \ln(P(\tau^i|\Theta,\Psi)) \tag{3.13}$$

$$= \arg\max_{\Theta,\Psi} \sum_{i=1}^{n}\sum_{j=1}^{m} P(c_j|\tau^i,\Theta,\Psi)\ln(P(c_j,\tau^i|\Theta,\Psi)) \tag{3.14}$$

We can get Eq.3.14 from Eq.3.13 using the usual steps for the expectation maximization as in Eq.A.8. The term inside arg max in Eq.3.14 is the expectation step. Consider

$$P(c_j|\tau^i,\Theta,\Psi) \propto P(\mathcal{D}^i|c_j,\Theta,\Psi)P(c_j|\Theta,\Psi) \tag{3.15}$$

Given no knowledge about the number of demonstrations that come from each behavior, we can initialize Ψ to the maximum entropy solution (uniform distribution), i.e., $\Psi^{(0)}(c_j) = P(c_j|\Theta,\Psi) = \frac{1}{m} \;\; \forall j$ for the 0^{th} iteration. The complete expectation step can be written as ,

$$\sum_i \sum_j \frac{P(\tau^i|c_j,\Theta,\Psi)P(c_j|\Theta,\Psi)}{\sum_k P(\tau^i|c_k,\Theta,\Psi)P(c_k|\Theta,\Psi)} \ln(P(c_j,\tau^i|\Theta,\Psi)) \tag{3.16}$$

Substituting the expectation probability term from the previous, t^{th}, iteration (as in Eq.A.8) we get the likelihood to be,

$$\mathcal{L}_t(\Theta,\Psi) = \sum_i \sum_j \frac{P(\tau^i|c_j,\Theta^{(t)},\Psi^{(t)})\Psi^{(t)}(c_j)}{\sum_k P(\tau^i|c_k,\Theta^{(t)},\Psi^{(t)})\Psi^{(t)}(c_k)} \ln\left(P(\tau^i|c_j,\Theta,\Psi)\Psi(c_j)\right) \tag{3.17}$$

Let us define

$$\beta_{ij}^{(t)} = \frac{P(\tau^i|c_j,\Theta^{(t)},\Psi^{(t)})\Psi^{(t)}(c_j)}{\sum_k P(\tau^i|c_k,\Theta^{(t)},\Psi^{(t)})\Psi^{(t)}(c_k)} \tag{3.18}$$

where $\beta_{ij}^{(t)}$ is the probability that demonstration τ^i comes from the cluster c_j, i.e., $P(c_j|\tau^i)$. this can be determined from the t^{th} iteration of the EM algorithm. Hence we know the values $\beta_{ij}^{(t)} \;\; \forall i,j$. Plugging these in, the maximization step reduces to

$$\Theta^{(t+1)}, \Psi^{(t+1)} = \max_{\Theta,\Psi} \sum_i \sum_j \beta_{ij}^{(t)} \ln \big(P(\tau^i|c_j, \Theta, \Psi)\Psi(c_j)\big) \tag{3.19}$$

$$\text{s.t.} \sum_j \Psi(c_j) = 1 \tag{3.20}$$

$$\tag{3.21}$$

The objective can be written as,

$$\Theta^{(t+1)}, \Psi^{(t+1)} = \arg\max_{\Theta,\Psi} \sum_i \sum_j \beta_{ij}^{(t)} \Big[\ln \big(P(\tau^i|c_j, \Theta, \Psi)\big) + \ln \big(\Psi(c_j)\big) \Big] \tag{3.22}$$

In the above equation, we can reduce the first term inside the summation as

$$P(\tau^i|c_j, \Theta, \Psi) = P(\tau^i|c_j, \Theta) = P(\tau^i|c_j, \theta_j) = P(\tau^i|\theta_j) \tag{3.23}$$

because c_j acts just like a switch which means that j^{th} reward function is selected.

Also, the first term in the summation is independent of Ψ and the second term is independent of θ. Hence we can maximize the terms separately as two independent problems. Note that the constraint $\sum_j \Psi(c_j) = 1$ applies to the second term only. We now have two independent problems in M-step: 1) Maximizing the likelihood of demonstrations and 2) Maximizing the likelihood of the prior.

Problem 1:

This is the first term in the double summation of Eq.3.22.

$$\max_{\Theta,\Psi} \sum_i \sum_j \beta_{ij}^{(t)} \Big[\ln \big(P(\tau^i|c_j, \Theta, \Psi)\big) \Big] \tag{3.24}$$

We can change the order of summation here and take the max operation into the first summation to get the maximizing argument as

$$\Theta^{(t+1)} = \sum_j \arg\max_{\theta_j} \sum_i \beta_{ij}^{(t)} \ln \big(P(\tau^i|\theta_j)\big) \tag{3.25}$$

$$\tag{3.26}$$

where each term in the outer summation is $\mathcal{L}_m(\theta_j) = \arg\max_{\theta_j} \sum_i \beta_{ij}^{(t)} \ln \big(P(\tau^i|\theta_j)\big)$ which is the same as the log likelihood of demonstrations under a single reward function [37] except that each likelihood is now weighed by β_{ij}. The gradient of $\mathcal{L}_m(\theta_j)$ is now given by,

$$\nabla_{\theta_j} \mathcal{L}_m(\theta_j) = \sum_i \beta_{ij}^{(t)} \tilde{f}^i - \sum_i \beta_{ij}^{(t)} D^{\pi_j}_{(s_i,a_i)} f_{s_i,a_i} \tag{3.27}$$

$$= \bar{\phi}_{\mathcal{D}_j} - \bar{\phi}_{\pi_j} \tag{3.28}$$

where $\tilde{f}^i$ is the feature expectation (section.2.1) of the i^{th} demonstration, $D_{(s_i,a_i)}$ is the state-action visitation frequency using the maximum entropy policy π_j, $\bar{\phi}_{\mathcal{D}_j}$ is the corrected[7] feature expectation of the demonstrations, and $\bar{\phi}_{\pi_j}$ is the corrected[7] feature expectation of the maximum entropy policy π_j. We use Algorithm.1 to compute $\bar{\phi}_{\mathcal{D}_j}$ and Algorithm.2 to compute $\bar{\phi}_{\pi_j}$. Weighing gradients from each term makes sense because, if the probability of a demonstration i coming from behavior j is very low, i.e., $\beta_{ij}^{(t)}$ is close to zero, then the contribution of the gradient from i^{th} demonstration to j^{th} reward parameters is zero. That is, we do not want to increase the likelihood of i^{th} demonstration by changing the parameters of j^{th} reward as the demonstration could not have come from j^{th} cluster anyway.

According to our assumption that demonstrations are independent, each term under the outer summation in Eq.3.25 is an independent maximization problem. Hence our problem will still find the global optimum in deterministic cases just as in MaxEnt IRL. In case of non-deterministic cases, an equivalently optimal solution as MaxEnt IRL solution [37] in stochastic cases. In other words, we do not loose optimality because of this summation.

Problem 2:

$$\Psi^{(t+1)} = \arg\max_{\Psi} \sum_i \sum_j \beta_{ij}^{(t)} \ln\big(\Psi(c_j)\big) \tag{3.29}$$

$$\text{s.t.} \quad \sum_k \Psi(c_k) = 1 \tag{3.30}$$

The Lagrangian of Problem 2 is given by,

$$\Gamma = \sum_i \sum_j \Big[\beta_{ij}^{(t)} \ln\big(\Psi(c_j)\big)\Big] + \lambda\Big[\sum_k \Psi(c_k) - 1\Big] \tag{3.31}$$

$$\text{s.t.} \quad \lambda \neq 0 \tag{3.32}$$

Finding the derivative of the Lagrangian and setting it to zero, we get,

$$\frac{\partial \Gamma}{\partial \Psi(c_j)} = \frac{1}{\Psi(c_j)} \sum_i \beta_{ij} + \lambda = 0 \tag{3.33}$$

$$(or) \qquad -\lambda \Psi(c_j) = \sum_i \beta_{ij} \tag{3.34}$$

$$\Psi(c_j) \propto \sum_i \beta_{ij} \tag{3.35}$$

$$\Psi(c_j) = \frac{\sum_i \beta_{ij}}{\sum_i \sum_j \beta_{ij}} \tag{3.36}$$

[7]By corrected we mean weighting the demonstrations according to the likelihood that they were generated from this cluster

Eq.3.36 comes from the constraint that the probabilities must sum to 1 (Eq.3.30). But from Eq.3.18, we see that

$$\sum_j \beta_{ij} = 1$$
$$\implies \Psi(c_j) = \frac{\sum_i \beta_{ij}}{\sum_i 1} = \frac{\sum_i \beta_{ij}}{n}$$

where n is the number of demonstrations. Reintroducing the time index

$$\Psi(c_j)^{(t+1)} = \frac{\sum_i \beta_{ij}^{(t)}}{n}$$

Algorithm 1 `visitationFromDemonstration`($\mathcal{D}$, β_j)

Input:

$\mathcal{D} \rightarrow$ The dataset of demonstrations

$\beta_j \rightarrow$ Vector of probabilities $P(c_j|\tau^i)$ for all i

Output:

$D_{(s,a)} \rightarrow$ The state action visitation frequency of demonstrations from c_j

$\gamma \leftarrow$ The discount factor of the MDP
$\epsilon \leftarrow 0.001$
$n \leftarrow$ Number of demonstrations in $\mathcal{D}$
for $i = 1 : n$ **do**
    for $(s, a), t \in \tau^i$ **do**
        $D^i_{(s,a),t} \leftarrow \gamma^t \frac{\beta_{ij}}{n}$ ▷ visitation frequency of t^{th} state-action pair from i^{th} trajectory
$D_{(s,a)} = \sum_{i,t} D^i_{(s,a),t}$
return $D_{(s,a)}$

Algorithm 2 `visitationFromPolicy`(π_j, ss, β_j, γ)

Input:

$\pi_j \rightarrow$ The policy from cluster j for which visitation frequency is to be found

$ss \rightarrow$ The start states of the demonstrations

$\beta_j \rightarrow$ Vector of probabilities $P(c_j|\tau^i)$ for all i

Output:

$D^{\pi_j}_{(s,a)} \rightarrow$ The state-action visitation frequency of the policy π

$\gamma \leftarrow$ The discount factor of the MDP

$P \leftarrow$ The transition probability of the MDP

$\epsilon \leftarrow 0.001$

$n \leftarrow$ Number of demonstrations in $\mathcal{D}$

$D^{\pi_j}_{s,0} \leftarrow 0 \quad \forall s \in \mathcal{S}$ ▷ state visitation at $t = 0$ [8]

for $i = 1 : n$ **do**

  $s_0 \leftarrow ss^i$ ▷ the start state in the i^{th} demonstration

  $D^{\pi_j}_{s_0,0} \leftarrow D^{\pi_j}_{s_0,0} + \frac{\beta_{ij}}{n}$ ▷ computes the initial distribution over states

$D^{\pi_j}_{(s,a),0} \leftarrow \pi(a|s) D^{\pi_j}_{s,0}$ ▷ state-action visitation at $t = 0$

$N \leftarrow \frac{1}{(1-\gamma)}$ ▷ some number of order $\frac{1}{(1-\gamma)}$

for $t = 0 : N$ **do**

  $D^{\pi_j}_{(s,a),t+1} = \sum_{s',a'} \pi(a|s) P(s|s',a') D^{\pi}_{(s',a'),t}$ ▷ state-action visitation at t

$D^{\pi_j}_{(s,a)} = \sum_t \gamma^t D^{\pi_j}_{(s,a),t}$

return $D^{\pi_j}_{(s,a)}$ ▷ state-action visitation frequency

Parametric Behavior clustering IRL Algorithm

In this section we explain the Algorithms.1, 2, and 3. We discuss the important steps in these algorithms and refer to the equations in the derivation of the parametric behavior clustering IRL method.

Algorithm.3 takes as inputs the demonstrations $\mathcal{D}$, the number m of clusters, the learning rate l and any standard maximum entropy based RL-algorithm [11, 18, 21, 37] RL. In this thesis we use the standard MaxEnt forward problem and RL would be a function that takes in the reward parameters and solves for the maximum entropy policy as in Eq.3.4. At every iteration, we perform one step of gradient ascent to maximize the likelihood in the maximization step and take expectation with respect to the distribution $P(c_j|\tau^i) = \beta_{ij}$. To compute the gradient using Eq.3.11, we need the feature expectations $\bar{\phi}_{\mathcal{D}}$ and $\bar{\phi}_{\pi}$ which are obtained using the state-action visitation frequencies.

$$\bar{\phi} = \sum_{s,a} D(s,a)\phi(s,a) \tag{3.37}$$

[8] The first subscript is the state and the second one after comma is the time index.

For the demonstrations, the state-action visitation frequency is nothing but the discounted counts of state-action pairs. Since in our case, we weigh each demonstration differently for a given cluster, we get

$$D_j(s,a) = \frac{1}{n}\sum_i \sum_t \gamma^t \beta_{ij} \mathbb{1}(s_{i,t} = s \wedge a_{i,t} = a) \tag{3.38}$$

where D_j is the state-action visitation frequency of the demonstrations for j^{th} cluster, n is the number of demonstrations and γ is the discount factor of the MDP. Computation of D_j is shown in Algorithm.1. The input β_j to the Algorithm.1 is the vector of probabilities $P(c_j|\tau^i)$ for all i which is required to evaluate Eq.3.38.

To compute the state-action visitation from policies however, it does matter where we start each demonstrations. One approach would be to approximate the start state distribution with some parametrized probability distribution and then sample n start states to compute the state-action visitation frequency by following the policy. Here, we just use a uniform distribution over the start states in the demonstration. Also, each demonstration is weighed differently similar to state-action visitation computation for demonstrations Eq.3.38.

$$D_{s,0} = \frac{1}{n}\sum_i \beta_{ij} \mathbb{1}(s = s_i \wedge t = 0) \tag{3.39}$$

where $D_{s,0}$ is the initial state distribution (at $t = 0$) and s_i is a state in i^{th} demonstration. Computation of D_{π_j} is shown in Algorithm.2.

In Algorithm.3, we address clustering demonstrations given the number of clusters (like the hyper parameter K in K-means). In the following section.3.6, we address the problem of learning the number of clusters using Chinese Restaurant Process (CRP) [12]. As we will see in the results section, this also allows the EM algorithm to escape local minima by creating new clusters in the early phases of convergence.

Algorithm 3 parametricBCIRL($\mathcal{D}$, m, l, RL)

Input:

$\mathcal{D} \rightarrow$ The dataset of demonstrations
m The number of clusters
l The learning factor
RL Some Reinforcement Learning algorithm such as soft value iteration

Output:

Finds which cluster each demonstration belongs to i.e., $P(c_j)|\tau^i$ and the reward function of each cluster

1: $n \leftarrow$ Number of demonstrations in $\mathcal{D}$
2: $P \leftarrow$ The transition probability of the MDP
3: $\theta_j \leftarrow$ Random sample for all j
4: $\Psi(c_j) \leftarrow \frac{1}{m}$ for all j
5: $\epsilon \leftarrow 0.001$
6: $ss \leftarrow$ `startStates(`$\mathcal{D}$`)`
7: **while** $True$ **do**
8: **for** $j = 1 : m$ **do**
9: $\pi_j \leftarrow RL(\theta_j)$ ▷ Some RL algorithm such as soft value iteration
10: **for** $i = 1 : n$ **do**
11: $\beta_{ij} \leftarrow \Psi(c_j) \cdot \prod_t \pi_j(a_t|s_t) P(s_{t+1}|s_t, a_t)$ where $s_t, a_t \in \tau^i$
12: $\beta_{ij} \leftarrow \frac{\beta_{ij}}{\sum_j \beta_{ij}} \ \forall i$ ▷ normalize β_{ij}
13: **for** $j = 1 : m$ **do**
14: $D_j \leftarrow$ `visitationFromDemonstration(`$\mathcal{D}$`,` β_j`)`
15: $D_{\pi_j} \leftarrow$ `visitationFromPolicy(`π_j`,`ss`,` β_j`)`
16: $\bar{\phi}_{\mathcal{D}} \leftarrow \sum_{s,a} D_j(s,a)\phi(s,a)$ ▷ weighted visitation frequency of demonstrations
17: $\bar{\phi}_{\pi_j} \leftarrow \sum_{s,a} D_{\pi_j}(s,a)\phi(s,a)$ ▷ weighted visitation frequency of the policy π_j
18: $\theta_j \leftarrow \theta_j - l \cdot (\bar{\phi}_{\mathcal{D}} - \bar{\phi}_{\pi_j})$
19: $\Psi(c_j) \leftarrow \frac{\sum_i \beta_i}{n}$
20: **if** $||(\bar{\phi}_{\mathcal{D}} - \bar{\phi}_{\pi_j})||_\infty < \epsilon$ **then** ▷ if the gradient $\rightarrow 0$
21: **return** θ

Non-parametric Behavior Clustering IRL Algorithm

Algorithm.4 shows the complete method to learn the number of clusters also. Here we use Chinese Restaurant Process (CRP) A.2 A.3 as the prior instead of fixed prior Ψ as in Algorithm.3. Lines 9 and 15 through 18 in Algorithm.4 show the crucial differences from using a fixed number of clusters. According to CRP, the prior probability of a demonstration being associated with an existing cluster is proportional to the probability density of demonstrations in that cluster, and that of a new cluster is proportional to the hyperparameter α

(Eq.A.11). The posterior $P(c_j|\tau^i)$ is given by

$$\beta_{ij} \triangleq P(c_j|\tau^i) = \frac{1}{\eta} P(\tau^i|c_j)P(c_j) \tag{3.40}$$

This posterior β_{ij} always has a non-zero probability of starting a new cluster at every iteration (due to the nature of CRP). Therefore at every iteration, it becomes a distribution over the existing number clusters plus one (new cluster). At one point, we would need to solve the reinforcement learning problem as many times as the number of demonstrations[9]. This can be quite costly and unnecessary as the density of the probability over most of these clusters are close to zero and solving the forward problem produces almost no update to the parameters. That is if a demonstration i could not have come from cluster j (β_{ij} is close to zero) then we would not see any update in the parameters θ_j.

To solve this problem, we perform *weighted re-sampling/bootstrapping* from the posterior to get rid of residual probability masses in new clusters. This is done in the `bootStrap` function in line 16 of Algorithm.4. This is analogous to resampling in particle filter to make the best use of the limited representation power of the particles. This also reduces high time complexity by redistributing the particles at high probability masses and eliminating particles at low probable regions. This re-sampling does not affect the point of convergence since in an expectation over several iterations we would be sampling from the original distribution (β'_i in the algorithm). Lines 15 and 16 in Algorithm.4 can be interpreted as removing i^{th} demonstration from its existing cluster and reassigning it. This procedure is similar to clustering using Dirichlet Process Mixture Models (DPMM)[22].

Intuitively, if a random reward function gives a comparable explanation to the demonstrations (i.e., likelihood $P(D_j|\theta_{random})$ is comparable), then we likely create a new cluster. Sampling from the posterior means that we not only look at the number of demonstrations in a cluster but also the likelihood that the demonstration at hand was generated from that cluster. That is, as evident from the relation given in Eq.3.40, if $P(c_j)$ is low and the demonstration at hand is actually from c_j, then $P(\tau^i|c_j)$ would be high to make sure c_j has a high probability of getting sampled from the posterior $P(c_j|\tau^i)$.

The non-parametric BCIRL with CRP, as shown in the results section, not only solves the problem of learning the number of clusters based on the likelihood of the demonstrations but also allows EM to escape local minima created in the beginning. As we will see in the results section, the parametric BCIRL often gets stuck in local minima. Non-parametric BCIRL detects the presence of a new cluster using non-parametric clustering (CRP) and thus saves the efforts of deciding a problem specific threshold variance beyond which we should create a new cluster. Though the parameter α in Algorithm.4 can be seen as a means to set this threshold, it is a robust parameter i.e. the algorithm is stable for a range of values of α.

[9]Actually the number of clusters can go till infinity but it does not make sense to have more clusters than there are demonstrations

Algorithm 4 non-parametricBCIRL($\mathcal{D}$, α, RL)

Input:

$\mathcal{D} \rightarrow$ The dataset of demonstrations

$\alpha \rightarrow$ The parameter of CRP

RL Some Reinforcement Learning algorithm such as soft value iteration

Output:

Finds the number of clusters, which cluster each demonstration belongs to i.e., $P(c_j)|\tau^i$, and the reward function of each cluster

$n \leftarrow$ Number of demonstrations in $\mathcal{D}$
`nc` $\leftarrow$ `[]` ▷ Number of demonstrations in each cluster
$P \leftarrow$ The transition probability of the MDP
$\beta_i \leftarrow 0 \ \ \forall i = 1 : n$
$ss \leftarrow$ `startStates(`$\mathcal{D}$`)`
while $True$ **do**
    `m` $\leftarrow$ `len(nc)`
    $\theta_{(m+1)} \leftarrow$ Random Sample from prior
    `p` $\leftarrow$ `normalize(merge(nc, [`α`]))`
    for $i = 1 : n$ **do**
        for $j = 1 : m$ **do**
            $\pi_j \leftarrow RL(\theta_j)$
            $\beta'_{ij} \leftarrow$ `p[j]` $\cdot \prod_t \pi_j(a_t|s_t)P(s_{t+1}|s_t, a_t)$ where $s_t, a_t \in \tau^i$
        $\beta'_{ij} \leftarrow \frac{\beta'_{ij}}{\sum_j \beta'_{ij}}$ ▷ normalize β_{ij}
        `nc = nc -` β_i ▷ element wise subtraction
        $\beta_i \leftarrow$ `bootStrap(`β'_i`)` ▷ weighted re-sampling from the distribution β'_i
        `nc = nc +` β_i ▷ reseat i^{th} demonstration according to new distribution[10]
        `nc.sparsify()` ▷ remove zero entries
    for $j = 1 : m$ **do**
        $D_j \leftarrow$ `visitationFromDemonstration(`$\mathcal{D}$`,` β_j`)`
        $D_{\pi_j} \leftarrow$ `visitationFromPolicy(`π_j`,`ss`,` β_j`)`
        $\bar{\phi}_{\mathcal{D}} \leftarrow \sum_{s,a} D_j(s, a)\phi(s, a)$
        $\bar{\phi}_{\pi_j} \leftarrow \sum_{s,a} D_{\pi_j}(s, a)\phi(s, a)$
        $\theta_j \leftarrow \theta_j - \alpha(\bar{\phi}_{\mathcal{D}} - \bar{\phi}_{\pi_j})$
        $\Psi(c_j) \leftarrow \frac{\sum_i \beta_i}{n}$
        if $||(\bar{\phi}_{\mathcal{D}} - \bar{\phi}_{\pi_j})||_\infty < threshold$ **then** ▷ if the gradient $\rightarrow 0$
            return θ

[10]This is element-wise addition. Might increase the size of the list when size of β_i is greater than that of nc

3.7 Experiments and Results

Experimental Setup

Grid World

In our first set of experiments, we used 15×15 grid world. The reward function is not known, and we use stochastic policies to obtain trajectories and compare the learned trajectory to that of expert's policy. The agent has four actions to chose from to move in the four directions ($A \equiv [Right, Up, Left, Down]$) but with 10% probability the action fails and results in a random move. We test the algorithm using demonstrations generated by four different state independent policies π_1, π_2, π_3 and π_4. We choose $\pi_1 = [0.5, 0.5, 0, 0]$ where each element in the list is the probability of taking the corresponding actions in A. Similarly, $\pi_2 = [0, 0.5, 0.5, 0]$, $\pi_3 = [0, 0, 0.5, 0.5]$ and $\pi_4 = [0.5, 0, 0, 0.5]$. With these policies, we have the element of both similarities and dissimilarities. That is, we see that π_1 and π_2 both move up 50% of the time (similarity) while they move in opposite direction the rest of the time (dissimilarity). We would like the algorithm to see the consistent movements in opposite directions to separate them into different clusters though there exists a consistent similarity rest of the time.

We generate 25 trajectories each from the four policies each of length 30. We add all these trajectories, in order, to the dataset with 100 demonstrations[11]. The value of γ is set to 0.9 so that the expected horizon is in the order of the size of the grid. If we were to use MaxEnt IRL to learn a single policy $\hat{\pi}$ that approximate all the 100 demonstrations we might see that the learned policy is $\hat{\pi} = [0.25, 0.25, 0.25, 0.25]$. However, under this policy every demonstration is very less likely since the π_1 would take a series of right and up actions which become exponentially less probable with the length of the demonstration. In fact, it is 0.5^t times less probable to generate a trajectory using $\hat{\pi}$ that was actually generated by π_1. Another main intuition is that even if a single expert generated these trajectories, it is not equivalent to replace the four policies with $\hat{\pi}$. The expert might be taking into consideration some of the features that have not been modeled yet in the reward features [20].

Results

The synthetic dataset $\mathcal{D}$ is generated in order from π_1 through π_4. Hence after convergence, we expect to see $P(c_j|\tau^i)$ to be either 1 or 0 given the difference in policies. Figure. 3.1 shows the plot of $P(c_j|\tau^i)$ vs the index of the demonstration after learning using the behavior clustering IRL algorithm.

[11] We use the order only for visualizing and interpreting the results and by itself is not required by the algorithm for clustering

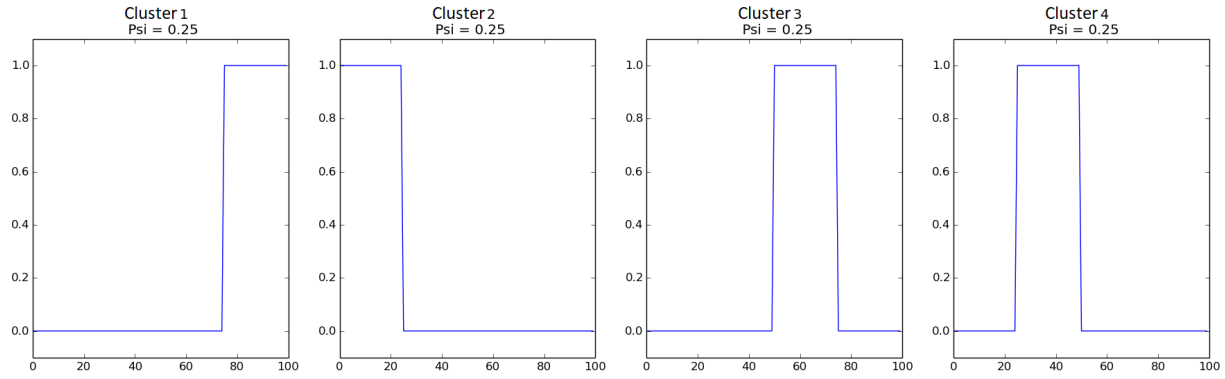


Figure 3.1: The probability of i^{th} demonstration belonging to j^{th} class that is, $P(c_j|\tau^i)$ or β_{ij} vs i. From left to right we have $j = 1$ through $j = 4$

Note that in Fig.3.1, the first cluster ($j = 1$) corresponds to the last 25 of the demonstrations. The learned policy for this cluster[12] was $[0.5, 0, 0, 0.5]$ with a variance of $[0.05, 0, 0, 0.05]$ for the corresponding dimension of the policy. This is, in fact, the policy π_4 with errors due to the limiting number of demonstrations. Similarly, for all the other clusters we obtained the corresponding policies. When vanilla MaxEnt IRL was used, we similarly obtained the average learned policy as $[0.25, 0.25, 0.25, 0.25]$ similar to what we would expect using $\hat{\pi}$. Though this is a simple example, this shows basically how non-parametric BCIRL automatically clusters behaviors based on their variances. Another important comparison is that while using parametric BCIRL, it often gets stuck in local minima. In 100 runs of parametric BCIRL, only about 25% of them converged to the correct solution. Others either 1) cluster all the demonstrations into one cluster and learn $\hat{\pi}$ since for all the 100 demonstrations $\hat{\pi}$ is a local minima or 2) Forms lower than 4 clusters with 25, 25 and 50 demonstrations each or 25 and 75 demonstrations each. Although the non-parametric BCIRL occasionally had such distributions across demonstrations before convergence, ultimately, they always converged to the optimal number of clusters (four here).

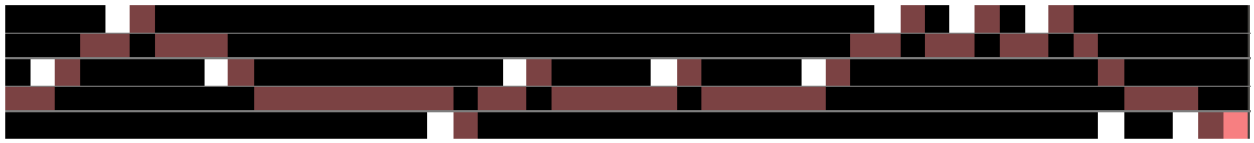
Grid-World Driving Task

In this experiment, we implemented a car driving task in grid world to learn different lane changing behavior. All the other cars are randomly initialized moving at a constant speed while the agent is moving much faster than the others. Fig.3.2 shows the the two different driving styles (aggressive and evasive). An aggressive driver tries to cut in front of other cars after overtaking them while an evasive driver tries to be as far away from the other cars as possible. To generate demonstration we hard code a probability over the possible actions (right, left, hard-right, hard-left, forward and break) depending on the relative pose of the

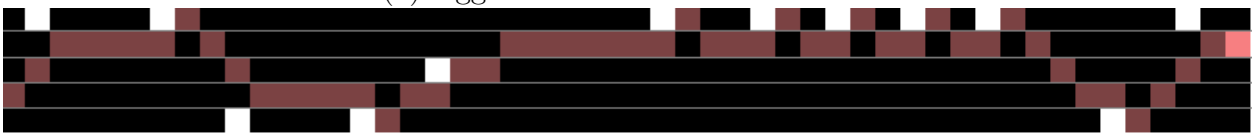
[12]The results were averaged over 100 runs of the algorithm on a single instance of generated data set

agent and other cars. The aggressive driver, when far away from the next car in the track takes the action forward with 70% probability

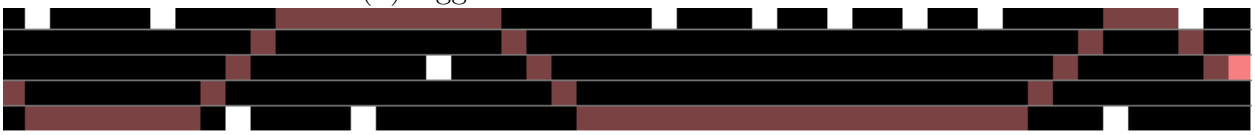
The algorithm is very robust to the parameter α of the CRP[13]. Though α is an indirect way to specify a threshold in the variance to create a new cluster it does not require a lot of tuning. A low value of α has a tendency to get stuck in the local minima in the initial phases of convergence while a very large value detects the correct number of clusters early on but takes a long time to converge[14]. For all the tasks, we use $\alpha = 3$ (though values ranging till 20 gave similar results).



(a) Aggressive driver demonstration



(b) Aggressive driver learned behavior



(c) Evasive driver demonstration

(d) Evasive driver learned behavior

Figure 3.2: The 5-lane road task where non-parametric behavior clustering IRL learns different lane switching behaviors. The left of each of the figure is the start position, and the right is the end position. Dark pink (maroon) shows the path taken by the driving behavior and the lighter pink (in far right) is the final position of the agent. Note how the learned behavior generalizes well to unseen situations.

For this task, we use a state action indicator feature for all action when the agent is far away from the next car, in the vicinity and while overtaking the next car. Also, we use the feature x, which encodes the tendency of the agent to go towards the right as in Fig.3.2. Fig 3.2(b) shows the learned aggressive behavior from one run of the non-parametric BCIRL

[13]This is not to be confused with the learning factor. This is the parameter proportional to which a new cluster is created in CRP

[14]This is because at every iteration we have a large probability of creating a new cluster

algorithm. Note how the positions of other cars are completely different from those in the demonstration.

Simulator Driving Task

We collected demonstrations from the Gazebo simulator in ROS [29]. The set-up is shown in Fig.3.3. We used potential fields [13], a heuristic control, to generate our dataset to show that the method extends well even for sub-optimal expert demonstrations with different behaviors. We did this especially to validate the assumption that the expert is optimal in some sense (since we use RL which is an optimal control in the inner loop of IRL).

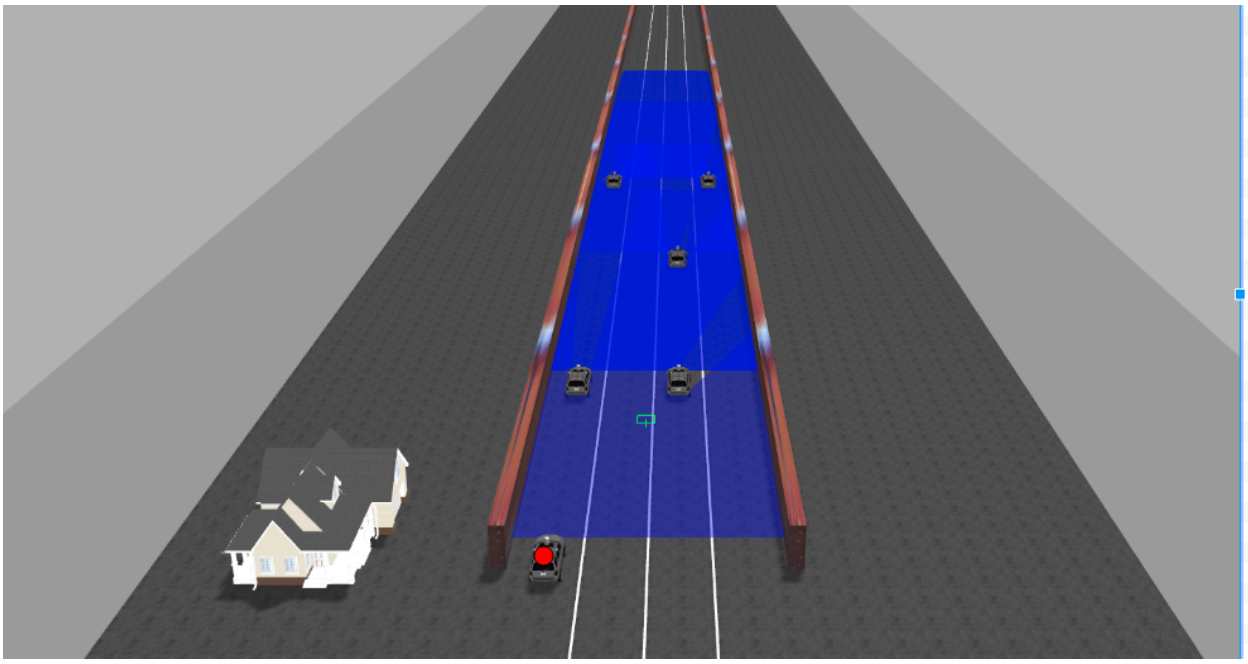
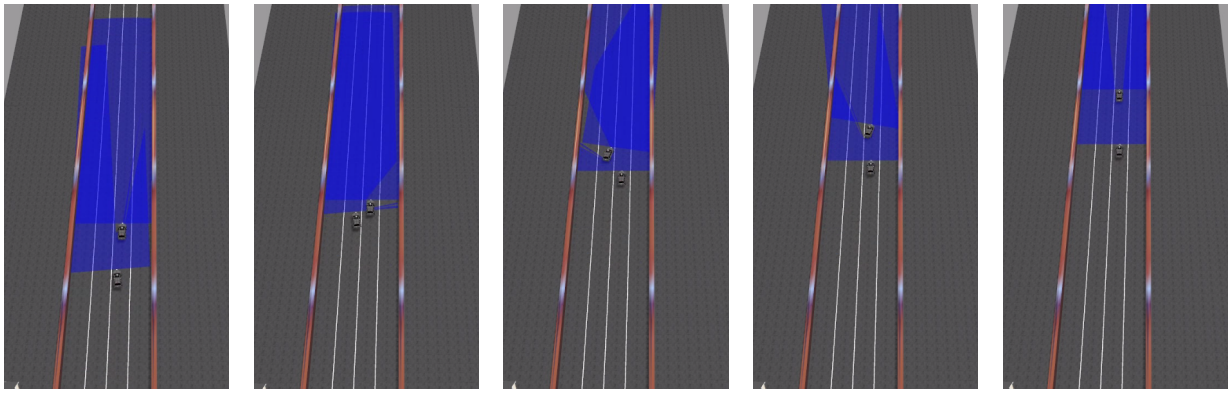


Figure 3.3: The overview of the simulation in Gazebo. The agent is marked in red. Other cars have a random constant speed while the agent has control over its speed and steering.

We have a linear attractive potential at the far end at the end of the course. Each car (other than the agent) is associated with a repulsive Gaussian potential. At both right and left end of the road, we have a log barrier to keep the car from straying away. We generate two behaviors with the potential field similar to the grid world driving task.

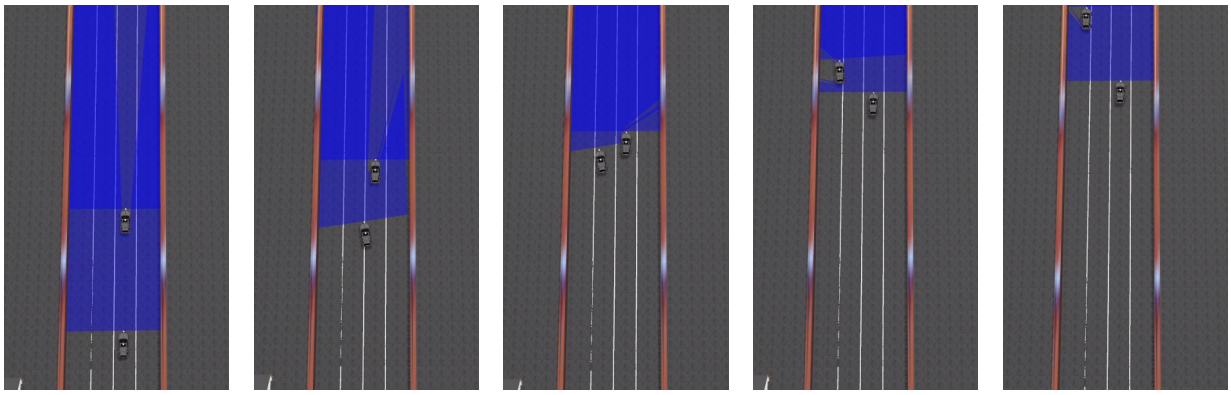
1. **Aggressive:** We use a weaker repulsive potential associated with other cars to allow close interactions. Also, each of them is associated with an attractive potential in front of them to allow the aggressive agent to cut in front of other cars. Aggressive behavior is shown in Fig.3.4.

2. **Evasive:** We use one stronger repulsive potential associated with other cars to make our agent be evasive. Evasive behavior is shown in Fig.3.5.



(a) (b) (c) (d) (e)

Figure 3.4: Shows the aggressive driving behavior in Gazebo simulator in the order a, b, c, d, and e generated using potential fields.



(a) (b) (c) (d) (e)

Figure 3.5: Shows the evasive driving behavior in Gazebo simulator in the order a, b, c, d, and e generated using potential fields.

We generated 25 demonstrations for each behavior. The trajectories obtained from the simulator was discretized into cells of size $l \times b$ where l is the length of the car and b is the width of each lane. In our simulator, it was $4.6m \times 4m$. With this discretization, we were able to cluster both behaviors in the demonstrations and learn their individual reward functions. The non-parametric behavior clustering algorithm was able to get a demonstration likelihood of 0.28 while the vanilla MaxEnt IRL could only get a likelihood of 0.09. However,

there were five clusters each containing 21, 19, 5, 4, and 1 demonstrations. The first two clusters correspond to evasive and aggressive behaviors while the remaining clusters did not belong to either of these behaviors. In fact, this was because these demonstrations were not consistent with the rest in any way due to the use heuristic potential field method used to generate them. This is a very important application of BCIRL – though there are bad demonstrations, the algorithm learns from the good [15] ones automatically.

3.8 Conclusion and Future Work

We presented the non-parametric behavior clustering IRL algorithm which uses non-parametric Bayesian clustering to simultaneously cluster demonstrations generated by different behaviors and their reward functions. The algorithm clusters different behaviors based on their consistently large variances. The algorithm is able to cluster different behaviors successfully even though several parts of their trajectories are similar. The approach works well as long as there exist consistent differences. One of the main challenges in extending this method to an arbitrary domain is to choose a feature space which captures the difference in each behavior. This feature space should have different feature expectations for visibly different behaviors. One important direction is to explore representation learning to find these features using Feature construction for Inverse Reinforcement Learning [20]. Although discretizing continuous trajectories work well for the driving style tasks; it is important to make this work on continuous domains directly since all the differences in behavior cannot be captured by discretization. Methods like Guided Policy Search [19] and Continuous Inverse Optimal Control [18] allow us to extend Inverse Reinforcement learning to continuous tasks. Extending the non-parametric BCIRL algorithm using these methods is a promising direction to applications in continuous tasks.

[15] We say good demonstrations hoping that majority of them would be good since the expert tries to be consistent. Technically, it learns every behavior in the dataset.

Chapter 4

Accelerating actor-critic temporal logic motion control using reward shaping

4.1 Introduction

Reinforcement Learning allows an autonomous agent to learn complex and intended behaviors through reward maximization. However, it is often unclear as to what reward function would result in the intended behavior. One way to tackle this problem is to give the agent a positive reward only when it has completed the task at hand. However learning with sparse reward proves to be intractable. In this chapter, we focus on learning with temporal logic constraints. We express these constraints using Linear Temporal Logic (LTL). LTL allows us to specify temporal requirements such as visiting regions, periodic monitoring and staying safe. In this thesis, we explore the possibility of using an LTL specification to design the reward and thus obtain a controller that satisfies the specification.

An LTL specification can either be satisfied or not. Also, often it requires a sequence of complex actions to satisfy an LTL specification. If one were to model this as giving a high reward after the specification is satisfied, the problem becomes extremely sparse. This is more than the sparsity that is encountered with Reinforcement Learning tasks since the reward now depends on a higher dimensional augmented state - primitive states of the environment and the state of the automaton to satisfy the specification. In this chapter, we will see how we can exploit the structure offered by the actor-critic framework to aid in satisfying an LTL specification. In fact, we break down a compositional LTL specification into simpler ones and use a heuristic value function initialization which acts as a shaping function in the actor-critic framework. The value function initialized actor critic can satisfy a simple LTL specification while uninitialized actor-critic fails to ever arrive at the goal even once. We draw insights from compositional reasoning to decompose a complex specification into simpler reach-avoid tasks. Currently, we break down a specification manually to show

the working of the proposed method on a toy problem. Automatically breaking down the specifications and sub-goal identification is an ongoing work.

In the following section, we show the similarity between using reward shaping and going from policy gradients to actor-critic methods. We also prove the policy invariance in case of shaping to show that both actor critic and policy gradients would arrive at the same solution but with different rates. We derived the policy gradients previously in Section.2.3 and use the results to go from policy gradients to actor critic with only a small change. In Section.4.3, we define the problem statement. The following Section.4.4 defines the complete method with the algorithm used to satisfy the given LTL specification. It also contains the heuristic value initialization for faster convergence with the LTL specification. In the results section, we compare the actor-critic with and without the value function initialization on a grid world example.

4.2 Background

Reward Shaping

When the high-level specifications for tasks are sparse, it becomes hard for Reinforcement Learning algorithms to explore and find the reward in the first place [17]. One way to tackle this problem is to give shaping rewards to hopefully guide the agent towards learning an optimal policy faster. This shaping reward is required to follow certain conditions so that the optimal policy under the new reward specification does not change from that of the original specification [24]. The aim of reward shaping is only to affect the rate of convergence and not exactly the point of convergence.

$$\tilde{r}(s, a, s') = r(s, a, s') + \underbrace{\gamma\xi(s') - \xi(s)}_{\text{shaping term}} \tag{4.1}$$

where $\hat{r}$ is the shaped reward, r is the true reward of original definition and $\xi : S \to \mathbb{R}$ is an arbitrary potential function [1]. Assuring that the shaping term takes the form as in the above equation, we can show that it does not change the optimal policy (argument that maximizes the new Q value) since the Q value for all actions from a given state are shifted by the same constant.

The Q-value in an MDP is the promised future return after executing an action a from state s then following policy π. Mathematically, the Q-value of a state-action pair is given by

$$Q^{\pi}(s, a) = r(s, a) + \gamma \sum_{s'} P(s'|s, a)V^{\pi}(s') \tag{4.2}$$

[1]We use the symbol ξ to denote the potential function instead of the standard ϕ in order to avoid confusion with reward features in the previous chapter

The Q-value using the shaped reward is

$$\tilde{Q}(s_0, a_0) = \mathbb{E}_{s_1,a_1,\cdots}\big[\tilde{r}(s_0, a_0, s_1) + \gamma\tilde{r}(s_1, a_1, s_2) + \cdots\big] \tag{4.3}$$
$$= \mathbb{E}\big[(r_0 + \gamma\xi(s_1) - \xi(s_0)) + \gamma(r_1 + \gamma\xi(s_2) - \xi(s_1)) + \cdots\big] \tag{4.4}$$
$$= \mathbb{E}\big[r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots\big] - \xi(s_0) \tag{4.5}$$
$$\tilde{Q}(s_0, a_0) = Q(s_0, a_0) - \xi(s_0) \tag{4.6}$$

where $r_t = r(s_t, a_t, s_{t+1})$ is the reward acquired at timestep t. We get Eq.4.5 using the telescoping sum in Eq.4.4. Eq.4.6 holds for all actions from state s_0 which ensures optimal policy invariance since

$$\pi^*(s_0) = \max_a Q^*(s_0, a) = \max_a \tilde{Q}^*(s_0, a) \tag{4.7}$$

Hence shaping rewards can be used to give a sense of direction to the agent towards the true reward without affecting the optimal policy.

Actor-Critic and Reward Shaping

In policy gradient methods, we directly optimize the expected return (also known as utility) obtained from sampled trajectories [27]. The utility of a policy with parameter θ is given by

$$U(\theta) = \sum_\tau P(\tau;\theta)R(\tau) \tag{4.8}$$

where τ is any trajectory, $P(\tau;\theta)$ is the probability of trajectory τ under the Markov Chain resulted by the policy with parameter θ, $R(\tau)$ is the discounted return under that trajectory. The above objective function can be optimized directly by taking the derivative with respect to the policy parameters as in [35]. The gradient of the utility can be obtained from the sampled trajectories using

$$\nabla_\theta U(\theta) = \sum_\tau \nabla_\theta P(\tau|\theta)R(\tau) \tag{4.9}$$
$$= \sum_\tau P(\tau|\theta)\frac{\nabla_\theta P(\tau|\theta)}{P(\tau|\theta)}R(\tau) \tag{4.10}$$
$$= \sum_\tau P(\tau|\theta)\nabla_\theta \ln P(\tau|\theta)R(\tau) \tag{4.11}$$
$$= \mathbb{E}_\tau[\nabla_\theta \ln P(\tau|\theta)R(\tau)] \tag{4.12}$$

We can empirically estimate this gradient by rolling the system out.

$$\nabla_\theta U(\theta) \approx \hat{g} = \frac{1}{m}\sum_i \nabla_\theta \ln P(\tau^{(i)}|\theta) R(\tau^{(i)}) \tag{4.13}$$

$$\mathbb{E}[\hat{g}] = \nabla_\theta U(\theta) \tag{4.14}$$

where $\hat{g}$ is the approximation of gradient computed using samples. Also, the gradient above is independent of the dynamics.

$$\nabla_\theta \ln P(\tau^{(i)}|\theta) = \nabla_\theta \ln \Big[\prod_{t=0}^{T} P(s_{t+1}^{(i)}|s_t^{(i)}, a_t^{(i)}) \pi(a_t^{(i)}|s_t^{(i)};\theta)\Big] \tag{4.15}$$

$$= \nabla_\theta \Big[\sum_{t=0}^{T} \ln P(s_{t+1}^{(i)}|s_t^{(i)}, a_t^{(i)}) + \sum_{t=0}^{T} \ln \pi(a_t^{(i)}|s_t^{(i)};\theta)\Big] \tag{4.16}$$

$$= \nabla_\theta \sum_{t=0}^{T} \ln \pi(a_t^{(i)}|s_t^{(i)};\theta) \tag{4.17}$$

$$\implies \hat{g} = \frac{1}{m}\nabla_\theta \sum_i \sum_{t=0}^{T} \ln \pi(a_t^{(i)}|s_t^{(i)};\theta) R(\tau^{(i)}) \tag{4.18}$$

$$\text{and} \quad \nabla_\theta U(\theta) = \nabla_\theta \mathbb{E}_\tau \Big[\sum_{a,s\in\tau} \ln \pi(a|s;\theta) R(\tau)\Big] \tag{4.19}$$

Further, to reduce the variance of this estimate we introduce any constant baseline b

$$\hat{g} = \frac{1}{m}\sum_i \nabla_\theta \ln P(\tau^{(i)}|\theta)(R(\tau^{(i)}) - b) \tag{4.20}$$

$$\tag{4.21}$$

We can use the fact that future actions do not affect the past rewards and introduce state dependent baseline to decrease the variance further.

$$\hat{g} = \frac{1}{m}\sum_{i=1}^{m} \Big(\sum_{t=0}^{T-1} \nabla_\theta \ln \pi(a_t^{(i)}|s_t^{(i)};\theta)\Big)\Big(\sum_{t=0}^{T-1} r(s_t^{(i)}, a_t^{(i)}) - b\Big) \tag{4.22}$$

$$= \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T-1} \left[\nabla_\theta \ln \pi(a_t^{(i)}|s_t^{(i)};\theta)\Big(\sum_{k=t}^{T-1} r(s_k^{(i)}, a_k^{(i)}) - b(s_t^{(i)})\Big)\right] \tag{4.23}$$

$$= \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{T-1} \left[\nabla_\theta \ln \pi(a_t^{(i)}|s_t^{(i)};\theta)\Big(r(s_t^{(i)}, a_t^{(i)}) + \gamma r(s_{t+1}^{(i)}, a_{t+1}^{(i)}) + \cdots - b(s_t^{(i)})\Big)\right] \tag{4.24}$$

Notice how the inner sum $r(s_t^{(i)}, a_t^{(i)}) + \gamma r(s_{t+1}^{(i)}, a_{t+1}^{(i)}) + \cdots - b(s_t^{(i)})$ is similar to the return obtained while using reward shaping except that the potential function ξ is replaced by

the baseline b. In actor-critic methods, this baseline is replaced with the value function to minimize the variance of the estimate $\hat{g}$.

$$\mathbb{E}\big[r(s_t^{(i)}, a_t^{(i)}) + \gamma r(s_{t+1}^{(i)}, a_{t+1}^{(i)}) + \cdots - b(s_t^{(i)})\big] \tag{4.25}$$

$$= \mathbb{E}\big[r_t + \gamma r_{t+1} + \cdots - V(s_t)\big] \tag{4.26}$$

$$= \mathbb{E}\big[\underbrace{(r_t + \gamma V(s_{t+1}) - V(s_t))}_{\tilde{r}_t} + \gamma \underbrace{(r_{t+1} + \gamma V(s_{t+2}) - V(s_{t+1}))}_{\tilde{r}_{t+1}} + \cdots\big] \tag{4.27}$$

$$= \mathbb{E}\big[(\tilde{r}_t + \gamma \tilde{r}_{t+1} + \cdots\big] \tag{4.28}$$

where $\tilde{r}_t$ is the shaped reward when the potential function being used is the value.

In actor-critic methods, we approximate both the policy and value. The policy is learned using the update $\theta \coloneqq \theta + \hat{g}$ and the value is learned using a TD update.

$$\hat{V}'(s) = \hat{V}(s) + \alpha(\hat{G}_s - \hat{V}(s)) \tag{4.29}$$

where $\hat{G}_s$ is the average return obtained from the roll-outs.

Often in actor-critic methods both policy and the value function are randomly initialized. We exploit the insight that value is nothing but a reward shaping mechanism added to vanilla policy gradient algorithm and come up with a value initialization to achieve faster convergence to the optimum.

4.3 Problem Overview

Given an MDP (S, A, P, R, γ) where the reward function R is extremely sparse due temporal sequence required to achieve this reward, we come up with a heuristic value function initialization that allows the agent to realize this reward. We model the temporal sequence required to achieve this reward using LTL formula φ. Hence we have an automaton $(\mathcal{Q}, \Sigma, \delta, q_0, F)$ where, $\mathcal{Q}$ is the set of automaton states, Σ is the alphabet, $\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ is the transition function, q_0 is the start state and F is the accept states. Fig.4.1 shows the automaton states and the transitions. We need to design a heuristic function that takes makes the system to have a high probability of satisfying the specification or reaching the accept states F. Our approach is to exploit compositional reasoning to break down the problem of satisfying the specification into sequentially satisfying sub-goals. Since each sub-goal is a simple reach-avoid task, we can initialize the reward shaping function as a heuristic function for sub-goals and then compose them to obtain a shaping function for the global specification.

4.4 Method

In this section, we explain the working of our method with a grid world example. Here we manually decompose a given specification into subtasks and show that the critic initialization

can solve the task much faster than vanilla actor-critic. In fact, without critic initialization, the actor-critic never converged.

We experiment using a grid world of dimensions 15×10 with obstacles O and with three regions R_1, R_2, and R_3. The agent can move through the obstacle but would receive a negative reward of -5. We want to synthesize a policy which satisfies the specification

$$\varphi = \big(\Diamond(R_1 \wedge R_3) \vee \Diamond(R_2 \wedge R_3)\big) \wedge \Box\neg O \tag{4.30}$$

where φ is the specification, $\Diamond$ is read as *eventually*, $\vee$ is read as *or*, $\wedge$ is read as *and*, $\Box$ is read as always, and $\neg$ is read as *not*. The specification φ says that we must visit either region R_1 or R_2, then reach the region R_3 to complete the task. Fig.4.1 shows the complete specification. This specification can be broken down into two subtasks

Task 1:

$$\varphi_1 = \Diamond(R_1 \vee R_2) \wedge \Box\neg O \tag{4.31}$$

Task 2:

$$\varphi_2 = \Diamond R_3 \wedge \Box\neg O \tag{4.32}$$

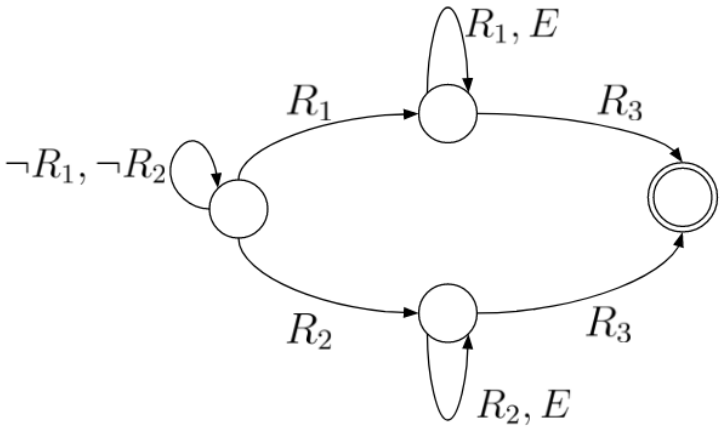


Figure 4.1: Automata showing the transitions for our specification $\varphi = \big(\Diamond(R_1 \wedge R_3) \vee \Diamond(R_2 \wedge R_3)\big) \wedge \Box\neg O$

If we were to assign a positive reward only when the specification is satisfied, then we would have the sparse reward problem since there is a temporal logic to be satisfied. Hence we need to direct the agent towards completion using the subtasks $Task1$ and $Task2$. We chose the following heuristic function.

$$\hat{V}(s, q) = \max_i(\gamma^{||s - g_i(q)||_1}) \tag{4.33}$$

where $g_i(q)$ is the i^{th} sub goal in a subtask when the state of the automata is q. For example, in $Task1$ there are two sub goals R_1 and R_2. In actor critic, both policy and values are parametrized. We parametrize the policy with one parameter for each action a from each augmented state $(S \times \mathcal{Q})$ using $\theta_{s,q,a}$ where $s \in S$, $q \in \mathcal{Q}$, and $a \in A$. The symbol θ denotes the vector of all the parameters $\theta_{s,q,a}$ for all the combinations if $S \times \mathcal{Q} \times A$. We will also use the symbols z and (s, a) interchangeably to denote an element in the augmented state space $S \times \mathcal{Q}$

$$\pi(a|s, q) = \frac{e^{\theta_{s,q,a}}}{\sum_{a_i} e^{\theta_{s,q,a_i}}} \tag{4.34}$$

Algorithm

We initialize all the weights $\theta_{s,q,a}$ for all states and actions of the policy to zero. From this initialization we perform the actor critic algorithm [16].

1. Initialize values and policy according to Eq.4.33 and Eq.4.34 respectively.
2. Roll-out the system m times for $T + 1$ time steps to obtain the trajectories

   $$(z_0^{(i)}, a_0^{(i)}, r_0^{(i)}), (z_1^{(i)}, a_1^{(i)}, r_1^{(i)}), \cdots (z_T^{(i)}, a_T^{(i)}, r_T^{(i)})$$

   where $z_t^{(i)} \in S \times \mathcal{Q}$ is the augmented state and i is the index of each trajectory
3. Update policy using (refer section.2.3 for complete derivation)

   $$\theta \coloneqq \theta + \alpha \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \left[\nabla_\theta \ln \pi(a_t^{(i)}|z_t^{(i)}; \theta) \Big(r(z_t^{(i)}, a_t^{(i)}) + \gamma V(z_{t+1}^{(i)}) - V(s_t^{(i)}) \Big) \right]$$

   where α is the learning factor of the policy function
4. Update the value function using (refer section.2.3 for explanation)

   $$V(z_t^{(i)}) \coloneqq V(z_t^{(i)}) + \beta(r(z_t^{(i)}, a_t^{(i)}) + \gamma V(z_{t+1}^{(i)}) - V(z_t^{(i)}))$$

5. Go to step.2

4.5 Experiments and Results

Experimental Setup

We test our algorithm on a 15×10 grid world as shown in Fig.4.2. We give the agent a reward of $+10$ if the specification $\varphi = \big(\Diamond(R_1 \wedge R_3) \vee \Diamond(R_2 \wedge R_3)\big) \wedge \Box\neg O$ is satisfied. For

moving into the blue region in Fig.4.2, we penalize the agent with a reward of -5. The agent gets a reward of 0 all other time. We run the Algorithm.4.4 for 20 times with and without the value function initialization. Fig.4.3 shows the mean reward accrued versus the iterations of the above algorithm.

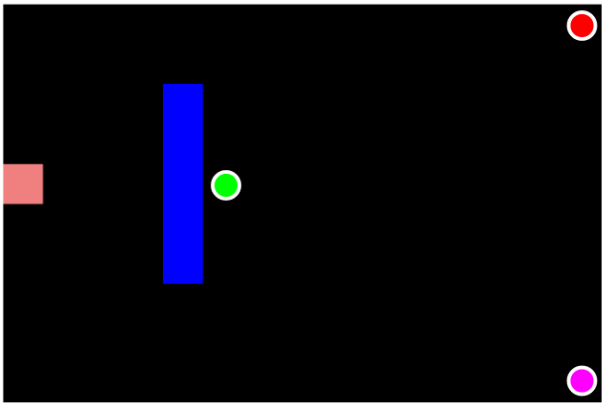


Figure 4.2: The agent is shown in light coral, the region R_1 in the specification is shown in green, the region R_2 is shown in pink, the region R_3 is shown in red, and the obstacles are shown in blue. Notice how the agent has to go around the obstacles to reach R_1 then R_3.

We can see that the agent, without initialization, quickly finds a way to not bump into the obstacle. In fact, the agent gets stuck in the top left corner of the grid world and does not explore since it tries to avoid the penalties that it might incur. In the case with value function initialization, the agent, though it first bumps into the obstacle, is motivated not to stay in a corner since the value difference at the corner and the neighboring cells give a signal towards the real goal. This can be seen as a downward spike at the beginning of the curve in Fig.4.3(a)

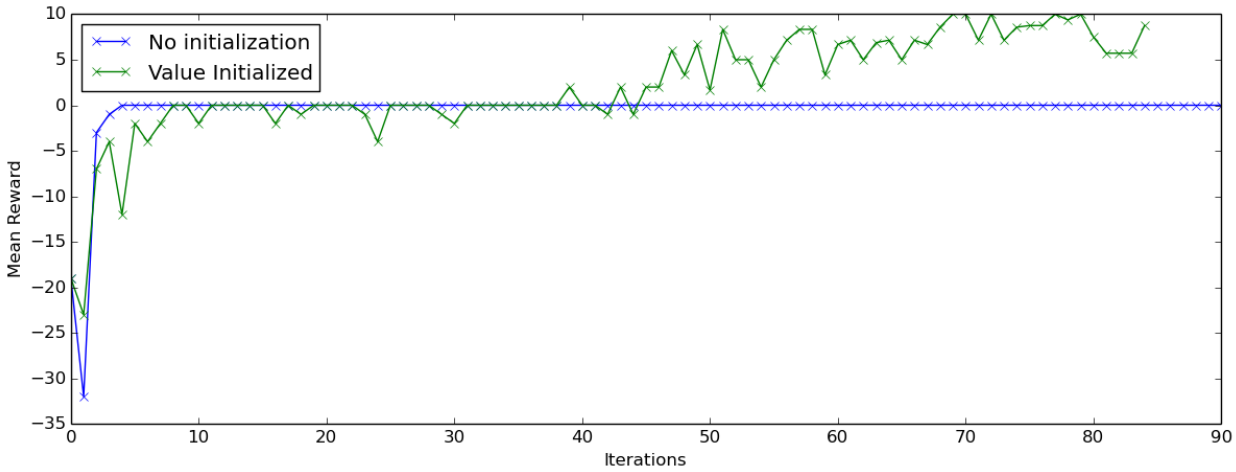
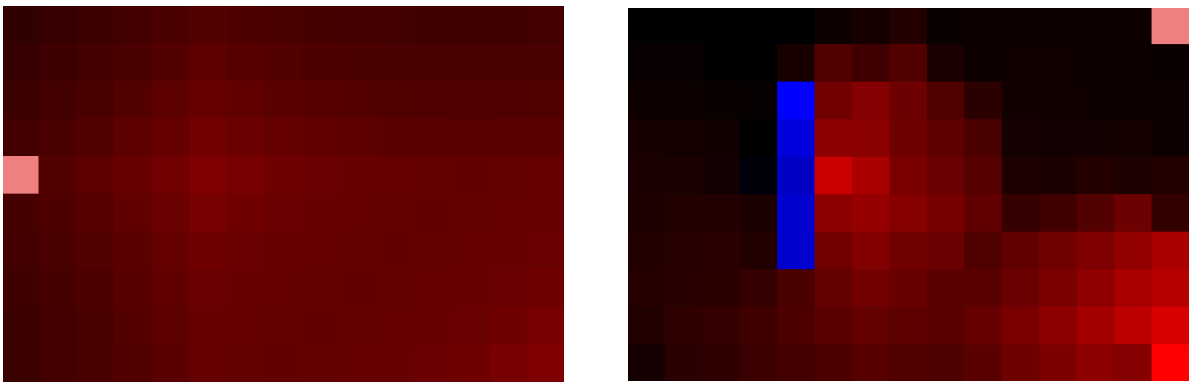


Figure 4.3: The mean reward vs. the number of iterations of the algorithm.4.4. Notice how with value initialization the agent is able to satisfy the specification φ. Without the initialization, the agent can quickly find a way not to explore and avoid the penalty of running into obstacles.

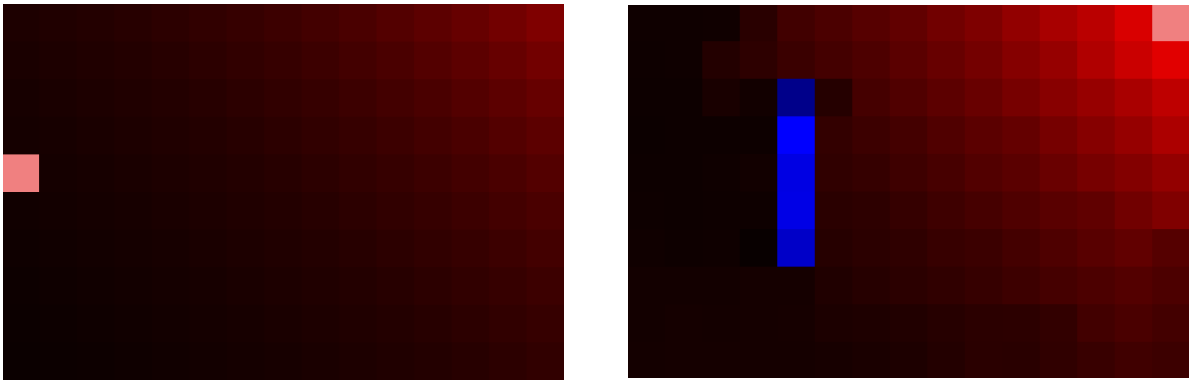
The value function initialization is shown in Fig.4.4(a). Note how the value increases as we move towards both region R_1 and R_2 in the initialization. In Fig.4.4(b), the final learned value has a very similar structure. We do not associate any positive reward for visiting either of the regions and yet we see that the agent expects (through value function) that the future reward is high by moving to these regions.



(a) Initialized value function (b) Learned value function

Figure 4.4: This is the value function before visiting either region R_1 or region R_2, that is, for the $Task1$. In this plot, blue represents a low negative value and red represents a high positive value.

In Fig.4.5, we see that the learned and the initialized value function increase as we move towards region R_3 after having visited either R_1 or R_2.



(a) Initialized value function

(b) Learned value function

Figure 4.5: This is the value function after visiting either region R_1 or region R_2, that is, for the $Task2$. In this plot, blue represents a low negative value and red represents a high positive value.

4.6 Conclusion and Future Work

We presented a method that exploits the structure in actor critic algorithms to learn with sparse reward specifications such as satisfying Linear Temporal Logic (LTL) constraints. We demonstrated the method on a simple example with simple and intuitive heuristic value function initialization. We compared the working of actor-critic with and without the initialization in several runs and showed that while the former can find a solution in few episodes, the latter does not complete the task even once. One of the promising directions is to break down compositional specifications to give value shaping function autonomously. This way we can help scale the method to more complex tasks.

Appendix A

Algorithms

A.1 The EM Algorithm

In this section, we will review the formulation and derivation of convergence of EM algorithm [26]. Assume we obtain samples of data from a distribution parametrized by θ. Assume we have some missing information y in the sample and we only observe a part of the data x. For instance, z may be a vector sampled from a parameterized distribution while y is some missing components of the vector z. The components x and y would together form the complete data point z. We also assume that we have access to the prior on the joint likelihood of observed and missing parts of the data i.e., $P(x, y|\theta)$. We would like to maximize the log likelihood of observed data x with respect to the parameter of the distribution θ since we do not have any information on y.

$$\max_{\theta} \ln p(x|\theta) \tag{A.1}$$

EM is used especially when it is hard to optimize the above problem explicitly, but it is easier to optimize the joint likelihood $P(x, y|\theta)$. The objective in Eq. A.1, for any probability distribution q, can be written as

$$\ln p(x|\theta) = \int q(y) \ln \frac{p(x, y|\theta)}{q(y)} dy + \int q(y) \ln \frac{q(y)}{p(y|x, \theta)} dy \tag{A.2}$$

$$= \int q(y) \ln \frac{p(x, y)q(y)}{p(y|x, \theta)q(y)} dy \tag{A.3}$$

$$= \int q(y) \ln p(x|\theta) dy \tag{A.4}$$

$$= \ln p(x|\theta) \int q(y) dy \tag{A.5}$$

$$= \ln p(x|\theta) \tag{A.6}$$

Hence optimizing Eq. A.1 is equivalent to optimizing the RHS of Eq. A.2.

$$\arg\max_{\theta} \ln p(x|\theta) = \arg\max_{\theta} \underbrace{\int q(y) \ln \frac{p(x,y|\theta)}{q(y)} dy}_{\text{Likelihood}=\mathcal{L}(x,\theta)} + \underbrace{\int q(y) \ln \frac{q(y)}{p(y|x,\theta)} dy}_{\text{KL divergence}} \tag{A.7}$$

We will use the fact that the KL divergence is a metric which is 0 only if $q(y) = p(y|x,\theta)$ and > 0 otherwise for the proof of convergence of EM.

The Algorithm

E-step: Given $\theta^{(t)}$, the value of parameter θ at iteration t, set $q_t(y) = p(y|x,\theta^{(t)})$. This makes the KL divergence at iteration t go to 0. The log likelihood is now given by,

$$\mathcal{L}_t(x,\theta) = \int p(y|x,\theta^{(t)}) \ln p(x,y|\theta) dy - \underbrace{\int p(y|x,\theta^{(t)}) \ln p(y|x,\theta^{(t)}) dy}_{\text{Independent of } \theta} \tag{A.8}$$

M-step: Set $\theta^{(t+1)} = \arg\max_\theta \mathcal{L}_t(x,\theta)$

Proof of convergence

$$\begin{aligned}
\ln p(x|\theta^{(t)}) &= \mathcal{L}(x,\theta^{(t)}) + \underbrace{KL\Big(q_t(y)||p(y|x,\theta^{(t)})\Big)}_{=0 \text{ by setting } q_t=p} \\
&= \mathcal{L}_t(x,\theta^{(t)}) \quad \leftarrow \textbf{E-step} \\
&\leq \mathcal{L}_t(x,\theta^{(t+1)}) \quad \leftarrow \textbf{M-step} \\
&\leq \mathcal{L}_t(x,\theta^{(t+1)}) + \underbrace{KL\Big(q_t(y)||p(y|x,\theta^{(t+1)})\Big)}_{>0 \text{ because q} \neq p} \\
&= \mathcal{L}(x,\theta^{(t+1)}) + KL\Big(q_t(y)||p(y|x,\theta^{(t+1)})\Big) \\
&= \ln p(x|\theta^{(t+1)})
\end{aligned}$$

Hence EM guarantees monotonic improvement.

A.2 Dirichlet Process to Estimate the Number of Agents

In the problem of clustering behaviors and learning their reward functions, consider the case where we do not know the number of behaviors beforehand. This is similar to using Dirichlet

Process Mixture Models (DPMM) [22] instead of K-mean clustering which required the hyperparameter K. Our problem exactly falls in the Chinese Restaurant Process category.

Dirichlet Process is a probability distribution in which every sample is a probability distribution (it is a probability distribution on probability distributions). This is similar to Gaussian Processes in a way (where every sample point is a function). A Dirichlet process has two parameters: the scaling parameter α and the base distribution G_0. Let G_0 be the base distribution on the measurable set X, then every sample G from $DP(\alpha, G_0)$ is a distribution over discrete points in the set. Every time, we can sample a G from $DP(\alpha, G_0)$, then sample x_i from G. To find the distribution of the points x_i directly, we can marginalize out the G

$$P(x_1, ..., x_N) = \int P(G|\alpha, G_0) \prod_{i=1}^{N} P(x_i|G) dG$$

Marginalizing out G introduces dependencies between variables and makes the distribution simple to express,

$$x_N|x_1, x_2, ..., x_{N-1} = \begin{cases} x_i & \text{with probability} \frac{1}{n-1+\alpha} \\ \text{new draw from } G_0 & \text{with probability} \frac{\alpha}{n-1+\alpha} \end{cases}$$

A.3 Chinese Restaurant Process (CRP)

Given an infinite number of tables in a restaurant, the probability a new customer joins a table T_j, for some j, is proportional to the number of people sitting in that table. Also, with probability proportional to α, a parameter of the distribution, the customer will choose to sit in a new table. Each table T_j is same as the value that each x_i is associated with. Though there are a large number of samples, most of the time, most samples will take same values.

Let $N-1$ tables be currently occupied, let a new customer choose a table T_N.

$$P(T_N = T_i : i \sim [1, 2, .., N-1]|T1, ..., T_{N-1}) = \frac{num(T_i)}{n-1+\alpha} \tag{A.9}$$

$$P(T_N = \text{new table}|T1, ..., T_{N-1}) = \frac{\alpha}{n-1+\alpha} \tag{A.10}$$

where $num(T_i)$ is the number of people in i^{th} table and n is the total number of customers including the new one and N is the total number of tables.

Our problem is similar to Chinese Restaurant Process except that instead of having the number of individuals sitting in a table, we have the probability masses assigned to each behavior. Let us denote the event of j^{th} class being assigned to a randomly picked

demonstration as c_j and the class assignment to every other demonstration as c_{-j}. We can rewrite the prior as,

$$c_j|c_1, c_2, ..., c_n = \begin{cases} c_k & \text{with probability } \frac{\sum_i \beta_{ik}}{\sum_i \sum_j \beta_{ij}+\alpha} = \frac{\sum_i \beta_{ik}}{n+\alpha} \\ \text{new draw from } G_0 & \text{with probability} \frac{\alpha}{n+\alpha} \end{cases} \tag{A.11}$$

Note that in the above problem, we consider all n demonstrations to compute the prior as opposed to $n-1$ in Eq. A.9. We can initialize the clusters with some random assignment if we have some knowledge about the system for faster convergence. Anyway, the Dirichlet prior will converge for any assignment.

Bibliography

[1] Learning to drive a bicycle using reinforcement learning and shaping.

[2] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1. ACM, 2004.

[3] Richard Bellman. The theory of dynamic programming. Technical report, RAND CORP SANTA MONICA CA, 1954.

[4] Dimitri P Bertsekas. Dynamic programming and optimal control 3rd edition, volume ii. 2011.

[5] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[6] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 39–57. IEEE, 2017.

[7] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[8] Shreyansh Daftry, J Andrew Bagnell, and Martial Hebert. Learning transferable policies for monocular reactive mav control. In *International Symposium on Experimental Robotics*, pages 3–11. Springer, 2016.

[9] Lih-Yuan Deng. The cross-entropy method: a unified approach to combinatorial optimization, monte-carlo simulation, and machine learning, 2006.

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

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

[12] Samuel J Gershman and David M Blei. A tutorial on bayesian nonparametric models. *Journal of Mathematical Psychology*, 56(1):1–12, 2012.

[13] Chad Goerzen, Zhaodan Kong, and Bernard Mettler. A survey of motion planning algorithms from the perspective of autonomous uav guidance. *Journal of Intelligent and Robotic Systems*, 57(1-4):65, 2010.

[14] Abhijit Gosavi. Reinforcement learning: A tutorial survey and recent advances. *INFORMS Journal on Computing*, 21(2):178–192, 2009.

[15] Evan Greensmith, Peter L Bartlett, and Jonathan Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5(Nov):1471–1530, 2004.

[16] Ivo Grondman, Lucian Busoniu, Gabriel AD Lopes, and Robert Babuska. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1291–1307, 2012.

[17] Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning.

[18] Sergey Levine and Vladlen Koltun. Continuous inverse optimal control with locally optimal examples. *arXiv preprint arXiv:1206.4617*, 2012.

[19] Sergey Levine and Vladlen Koltun. Guided policy search. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1–9, 2013.

[20] Sergey Levine, Zoran Popovic, and Vladlen Koltun. Feature construction for inverse reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1342–1350, 2010.

[21] Sergey Levine, Zoran Popovic, and Vladlen Koltun. Nonlinear inverse reinforcement learning with gaussian processes. In *Advances in Neural Information Processing Systems*, pages 19–27, 2011.

[22] Peter McCullagh, Jie Yang, et al. How many clusters? *Bayesian Analysis*, 3(1):101–120, 2008.

[23] Andrew Y Ng. *Shaping and policy search in reinforcement learning*. PhD thesis, University of California, Berkeley, 2003.

[24] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping.

[25] Andrew Y Ng, Stuart J Russell, et al. Algorithms for inverse reinforcement learning.

[26] John Paisley. Columbiax-machine learning. In *https://courses.edx.org/courses/course-v1:ColumbiaX+CSMM.102x+1T2017/courseware*.

[27] Jan Peters and Stefan Schaal. Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4):682–697, 2008.

[28] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014. pg. 134.

[29] Morgan Quigley, Josh Faust, Tully Foote, and Jeremy Leibs. Ros: an open-source robot operating system.

[30] Kristin Y Rozier. Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163–203, 2011.

[31] Stuart Russell and Peter Norvig. A modern approach. 1995.

[32] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[33] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1.

[34] Gautham Vasan and Patrick M Pilarski. Learning from demonstration: Teaching a myoelectric prosthesis with an intact limb via reinforcement learning. *IEEE-RAS-EMBS International Conference on Rehabilitation Robotics*, 2017.

[35] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[36] Brian D Ziebart. *Modeling purposeful adaptive behavior with the principle of maximum causal entropy*. Carnegie Mellon University, 2010.

[37] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning. 2008.