Robust Adversarial Inverse Reinforcement Learning with Temporally Extended Actions

David Venuto, School of Computer Science

McGill University, Montreal

August, 2020

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Science

Abstract

Explicit engineering of reward functions for given environments has been a major hindrance to reinforcement learning methods. While inverse reinforcement learning is a solution to recover reward functions from demonstrations only, these learned rewards are generally heavily entangled with the dynamics of the environment and therefore not portable or robust to the changing environments. Modern adversarial methods have yielded some success in reducing reward entanglement but only for a single policy. Options provide a framework to obtain better reward generalization through breaking problems into task-specific small policies, which leads to better generalization of the underlying policy. In this work, we leverage Adversarial Inverse Reinforcement Learning to create a novel algorithm that learns disentangled rewards with a policy over options. We show that this method has the ability to create portable reward functions in highly complex transfer learning tasks, while yielding results in continuous control benchmarks that are comparable to those of the state-of-the-art methods.

Abrégé

L'ingénierie explicite des fonctions de récompense pour des environnements donnés a été un obstacle majeur pour les méthodes d'apprentissage par renforcement. Bien que l'apprentissage par renforcement inverse représente une solution pour récupérer les fonctions de récompense uniquement à partir de démonstrations, ces récompenses apprises sont généralement fortement intriquées avec la dynamique de l'environnement et donc non portables ou robustes dans des environnements dont les dynamiques sont changées. Les méthodes adversariales modernes ont réussi à réduire l'intrication des récompenses, mais seulement pour une seule politique. Les options fournissent un cadre pour obtenir une meilleure généralisation des récompenses en décomposant les problèmes en petites politiques spécifiques à la tâche. Cela conduit ainsi à une meilleure généralisation de la politique sous-jacente. Dans ce travail, nous tirons parti de l'apprentissage par renforcement inverse adversarial pour créer un nouvel algorithme qui apprend les récompenses démêlées avec une politique sur les options. Nous montrons que cette méthode a la capacité de créer des fonctions de récompense portables dans des tâches d'apprentissage de transfert très complexes, tout en donnant des résultats comparables à ceux des méthodes de pointe dans les environnements continus classiques.

Acknowledgements

Most of all, I thank my supervisor Doina Precup for being very encouraging and teaching me so much and guiding this work. Jhelum Chakravorty was also instrumental in helping me develop my ideas and experiments. Junhao Wang and Leonard Boussioux were also important in helping implement the methods and experiment code.

Contribution of Authors

Chapters 1, 2, 3, 4, 5, 6 - Written by David Venuto. Designed and formulated the method oIRL. Chapters 1, 2, 3, 4, 5, 6 - Comments and edits made by Doina Precup and Jhelum Chakravorty. Jhelum Chakravorty also helped with the convergence proof in Chapter 4. Chapter 4, 5 - Leonard Boussioux and Junhao Wang helped design some of the Grid environments, code and helped run experiments. In all chapters I received guidance and advice on my ideas by Doina Precup and Jhelum Chakravorty. Both helped formulate the method and ideas mathematically. Doina Precup advised this project.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reinforcement learning (RL) algorithms have been used to learn policies in complex environments, but they usually require designing suitable reward functions for successful learning. This can be difficult and may lead to learning sub-optimal policies with unsafe behavior [2] in the case of poor engineering. Inverse Reinforcement Learning (IRL) [1,27] can facilitate such reward engineering through learning an expert's reward function from expert demonstrations.

IRL, however, comes with many difficulties and the problem is not well-defined because, for a given set of demonstrations, the number of optimal policies and corresponding rewards can be very large, especially for high dimensional complex tasks. Also, many IRL algorithms learn reward functions that are heavily shaped by environmental dynamics. Rewards learned on such reward functions may not remain optimal with slight changes in the environment. Adversarial Inverse Reinforcement Learning (AIRL) [16] generates more *generalizable* policy learning with environmentally *disentangled reward* functions that are invariant to the environmental dynamics. The reward and value function are learned simultaneously to compute the reward function which depends only on states. An instance of a *transfer learning problem with changing dynamics* is where the agent learns an optimal reward function in one environment and then transfers it to an environment with different dynamics. A practical example of this transfer learning problem

would be teaching a robot to walk with some mechanical structure, and then generalize this knowledge to perform the task with differently sized structural components.

In complex tasks with many demonstrations that can often be explained by several different reward functions, methods such as Maximum Entropy IRL and GAN-Guided Cost Learning (GAN-GCL) tend to overfit [14]. One way to help solve the problem of overfitting is to break down a policy into small *option* (temporally extended action) policies that solve various aspects of an overall task. This method has been shown to be able to create policies that transfer better when the task changes [35, 36]. Methods such as Option-Critic have implemented modern RL architectures with a policy over options and have shown improvements in generalization compared to usual RL algorithms [3]. OptionGAN [20] also proposed an IRL framework for a policy over options and showed some improvement in one-shot transfer learning tasks, but it is not able to construct disentangled rewards.

In this thesis, we introduce Option-Inverse Reinforcement Learning (oIRL) [40], to investigate transfer learning with options. Following the AIRL framework, we propose an algorithm that computes disentangled rewards to learn joint reward-policy options, with each option having rewards that are disentangled from environmental dynamics. The proposed approach is also relatively easy to implement. The rewards produced are portable in transfer learning tasks. We evaluate this method in a variety of continuous control tasks in the Open AI Gym environment using the MuJoCo simulator [6, 38]. We also experiment with grid worlds constructed with MiniGrid [7]. Our method shows improvements in terms of performance on a variety of transfer learning tasks, while still performing better than benchmarks for standard continuous and non-continuous control tasks.

The main impact we see for this work is in applications to robotics tasks. Options have already been used in previous work to provide good control ability in robotics tasks [23,25]. Combining this framework with the ability to learn generalizable rewards should further increase the ability to transfer knowledge in robotic tasks.

The thesis is structured as follows. In Chapter 2 we provide necessary background on sequential decision making, including reinforcement learning. In Chapter 3 we review background on deep learning architectures. Chapter 4 provides the main contribution of the thesis. Experimental results are presented in Chapter 5. Finally, in Chapter 6 we provide conclusions and discuss future work.

Chapter 2

Sequential Decision Making

In this chapter, we provide background on reinforcement learning, including tabular RL, policy gradient methods, and temporal abstraction. We will also give some background on inverse reinforcement learning.

2.1 Markov Decision Process

A Markov Decision Process (MDP) is a mathematical framework used to model sequential decision making. It is a process, where at each time step, an *agent* that is at a *state* chooses an *action*, which causes a transition to a new *state*. The agent gains *reward* at each transition by interacting with the *environment*. The transition to the next state and the reward can be stochastic.

An MDP is defined as a tuple $\langle \mathcal{S}, \mathcal{A}, r, \mathbb{P} \rangle$, where $\mathcal{S}$ is a set of states that the agent can be in, $\mathcal{A}$ is a set of actions that the agent can take at each state, $r : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is real-valued reward function, $\mathbb{P}(s'|s, a)$ is the probability of transitioning to state $s' \in \mathcal{S}$ given that the agent is in state $s \in \mathcal{S}$ and takes action $a \in \mathcal{A}$.

By definition, an MDP has the Markov property: the transitions are only conditioned upon the current state and action, and not on past history. This is more formally stated

as,

$$P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0). \tag{2.1}$$

2.1.1 Policies

A policy π is a probability distribution that maps states to actions. It can be thought of as the agent's behavior as it traverses the MDP. It is defined formally as $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$. A stochastic policy is a conditional probability distribution for actions given states $\pi(a|s)$.

The main goal in reinforcement learning is to find a policy that is *optimal* for the given environment that the agent interacts with, π^*. We say that a policy is optimal when it is better in terms of its expected reward than all other policies.

2.1.2 Value Functions

In reinforcement learning methods, we wish to define some metrics for comparing different policies. The value function of a state s is the expected discounted sum of rewards (return) for a given policy π. It is defined as,

$$V_\pi(s) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s]. \tag{2.2}$$

The scalar $\gamma \in [0, 1]$ is a discount factor. It is used to define the importance of current rewards compared to future rewards. If γ is 1, we call this an un-discounted process where current and future rewards have the same importance. The smaller γ is, the more an agent values current rewards compared to future rewards (more discounting).

The optimal value function is the value function with the highest value at all states: $V^*(s) = \max_\pi V_\pi(s)$.

Since it is difficult to look at the return in terms of the entire future (horizon), the Bellman equation [5] defines the value function recursively in terms of value functions of

the next state as,

$$V_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} P(s'|s,a)[r(s,s',a) + \gamma V_\pi(s')]. \tag{2.3}$$

We can additionally define the state-action value function as, $Q_\pi(s,a) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a]$, where the agent's initial action a is allowed to deviate from policy π. The corresponding Bellman equation is defined as,

$$Q_\pi(s,a) = \sum_{s' \in \mathcal{S}} P(s'|s,a)[r(s,s',a) + \gamma V_\pi(s')]. \tag{2.4}$$

2.2 Learning Value Functions

While the Bellman equations provide a framework to compute value functions when a model of the environment is known, such a model can be hard to estimate. Ideally, an agent should be able to improve these value function estimates and its policy directly, as it explores the environment. A simple method to do this takes a Monte Carlo approach. In this approach, trajectories are sampled and then the average of the discounted sum of rewards for each state is used to estimate the state value function.

Temporal difference learning [33] allows the agent to change its value function estimates after each step of experience, rather than waiting for the end of a trajectory. Q-learning [43] can be used to learn value estimates and improve the policy at the same time. We now briefly review all these methods. We would like to note that we assume in these two algorithms (as we are describing them next) that we have a sufficiently small and finite set of states and actions. This is called the *tabular setting*.

2.2.1 Temporal Difference (TD) Learning

One strategy to learn value functions requires utilizing the Bellman equations to minimize the difference of the estimated state value function between two consecutive time

steps in a trajectory [33]. In the Bellman equation for the state value function (2.3), the value of the next state is an expectation over all possible states. The agent is able to sample a state from that distribution through its interaction with the environment, without explicitly computing the transition probabilities between the two states. Methods that do not explicitly compute the transition probabilities are referred to as *model free*.

Given a sampled state s' from the agent acting in the environment, the error we wish to minimize is the Temporal Difference error defined as,

$$\text{TD-Error} = r(s, s', a) + \gamma V_\pi(s') - V_\pi(s). \tag{2.5}$$

The TD-learning algorithm is described in Algorithm 1. The termination condition is typically either a maximum number of time steps, or the fact that the changes in the value function fall below a pre-specified threshold. The learning rate $\alpha \in (0, 1)$ controls the speed of the learning algorithm and is a hyper-parameter that needs to be optimized for best results.

Algorithm 1 Tabular TD-Learning

Input: Initial $V(s)$, Learning Rate: α

repeat
  $s \leftarrow s_0$
  while s' is not terminal **do**
    $a \leftarrow \pi(a|s)$
    Observe s', r from applying a on environment
    $V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s))$
    $s \leftarrow s'$
  end while
until Termination condition is met

2.2.2 Q-Learning

Q-learning [43] uses one policy to obtain trajectories from the environment and then uses those trajectories to improve the policy and estimate the state-action value function. The Q-learning algorithm is outlined in Algorithm 2. The learning rate and termination condi-

tion are like in TD. The policy used to generate actions usually takes the action currently estimated to have the highest value (also called greedy action) with the highest probability, and ensures that all actions are taken with some non-zero probability. This can be ensured, for example, by taking the greedy action with probability $(1 - \epsilon)$ and a uniformly random action with probability ϵ. This approach is called ϵ-Greedy policy. There are many strategies used to randomize action choices.

Algorithm 2 Tabular Q-learning

Input: Initial $Q(s, a)$, Learning Rate: α

repeat
  $s \leftarrow s_0$
  while s' is not terminal **do**
    $a \leftarrow \epsilon$-Greedy Policy from Q
    Observe s', r from applying a on environment
    $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
    $s \leftarrow s'$
  end while
until Termination condition is reached

2.3 Function Approximation

As stated, the algorithms discussed previously operate in the *tabular setting*. Since the state space is finite and sufficiently small, we can think of the state s as an index into a vector of values. This is highly impractical for a few reasons. In this setting, we assume that there is no information sharing between states, although in many cases states can be similar to each other, and sharing information is desirable. Also, state spaces can be very large and are therefore impossible to handle in the tabular setting. For example, in an arcade game, every possible combination of pixels that makes up every possible frame in the game would be considered a different state. Any single dimension is continuous in this system.

It is, therefore, useful to learn a function that takes state or action features as input and returns approximate state values. We can have parameterized state value and state-

action value functions $V_\theta(s)$ and $Q_\theta(s, a)$ respectively. Instead of memorizing state values for every input, we learn a mapping for these values which can then predict values for unseen states. This greatly speeds up the training process while the derivation of many methods remains similar to the tabular setting.

2.4 Learning policies with policy gradients

Policy gradient methods [34, 44] learn parameterized policies directly instead of first learning value functions and then optimizing policies. These methods typically learn a parameterized stochastic policy $\pi_\theta(a|s)$. This set of methods is beneficial when value functions are difficult to approximate.

2.4.1 REINFORCE

REINFORCE [44] is a *policy search* method that learns a policy directly from sampled trajectories. The policy is then updated using an estimate of the gradient with respect to a specific loss objective. The objective, in this case, is the expected cumulative discounted reward,

$$J(\theta) = \mathbb{E}_{\tau,\pi_\theta}[\sum_{t=1}^{T} r(s_t, a_t)|s_0], \tag{2.6}$$

where T is the trajectory length and τ is a trajectory. We will also denote $G(\tau)$ as the expected discounted return for a trajectory τ. Since we have an objective, we can now

take the gradient to obtain a gradient ascent update rule for the policy parameters θ,

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \mathbb{E}_{\tau,\pi_\theta}[G(\tau)] \\
&= \nabla_\theta \int_\tau P(\tau|\pi_\theta) G(\tau) d\tau \\
&= \int_\tau \nabla_\theta \Big(P(\tau|\pi_\theta) G(\tau) \Big) d\tau \\
&= \int_\tau \Big(G(\tau) \nabla_\theta P(\tau|\pi_\theta) + \nabla_\theta G(\tau) P(\tau|\pi_\theta) \Big) d\tau \\
&= \int_\tau \Big(\frac{G(\tau)}{P(\tau|\pi_\theta)} \nabla_\theta P(\tau|\pi_\theta) + \nabla_\theta G(\tau) \Big) P(\tau|\pi_\theta) d\tau.
\end{aligned} \tag{2.7}$$

We then apply a log derivative trick after factoring out $P(\tau|\pi_\theta)$. This is a rule which states $\frac{\nabla_\theta P(x|\theta)}{P(x|\theta)} = \nabla_\theta \log P(x|\theta)$. We turn our integral into an expectation and obtain,

$$= \mathbb{E}_{\tau,\pi_\theta} \Big[G(\tau) \nabla_\theta \text{log}(P(\tau|\pi_\theta)) + \nabla_\theta G(\tau) \Big]. \tag{2.8}$$

Now, we see that $G(\tau)$ is not a function of θ. Therefore, $\nabla_\theta G(\tau) = 0$. In the end, we have,

$$= \mathbb{E}_{\tau,\pi_\theta} \Big[G(\tau) \nabla_\theta \text{log}(P(\tau|\pi_\theta) \Big]. \tag{2.9}$$

We can write out the sequence of transitions taking advantage of the Markov property and obtain,

$$\begin{aligned}
\nabla_\theta \text{log} \Big(P(\tau|\pi_\theta) \Big) &= \nabla_\theta \text{log} \Big[p(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t) \pi_\theta(a_t|s_t) \Big] \\
&= \nabla_\theta \text{log} \Big(p(s_0) \Big) + \sum_{t=0}^{T-1} \nabla_\theta \text{log} \Big(P(s_{t+1}|s_t, a_t) \Big) + \nabla_\theta \text{log} \Big(\pi_\theta(a_t|s_t) \Big) \\
&= \sum_{t=0}^{T-1} \nabla_\theta \text{log} \Big(\pi_\theta(a_t|s_t) \Big),
\end{aligned} \tag{2.10}$$

where $p(s_0)$ is the probability of starting in state s_0. The one-step equivalent removes the summation and is therefore

$$\nabla_\theta J(\theta) = \mathbb{E}_{s_t,\pi_\theta}\Big[G(s_t)\nabla_\theta \log(\pi_\theta(a_t|s_t))\Big]. \tag{2.11}$$

We can now optimize the policy directly with gradient ascent, using the update rule as,

$$\theta_{k+1} = \theta_k + \alpha\nabla_\theta \log\Big(\pi_{\theta_k}(a_t|s_t)\Big)G(s_t), \tag{2.12}$$

where k is the iteration of the algorithm.

2.4.2 Policy Gradient

Policy gradient methods [34] optimize the parameterized policy π_θ using gradient ascent just as we saw in REINFORCE. Instead of using the discounted return over an entire trajectory, we can use value function estimates learned through any estimation method, such as TD or Monte Carlo. Taking the same ideas from the derivation of REINFORCE, we must maximize,

$$\rho(s_t) = \sum_{a\in\mathcal{A}} \pi_\theta(a|s_t)Q_{\pi_\theta}(s_t,a) = V_{\pi_\theta}(s_t). \tag{2.13}$$

Taking the gradient of this objective results in

$$\begin{aligned}
\nabla_\theta V_{\pi_\theta}(s_t) &= \nabla_\theta \sum_{a\in\mathcal{A}} \pi_\theta(a|s_t)Q_{\pi_\theta}(s_t,a) \\
&= \sum_{a\in\mathcal{A}} \Big[\nabla_\theta \pi_\theta(a|s_t)Q_{\pi_\theta}(s_t,a) + \pi_\theta(a|s_t)\nabla_\theta Q_{\pi_\theta}(s_t,a)\Big] \\
&= \sum_{a\in\mathcal{A}} \Big[\nabla_\theta \pi_\theta(a|s_t)Q_{\pi_\theta}(s_t,a) + \pi_\theta(a|s_t)\nabla_\theta (r(s_t,a) + \gamma\sum_{s'\in\mathcal{S}} P(s'|s,a)V_{\pi_\theta}(s'))\Big] \\
&= \sum_{a\in\mathcal{A}} \Big[\nabla_\theta \pi_{\theta_t}(a|s_t)Q_{\pi_\theta}(s_t,a) + \pi_\theta(a|s_t)\gamma\sum_{s'\in\mathcal{S}} P(s'|s,a)\nabla_\theta V_{\pi_\theta}(s')\Big].
\end{aligned} \tag{2.14}$$

If we unroll $\sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s_t) Q_{\pi_\theta}(s_t, a)$, it becomes apparent that,

$$\nabla_\theta V_{\pi_\theta}(s_t) = \sum_{x\in\mathcal{S}} \sum_{s\in\mathcal{S}} \gamma^k P(s_t \to x, k, \pi_\theta) \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|x) Q_{\pi_\theta}(x, a), \tag{2.15}$$

where $P(s \to x, k, \pi_\theta)$ denotes the probability of going from s to state x in k steps under the policy π_θ.

If we wish to examine this gradient from the initial state, it is helpful to define the stationary distribution,

$$d^{\pi_\theta}(s) = \sum_{k=0}^{\infty} P(s_t \to s, k, \pi_\theta), \tag{2.16}$$

which is the discounted weighting of states encountered if starting at initial state s_0. We now have a gradient,

$$\nabla_\theta V_{\pi_\theta}(s_t) = \sum_{s_t\in\mathcal{S}} d^{\pi_\theta}(s_t) \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s_t) Q_{\pi_\theta}(s_t, a). \tag{2.17}$$

It is difficult and expensive to take the sum over all actions in the gradient as the action space becomes large. We can perform the following derivation by introducing a log to reduce the summation over action gradients to an expectation, which allows us to sample actions instead of calculating the gradient for every action,

$$\begin{aligned}
\nabla_\theta V_{\pi_\theta}(s_t) &= \sum_{s_t\in\mathcal{S}} d^{\pi_\theta}(s_t) \sum_{a\in\mathcal{A}} \frac{\pi_\theta(a|s_t)}{\pi_\theta(a|s_t)} \nabla_\theta \pi_\theta(a|s_t) Q_{\pi_\theta}(s_t, a) \\
&= \sum_{s_t\in\mathcal{S}} d^{\pi_\theta}(s_t) \sum_{a\in\mathcal{A}} \pi_\theta(a|s_t) \nabla_\theta \frac{\pi_\theta(a|s_t)}{\pi_\theta(a|s_t)} Q_{\pi_\theta}(s_t, a) \\
&= \sum_{s_t\in\mathcal{S}} d^{\pi_\theta}(s_t) \sum_{a\in\mathcal{A}} \pi_\theta(a|s_t) \nabla_\theta \log\Big(\pi_\theta(a|s_t)\Big) Q_{\pi_\theta}(s_t, a).
\end{aligned} \tag{2.18}$$

In Equation 2.18, we applied the same log of a derivative trick as seen in Equation 2.8 by introducing $\frac{\pi_\theta(a|s_t)}{\pi_\theta(a|s_t)}$. This results in the gradient,

$$\nabla_\theta V_{\pi_\theta}(s_t) = \sum_{s_t\in\mathcal{S}} d^{\pi_\theta}(s_t) \mathbb{E}_{a\in\mathcal{A}}\Big[\nabla_\theta \log\Big(\pi_\theta(a|s_t)\Big) Q_{\pi_\theta}(s_t, a)\Big]. \tag{2.19}$$

We can also add a baseline (B) to reduce variance. The state-value function is commonly used as a baseline. Policy gradient with a baseline takes on the form,

$$\sum_{s_t \in \mathcal{S}} d^{\pi_\theta}(s_t) \mathbb{E}_{a \in \mathcal{A}} \Big[\nabla_\theta \log \Big(\pi_\theta(a|s_t) \Big) [Q_{\pi_\theta}(s_t, a) - B] \Big]. \tag{2.20}$$

Sometimes it is useful to learn an estimate of the value function and use sampled state-action pairs to update the gradient. If we have a sampled trajectory and an estimate of the value function at time step t, $\hat{Q}_t$, we end up with the objective,

$$J(\theta) = \mathbb{E}_t \Big[\nabla_\theta \log(\pi_\theta(a|s)) \hat{Q}_t(s, a) \Big]. \tag{2.21}$$

2.4.3 Actor-Critic

As described previously, it is useful to learn both an estimate of the value function and the policy. Actor-critic algorithms [9] maintain two gradients, one for the policy, similar to REINFORCE as 'the actor', and one for the estimate of the value function as 'the critic', using a method like TD. The gradient of the actor is described in the previous section (Equation 2.19) and the gradient of the critic is approximated as,

$$\nabla_\beta Q_\beta(s, a) = \nabla_\beta \Big(R(s, a) + \gamma \mathbb{E}_{s' \sim P(s'|s,a)} [V(s') - Q_\beta(s, a)] \Big), \tag{2.22}$$

where our state-action value function is parameterized by β. In the on-policy setting, we simply train using samples taken by taking actions according to the current policy in the environment. The resulting algorithm is given in Algorithm 3. Note that Q_β does not actually have to be the exact state-action value function, and in particular can include an action-independent baseline, for the purpose of reducing variance. A popular choice is to use the advantage function,

$$A(s, a) = Q(s, a) - V(s). \tag{2.23}$$

Algorithm 3 Online Actor Critic with Q-value function

Input: Initial $Q_\beta(s,a), \pi_\theta$, Learning Rate: α_1, α_2

repeat
  $s \leftarrow s_0$
  while s' is not terminal **do**
    $a \leftarrow \pi_\theta(a|s)$
    Observe s', r from applying a on environment
    $\beta \leftarrow \beta + \alpha_1((r + \gamma V(s') - Q_\beta(s,a))\nabla_\beta Q_\beta(s,a))$
    $\theta \leftarrow \theta + \alpha_2 \nabla_\theta \log \pi_\theta(a|s) Q_\beta(s,a)$
    $s \leftarrow s'$
  end while
until Termination condition is reached

2.4.4 Proximal Policy Optimization

In Proximal Policy Optimization [29], we have a Monte Carlo REINFORCE style policy gradient. We also use an estimate of the advantage function as the value function.

We set up a probability ratio between the new and old parameterized policy as $\rho_t(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{\text{old}}}(a|s)}$. It is easy to see that if we maximize this objective without constraint it would lead to large policy updates. PPO modifies this object and the resulting optimization problem is

$$J^{\text{CLIP}}(\theta) = \mathbb{E}_\tau\Big[\min(\rho_t(\theta)\hat{A}_t(s_t,a_t), \text{clip}(\rho_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t(s_t,a_t))\Big], \tag{2.24}$$

where ϵ is a hyper-parameter. In this objective, we clip the probability ratio, which prevents ρ_t from moving outside of $[1-\epsilon, 1+\epsilon]$ at each update. The lower bound is the unclipped objective.

The advantage function is estimated using Generalized Advantage Estimation (GAE) [28] as,

$$\hat{A}_t = \sum_{i=0}^{T} (\gamma\lambda)^i \delta_{t+i}, \tag{2.25}$$

where $\delta_{t+i} = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma V(s_{t+i})$. This method to estimate the advantage reduces the variances of updates.

2.5 Imitation Learning

Humans learn many behaviors by observing a demonstrator acting optimally and then imitating this behavior. In RL, this concept is captured by imitation learning algorithms[4].

In the imitation learning setting, an agent observes a set of state-action trajectories from an expert demonstrator (who we assume is acting according to the optimal policy). We let $\mathcal{T}_{\mathcal{D}} = \{\tau_1^E, \tau_2^E, \ldots, \tau_n^E\}$ be the state-action trajectories of the expert, where $\tau_i^E = \{s_0, a_0, s_1, a_1 \ldots, s_k, a_k\}$. We wish to learn the policy of the expert by leveraging these demonstrations.

2.5.1 Inverse Reinforcement Learning

Inverse Reinforcement Learning (IRL) [1, 27] is an imitation learning method where the agent first recovers the expert's reward function and then learns its own optimal policy using the estimated expert reward function. IRL algorithms maintain an estimate of the reward function $\hat{r}(s, a)$ and update this estimate at each iteration.

In IRL, it is assumed that the expert acts optimally with respect to its reward, i.e.,

$$\mathbb{E}_{\pi_E}\left[\sum_t \gamma^t \hat{r}\left(s_t, a_t\right)\right] \geq \mathbb{E}_{\pi}\left[\sum_t \gamma^t \hat{r}\left(s_t, a_t\right)\right] \quad \forall \pi, \tag{2.26}$$

where π_E is the optimal expert policy, π is any policy.

The agent learns an optimal policy to maximize $\mathbb{E}_{\pi_A}\left[\sum_{t=0}^{\infty} \gamma^t \hat{r}\ \left(s_t, a_t\right)\right]$, where π_A is the agent's policy. The reward function can be a parameterized function approximator $r_\theta(s_t, a_t)$.

Another interpretation of IRL is solving the maximum likelihood problem,

$$\max_\theta \mathbb{E}_{\tau \sim \mathcal{D}}[\log p_\theta(\tau)], \tag{2.27}$$

with $p_\theta(\tau) \propto p(s_0) \prod_{t=1}^{T} p(s_{t+1}|s_t, a_t) e^{\gamma^t r_\theta(s_t, a_t)}$. We are learning a parametrized reward function $r_\theta(s_t, a_t)$ in this formulation. Another interpretation of $p_\theta(\tau)$ in IRL for arbitrary

cost function $c_\theta(\tau)$ is based off a Boltzmann distribution where the energy model is given by

$$p_\theta(\tau) = \frac{1}{Z_\theta}\exp(-c_\theta(\tau)), \tag{2.28}$$

where Z_θ is the intractable partition function defined by the integral $\int_\tau \exp(-c_\theta(\tau))$ [13].

2.5.2 Reward Ambiguity

In Fu et al. [16], the authors showed why IRL methods fail to learn robust reward functions, due to the problem of reward *ambiguity* described below.

Given an arbitrary function $\Phi : \mathcal{S} \to \mathbb{R}$, in Ng et al. [26] it is shown that only for the reward function transformation,

$$\hat{r}(s, a, s') = r(s, a, s') + \gamma\Phi(s') - \Phi(s), \tag{2.29}$$

the optimal policy will remain the same. In IRL, we only learn rewards from demonstrations of the agent following the optimal policy, and therefore IRL cannot disambiguate between reward functions within the above transformation types.

The resulting shaped rewards are not robust to changes in transition dynamics [16]. To show this, Fu et al. first defines MDPs M and M' that both have the same reward function but have differing transition dynamics T and T' respectively.

Suppose that we perform IRL with an algorithm that finds a shaped reward $\hat{r}(s, s', a)$ under M with $\Phi \neq 0$. Now we have MDPs M and M' where if we change the transition model, we break policy invariance on M'. To simplify, let's say that we have deterministic dynamics $(s, a) \to s' = T(s, a)$ and state-action rewards. We can therefore write the reward transformation as $\hat{r}(s, a, s') = r(s, a, s') + \gamma\Phi(T(s, a)) - \Phi(s)$. By changing the dynamics T to T' such that $T(s, a) \neq T'(s, a)$, the reward $\hat{r}(s, a)$ is not policy invariant for M'.

2.5.3 Disentangled Rewards

Disentangled Rewards refer to reward functions $r^*_\theta(s, a, s')$ such that under all possible dynamics $T \in \mathcal{T}$, the optimal policy computed with respect to the reward function is the same. Hence, the reward is disentangled from the dynamics.

2.6 Temporal Abstraction

Options [35] are a mathematical framework for hierarchical reinforcement learning. An agent now has a higher-level policy, a policy over options, denoted by π_Ω, which chooses among multiple option policies (denoted by π_ω). At the initial state, the agent chooses a temporally extended option policy, and then follows this policy for a period of time until the option terminates. The agent can then select a new option policy to follow given the policy over options. The corresponding concept in psychology is to solve separate short problems with a set of policies [31].

The policy over options is $\pi_\Omega : \mathcal{S} \times \omega \to [0, 1]$. An option is defined by an intra-option policy $\pi_\omega : \mathcal{S} \times \mathcal{A} \to [0, 1]$ and a termination function $\beta_\omega : \mathcal{S} \to [0, 1]$ which gives the probability of terminating the option policy at each state. Optionally, an initiation set contains all the states that an option can start from $I_\omega \subseteq \mathcal{S}$.

Options are able to break down a large task into many sub-problems and therefore improve policy generalization and learning speed.

2.6.1 Semi-Markov Decision Processes

An intra-option policy will be executed for some number of steps depending on the state termination probabilities. Options are applied at every timestep in the MDP, so the past information (the option you are currently following) does affect the future and therefore we do not have the Markov property. MDPs with options are called Semi Markov Decision Processes.

It is useful to redefine transition probabilities in terms of options [3, 35]. At each step we have an additional consideration, we can continue following the policy of the current option we are in or terminate the option with some probability, sample a new option from the inter-option policy and follow that option's policy. We have therefore have transition probabilities,

$$P(s_{t+1}, \omega_{t+1}|s_t, \omega_t) = \sum_{a\in\mathcal{A}} \pi_\omega(a|s_t)P(s_{t+1}|s_t, a)((1-\beta_{\omega_t}(s_{t+1}))\mathbf{1}_{\omega_t=\omega_{t+1}}+ \tag{2.30}$$
$$\beta_{\omega_t}(s_{t+1})\pi_\Omega(\omega_{t+1}|s_{t+1})),$$

and,

$$P(s_{t+1}, \omega_{t+1}|s_t) = \sum_{\omega\in\Omega} \pi_\Omega(\omega|s_t) \sum_{a\in\mathcal{A}} \pi_\omega(a|s_t)P(s_{t+1}|s_t, a)((1-\beta_{\omega_t}(s_{t+1}))\mathbf{1}_{\omega_t=\omega_{t+1}}+ \tag{2.31}$$
$$\beta_{\omega_t}(s_{t+1})\pi_\Omega(\omega_{t+1}|s_{t+1})),$$

and,

$$P(s_{t+1}, w_{t+1}|s_t, \omega_t, a_t) = P(s_{t+1}|s_t, a_t)((1-\beta_{\omega_t}(s_{t+1}))\mathbf{1}_{\omega_t=\omega_{t+1}}+ \tag{2.32}$$
$$\beta_{\omega_t}(s_{t+1})\pi_\Omega(\omega_{t+1}|s_{t+1})),$$

where $\mathbf{1}_{w_t=\omega_{t+1}}$ is the a probability of 1 of following option ω at timestep $t+1$. Termination of an option is a Bernoulli random variable dependent on the state and option. Each option is a stochastic distribution with actions sampled dependent on states.

We can define an option Q-function [3] as a state option value function and write out the probability of selecting each action,

$$Q_\Omega(s, \omega) = \sum_{a\in\mathcal{A}} \pi_{\omega,\alpha}(a|s)Q_U(s, \omega, a), \tag{2.33}$$

where Q_U is defined below. The state-option-action value function is an expectation over states with a utility term $U(\omega, s')$ that sums the probability of an option terminating or not. If options terminate, we have the state value of the next state weighted by the next option sampled from the policy over options. If we continue the option, we have the value function of the next state and option. This is defined as,

$$Q_U(s, \omega, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) U(\omega, s'), \tag{2.34}$$

where,

$$U(\omega, s') = (1 - \beta_{\omega,\delta}(s')) Q_\Omega(s', \omega) + \beta_{\omega,\delta}(s') V_\Omega(s'). \tag{2.35}$$

2.6.2 Intra-Option Policy Gradient

We must compute a policy gradient for a single option where we can maximize the expected return of a state and option. The expected return to optimize is given as,

$$\mathbb{E}[\sum_{t=1}^{\infty} \gamma^{t-1} R_t | s_0, \omega_0, \Omega] = Q_\Omega(s_0, \omega_o). \tag{2.36}$$

The intra-option policy gradient is similar in derivation to the previous policy gradients. For these derivations, the intra-option policies are parameterized by α for each option, and the option termination probabilities by δ. Taking the derivative state-option value function yields

$$\nabla_\alpha Q_\Omega(s, \omega) = \sum_{a \in \mathcal{A}} \nabla_\alpha \pi_{\omega,\alpha}(a|s) Q_U(s, \omega, a) + \sum_{a \in A} \pi_{\omega,\alpha}(a|s) \sum_{s' \in \mathcal{S}} \gamma P(s'|s, a) \nabla_\alpha U(\omega, s'). \tag{2.37}$$

Expanding $\nabla_\alpha U(\omega, s')$ yields,

$$\nabla_\alpha U(\omega, s') = \sum_{\omega' \in \Omega} \Big((1 - \beta_{\omega,\delta}(s')) \mathbf{1}_{\omega'=\omega} + \beta_{\omega,\delta}(s') \pi_\Omega(\omega'|s') \Big) \nabla_\alpha Q_\Omega(s', \omega'). \tag{2.38}$$

Finally, it is shown that the gradient of the expected return for the initial state and option (s_0, ω_0) is

$$\nabla_\alpha Q_\Omega(s_0, \omega_0) = \sum_{s,\omega \in \mathcal{S},\Omega} \mu_\Omega(s, \omega | s_0, \omega_0) \sum_{a \in \mathcal{A}} \nabla_\alpha \pi_{\omega,\alpha}(a|s) Q_U(s, \omega, a), \tag{2.39}$$

with $\mu_\Omega(s, \omega | s_0, \omega_0) = \sum_{t=0}^{\infty} \gamma^t P(s_t = s, \omega_t = \omega | s_0, \omega_0)$ (similar to traditional policy gradient).

Chapter 3

Deep Learning

In this chapter, we introduce Deep Learning and the background information on using it for RL problems. Deep learning is a set of machine learning algorithms that utilize artificial neural networks (ANNs). ANNs can learn non-linear functions by using multiple layers to extract higher-level features from the input. This class of models is frequently used in RL methods.

3.1 Function Approximation and Supervised Learning

In machine learning problems, we assume the presence of a data generating distribution, which we denote as p_{data} and we have some samples of observations of data from this distribution. The data is paired samples $(x, y) \sim p_{\text{data}}$ and a fundamental problem is to learn the mapping between $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ which are both of fixed size. The variable x is frequently referred to as a set of features (frequently we reduce x to a vector $x \in \mathbb{R}^n$). The variable y is referred to as the *label* and is frequently a one-hot encoded vector when our outputs are categorical.

The core problem in *supervised learning* is to learn a *good* mapping $f : \mathcal{X} \to \mathcal{Y}$. The goal is for every pair of points (x, y) that can possibly be generated from p_{data}, we want our function to have $f(x) = y$.

In supervised learning, it is assumed that pairs (x, y) are collected from independently and identically distributed random variables (i.i.d.). Therefore if we are given a set of $\{x_1, x_2, \ldots, x_n\} \sim x$ data points, we can use these points as an unbiased estimator of $\mathbb{E}[X]$ and therefore learn a mapping that will generalize to unseen points, not in the fixed set but generated from p_{data}.

As we saw previously, tabular methods in RL provide a framework to store values for every possible state, but as the state space becomes large we must use function approximation. The same idea applies to supervised learning frameworks. For a small fixed set of $\{x_i, \ldots, x_n\}$ values we can store the corresponding labels (y), but as the set becomes large we need a function approximator to learn a mapping function. The mapping function (f) can take on a set of heuristics or a wide range of algorithms, but it is most common to learn a function that depends on some parameters $\theta = \{\theta_1, \theta_2, \ldots, \theta_n\}$ with $f_\theta : \mathcal{X} \to \mathcal{Y}$. The core goal in machine learning methods is to learn the parameter θ which best learns the mapping from feature to labels for all possible data generated from p_{data}.

3.2 Linear Modelling

We will first introduce the linear regression model, which is not actually a deep learning model but is fundamental to understanding the concepts of deep learning. These are simple models which learn a function $f : \mathbb{R}^n \to \mathbb{R}^m$ (or output $\mathbb{R}$ for scalar labels). Linear models generally require a matrix of weights $\mathbf{W} \in \mathbb{R}^{(n \times m)}$ (the learned parameters in the case of this model), an input vector $\mathbf{x} \in \mathbb{R}^{(n \times 1)}$, and an output $\mathbf{y} \in \mathbb{R}^{m \times 1}$. A *bias* term $\beta \in \mathbb{R}^{m \times 1}$ is also usually learned. The function takes the form of

$$f(\mathbf{x}) = \mathbf{W}^T \mathbf{x} + \beta, \tag{3.1}$$

where we wish to have $f(\mathbf{x}_i) = \mathbf{y}_i$ for all possible point pairs $(\mathbf{x}_i, \mathbf{y}_i) \sim p_{\text{data}}$ and find the appropriate $\mathbf{W}$ to achieve this.

To solve this problem, we are generally given a size n set of data generated from p_{data} denoted as $D = \{(\mathbf{x}_i, \mathbf{y}_i), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\}$. We then wish to minimize some arbitrary measure of performance or *loss*. In the derivations we will say that f is parameterized by $\theta = \{\mathbf{W}, \beta\}$. A common objective which is independent of positive or negative differences is the mean squared error (MSE) loss,

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} ||f(x_i; \theta) - y_i||^2. \tag{3.2}$$

The function has a better *fit* to the dataset D for lower values of the loss (but not necessarily our true goal p_{data} as we will see later) and our optimization goal to find the optimal θ is

$$\theta_{\text{optimal}} = \arg\min_{\theta} J(\theta). \tag{3.3}$$

3.3 Neural Networks

We have now seen simple linear functions that can represent a wide class of regression problems, but some more complex data distributions can only be represented as nonlinear functions. Neural networks can learn complex nonlinear functions (shown to be able to learn *any* function) [17].

They are biologically inspired by the architecture of neurons in the human brain. They receive input, combine this input with the neuron state (this is called activation) and then learn an optimal threshold to decide when to output information. Human neurons receive electrochemical input at dendrites, process this information in the soma, and output an electric signal at the axon.

A neural network contains a set of stacked linear models in each layer, these are the *neurons*. These neurons take any number of inputs and create an output which is a composition of the outputs of previous layer. Each neuron has a nonlinear activation function which gives the output of the layer. A layer has a set of neurons that take input from the

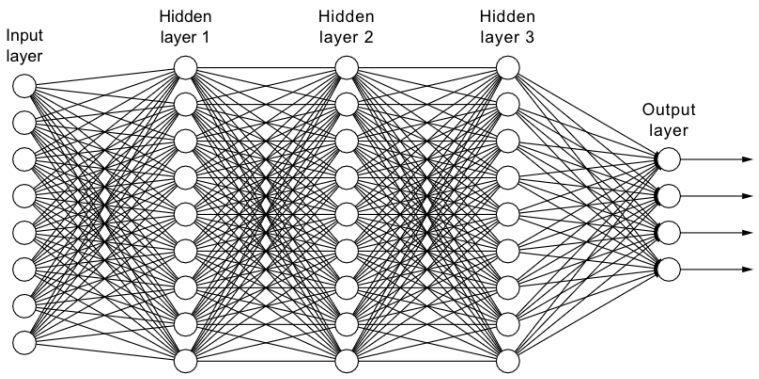


Figure 3.1: An example of a fully connected neural network with an output layer. Image from: Probabilistic Deep Learning With Python, Keras and TensorFlow Probability [12].

previous layer (or initial feature values for the initial layer). This is explained in the next section. A single neuron has the form

$$h(\mathbf{x}) = \sigma(\sum_i w_i x_i + b), \tag{3.4}$$

where w_i is the weight for input position i in this particular neuron, b is a bias term and $x_i \in \mathbf{x}$ is a vector of input values to the neuron from the previous layer. An *activation function* is denoted by σ.

We can represent the output of an entire layer in a neural network as a vector $\mathbf{x}_h$. The term $\mathbf{W}_h$ denotes a matrix of weight vectors for each neuron for the layer h. The vector $\mathbf{b}$ is a vector of bias terms for each neuron in the layer. In a neural network with initial input $\mathbf{x}_0$, we have,

$$\mathbf{x}_h = \sigma(\mathbf{W}_h \mathbf{x}_{h-1} + \mathbf{b}_h). \tag{3.5}$$

Before training a neural network, it is required to choose the number of layers and the number of neurons in each layer. If all neurons in all layers are connected to every other neuron in the next layer, we have a fully connected neural network. The final layer is of fixed output. A fully connected neural network is shown in Figure 3.1.

3.3.1 Activation Functions

As described, each layer has a non-linear activation function that is applied to the output of the neurons (the weight matrix in the layer ($\mathbf{W}_h$) multiplied by an input vector $\mathbf{x}$) in every layer.

Activation functions sometimes restrict the output domain to $[0, 1]^n$ or $[-1, 1]^n$. The sigmoid function, denoted by $\sigma(x)$, is bounded by 0 and 1 so it is often used when probability valued outputs are required. It is also smooth and easily differentiable. We will see later how differentiable activation functions are required for gradient learning methods. The sigmoid function is defined as,

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{3.6}$$

The sigmoid function has a derivative that can be found as a function of the input,

$$\nabla_x \sigma(x) = \sigma(x)(1 - \sigma(x)). \tag{3.7}$$

The hyperbolic tangent function (denoted as tanh) is also smooth and differentiable but bounded by -1 and 1. It has a similar property of having a derivative that is a function of its input. It is defined as,

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{3.8}$$

The ReLU function is a very simple activation function which guarantees that the output is larger than 0. It is defined as,

$$\text{ReLU}(x) = \max(0, x). \tag{3.9}$$

In the case of the sigmoid and tanh activation functions, the gradient becomes very small when x is either very large or very small. This is not optimal for gradient-based learning and increases training time (This is discussed in Section 3.4). In the case of ReLU

activation, the gradient is large and consistent at any time the neuron is active ($x > 0$) in terms of output.

The softmax activation function is applied to the entire layer output and it compares each neuron output value to the rest of outputs in the layer. It defined as,

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}, \tag{3.10}$$

where n is the number of outputs in the layer. The softmax function is extremely useful as it can be interpreted as a probability with $\text{softmax}(\mathbf{x})_i = p_i$ and $\sum_{i=1}^{n} p_i = 1$. It can be used as the output of a classifier to represent a probability distribution of each output over n different classes.

3.4 Learning Deep Neural Networks

The most common method for learning the parameters in neural networks (shown thus far as the weight values and bias terms) are gradient descent methods. Similar to gradient descent methods used in reinforcement learning, we must define some objective to use in our optimization problem.

We can think of our neural network as a parametric model $p(\mathbf{y}|\mathbf{x};\theta)$ and optimize using a maximum likelihood estimation (MLE) method. We are therefore using the cross-entropy between the training data and model distribution as a cost function and take the negative log-likelihood given by,

$$J(\theta) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}}[\log p_{\text{model},\theta}(\mathbf{y}|\mathbf{x})]. \tag{3.11}$$

In deep learning, we commonly use a cross-entropy loss where a log function *undoes* the exponent in the softmax, therefore preventing gradients from exploding. For a vector

of predictions $\hat{\mathbf{y}}$ made by the model, our cross entropy loss $L(\mathbf{y}, \hat{\mathbf{y}})$ is defined as,

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i y_i \log(\hat{y_i}). \tag{3.12}$$

Our goal is now to optimize our model parameters using Gradient Descent .

3.4.1 Gradient Descent

Here we will formally define Gradient Descent. Given a function $f(\mathbf{x})$ with input vector $\mathbf{x} \in \mathbb{R}^n$, we wish to find the value of $\mathbf{x}$ that minimizes $f(\mathbf{x})$. The gradient of f with respect to $\mathbf{x}$ is a vector of first order partial derivatives which is $\nabla_{\mathbf{x}} f(\mathbf{x}) = [\frac{\partial f}{\partial \mathbf{x}_1}, \frac{\partial f}{\partial \mathbf{x}_2}, \ldots, \frac{\partial f}{\partial \mathbf{x}_n}]$. To find the value of $\mathbf{x}$, we use the Gradient Descent algorithm (described in Algorithm 4), which iteratively takes small steps at a *learning* rate of α using the gradient of our objective. These steps are said to be in the direction which minimizes our objective since they are in the direction of the gradient. Repeating this iterative procedure for a large number of steps, we converge to our optimal $\mathbf{x}$ which minimizes $f(\mathbf{x})$ as long as these steps are small enough. The step size is determined by α and is used to scale the magnitude of the change at each step.

Algorithm 4 Gradient Descent

Input: Randomly initialized $\mathbf{x}$

1: **repeat**
2: $\quad \mathbf{x} \leftarrow \mathbf{x} - \alpha \nabla_{\mathbf{x}} f(\mathbf{x})$
3: **until** $\mathbf{x}$ has converged (Termination condition is met)

Typically, in actual implementations of Gradient Descent, we say that $\mathbf{x}$ has converged when at each iteration the norm of $\mathbf{x}$ has not changed more than some value ϵ. There are many other methods that exist to determine when to stop gradient descent [10].

For a convex optimization problem, gradient descent will converge to a global minimum. In non-convex optimization problems like in neural network optimization, we are only guaranteed to converge to a local optimum for small step sizes.

3.4.2 Differentiable Models in Neural Networks

To perform gradient descent, we must be able to take the gradient of the neural network objective with respect to the parameters. It is possible to write a non-linear neural network model as a differentiable function. Here we will show how to write a differentiable loss function for our feed-forward neural network and take the gradient with respect to the weights (parameters).

We will first rewrite our loss function in terms of a simple 1 layer neural network output $f(\mathbf{x}) = \sigma(\mathbf{x}^T\mathbf{W} + \mathbf{b})$, where $\mathbf{W}$ is a matrix of weights for neurons in the layer and x is an input vector. The cross-entropy loss is

$$L_{\mathbf{W}}(\mathbf{x}) = -\sum_i y_i \log(f(\mathbf{x})_i). \tag{3.13}$$

To calculate the gradients of the weights, we exploit the chain rule in calculus to perform *backpropagation* [41]. This allows us to *propagate* the gradient from the output loss to the first layer weights. If we want to find the gradient of the loss with respect to the weights in our 1-layer network, we have,

$$\frac{\partial L_{\mathbf{W}}(\mathbf{x})}{\partial \mathbf{W}} = \frac{\partial L_{\mathbf{W}}(\mathbf{x})}{\partial f(\mathbf{x})} \frac{\partial \sigma(\mathbf{x}^T\mathbf{W} + \mathbf{b})}{\partial (\mathbf{x}^T\mathbf{W} + \mathbf{b})} \frac{\partial (\mathbf{x}^T\mathbf{W} + \mathbf{b})}{\partial \mathbf{W}}. \tag{3.14}$$

It is easy now see that we can perform this chain rule computation with any number of layers. With h layers and each layer having output $f_h(\mathbf{x}) = \sigma(\mathbf{W}_h\mathbf{x}_{h-1} + \mathbf{b}_h)$ we will now find our gradient with respect to $\mathbf{W}_1$. We have the computation,

$$\frac{\partial L_{\mathbf{W}}(\mathbf{x})}{\partial \mathbf{W}_1} = \frac{\partial L_{\mathbf{W}}(\mathbf{x})}{\partial f_h(\mathbf{x}_h)} \prod_{i=1}^{h-1} \Big(\frac{\partial f_i(\mathbf{x}_{h-i})}{\partial \mathbf{x}_{h-i}} \Big) \frac{\partial f_1(\mathbf{x}_1)}{\partial \mathbf{W}_1}, \tag{3.15}$$

where $\mathbf{W}_h$ is the weight matrix of the h-th layer.

We perform a similar process calculating gradients with respect to the weight vectors of each layer. Each layer requires gradients calculated for the following layer, so we can

go backward through our network and calculate the gradient for all weights. In turn, we find the optimal weights to minimize the loss function and therefore increase the fit of our model to the dataset trained on.

3.4.3 Regularization

The goal in machine learning problems is to model the distribution of the generator p_{data}, but we are only given a sample dataset that is generated by this generator. If we make our model very complex, we can perfectly model our dataset but may not generalize well to unseen data produced from the generator. With a very large number of layers and neurons, we will create a highly complex model. It is, therefore, an important issue to decide on an appropriate level of model complexity in a neural network.

In machine learning, we generally separate data into a *training*, *validation*, and *testing* set. We train the model parameters using the training set. After training, we use our models to make predictions on the validation set to determine how well we generalize, and then finally report model performance on a small testing set.

A model f_θ has low bias on a dataset D (same notation as used previously) if the predictions made by the model ($\hat{y}_i = f_\theta(x_i)$) are very good in terms of being accurate with respect to every data point y_i. High bias is the opposite (predictions are far away from the labels).

If we retrain our model on a smaller subset of the training set (by removing a few examples from our initial training set) and the parameters of the model have large changes after training, then our model is high variance. Complex models that are sensitive to small changes in the training set and are said to be over-fitting. Models that over-fit do not generalize well to unseen data. On the other hand, models that under-fit generally have high bias and therefore generalize too much.

When we say that our model is complex, we are referring to the hyperparameters of the model. These are parameters that cannot be optimized directly (such as the number of layers in an ANN). To optimize our hyperparameters, we generally test a variety of

models with different hyperparameters on our validation set and then pick the one with the best performance.

Regularization also allows us to prevent over-fitting. A regularizer can control the optimization so we have some property that we wish to achieve. An example of a property could be the optimal model complexity. This can be achieved by not allowing the model to use every input feature (therefore resulting in a less complex model).

,The most commonly used regularizer is called *weight decay*. This adds an L1 or L2 regularization term to the loss and has been heavily studied in statistics literature [37]. The L2 regularization term is the squared norm of the parameter vector is

$$L_{\text{L2-norm}} = \sum_i \theta_i^2. \tag{3.16}$$

The L1 regularization term is the sum of the absolute values of every parameter θ_i denoted as $L_{\text{L1-norm}} = \sum_i |\theta_i|$. The resulting cross entropy loss is

$$J(\theta) = -\sum_i y_i \log(f_\theta(\mathbf{x}_i)) + \lambda \sum_j \theta_j^2, \tag{3.17}$$

where hyper parameter λ determines the weight of the L2 regularization. The result of regularization is a model that will not learn very large weight values.

Another very simple way of performing regularization is to introduce *dropout* to a network [32]. This is a method in which we randomly ”drop” certain activations (add 0 in place of the activation output) with some probability. This results in an increase in model robustness. Two neurons cannot coordinate with each other explicitly and the model must learn a more redundant representation.

3.5 Generative Adversarial Networks

Generative Adversarial Networks (GANs) [18] is a method that uses neural networks in an adversarial framework to estimate a generative model.

In the generative modelling problem, we have some data generator $\mathbf{x} \sim p_{\text{data}}$ and access to a dataset of samples from this generator $X \sim p_{\text{data}}$ with $\mathbf{x} \in X$. We wish to learn a model which approximates the distribution of p_{data} and can be used to generate samples which come from that distribution such that $Y \sim p_{\text{model}}$ and X is very close to Y.

In the GAN framework, two models are trained simultaneously. We have a generative model G that is used to estimate the given training dataset distribution and a discriminative model D that estimates the probability that a sample came from the training data instead of G.

Given data set $\mathbf{x}$, we must learn the generator distribution p_g over $\mathbf{x}$. They prior distribution over input noise variables $p(\mathbf{z})$. Given these input noise variables, a parameterized neural network is learned $G_{\theta_g}(\mathbf{z})$, which maps these input noise variables to the data set space. A 2nd parameterized neural network is learned $D_{\theta_D}(\mathbf{x})$, which learns to estimate the probability that $\mathbf{x}$ came from the data set and not the generator p_g.

In the two-player adversarial training procedure, D is trained to maximize the probability of assigning the correct labels to the dataset and the generated samples. G is trained to minimize the objective $\log(1 - D_{\theta_D}(G_{\theta_G}(\mathbf{z})))$, which causes it to generate samples that are more likely to fool the discriminator. The dual objective is

$$\mathbb{E}_{\mathbf{x} \sim p_{\text{data}}}[\log D_{\theta_D}(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})}[\log(1 - D_{\theta_D}(G_{\theta_G}(\mathbf{z})))]. \tag{3.18}$$

At the learned global optimality when $p_g = p_{\text{data}}$, it is shown that the optimal discriminator for G is

$$D^*_{\theta_D}(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})}. \tag{3.19}$$

3.6 Imitation Learning with Deep Learning

It is easy to see how neural networks can be used as function approximators for parameterized value functions or policies. They can also be used to learn parametrized reward functions which makes learning a complex reward function for IRL possible.

It is possible to exploit the adversarial nature of a GAN for IRL by feature expectation matching with deep neural networks. This technique is known as Generative Adversarial Imitation Learning (GAIL) [21].

3.6.1 Generative Adversarial Imitation Learning (GAIL)

GAIL is an imitation learning method that uses the GAN structure [21]. The discriminator takes state-action pair inputs from expert demonstrations and rollouts from a (learned) novice parameterized policy. The discriminator is optimized with binary logistic regression to learn the of the probability that the state-action pair came from the expert or novice policy roll-out. The probability of the state-action pair belonging to the expert demonstration can be used as a reward function for policy optimization. At each step, after estimated rewards are obtained using this method, we optimize the novice parameterized policy with respect to those rewards.

For a discriminator parameterized by θ_D, given sampled novice trajectories $\tau_i \sim \pi_{\theta_i}$ from our learn-able policy, the objective is

$$J(\theta_D) = \mathbb{E}_{\tau_i}[\nabla_{\theta_D} \log(D_{\theta_D}(s,a))] + \mathbb{E}_{\tau^E}[\nabla_{\theta_D} \log(1 - D_{\theta_D}(s,a))]. \tag{3.20}$$

We then update $\pi_{\theta_{\text{policy}}}$ given the rewards from our discriminator probabilities (of each state action pairs) using a policy optimization method. We can see that GAIL does not directly recover rewards. At optimality, the discriminator will output 0.5 for all pairs.

3.6.2 Adversarial Inverse Reinforcement Learning with Robust Rewards

We will now examine AIRL [16], an IRL method that presents concepts instrumental to the main algorithm of this thesis.

Adversarial Inverse Reinforcement Learning (AIRL) is an adversarial IRL method that recovers a reward function. In addition, it shows that these reward functions are robust to changes in dynamics and are shown to be disentangled rewards.

AIRL is based on GAN-Guided Cost Learning [13], which casts the IRL MLE (Equation 2.27) objective as a Generative Adversarial Network (GAN) [18] optimization problem over trajectories. In GAN-GCL, they maximize the likelihood over trajectories in an energy based model (Boltzmann distribution) to form a generator objective. The discriminator in this GAN operates over an entire trajectory and is defined as,

$$D_\theta(\tau) = \frac{\exp(f_\theta(\tau))}{\exp(f_\theta(\tau)) + \pi(\tau)}. \tag{3.21}$$

Here we have a learned function $f_\theta(\tau)$ with $\pi(\tau)$ being precomputed (since we know the output of our discriminator and $f_\theta(\tau)$). The objective to be maximized is $\hat{R}(\tau) = \log(1 - D(\tau)) - \log D(\tau)$. It is also shown that the optimal reward function is found to be $f^*(\tau) = R^*(\tau) + \text{const}$ when the GAN is trained to optimality. The policy π is also shown to be optimal.

In AIRL [16], the discriminator probability D_θ is evaluated using the state-action pairs from the generator (agent), as given by

$$D_\theta(s, a) = \frac{\exp(f_\theta(s, a))}{\exp(f_\theta(s, a)) + \pi(a|s)}, \tag{3.22}$$

where $f_\theta(s, a)$ is again a learned function and $\pi(a|s)$ is pre-computed. This is because full trajectories can result in high variance estimates. It is also shown that at optimality $f^*(s, a) = \log \pi^*(a|s) = A^*(s, a)$. The rewards are therefore recoverable. The agent's

reward function is estimated as,

$$\hat{r}(s,a) = \log(1 - D_\theta(s,a)) - \log(D_\theta(s,a)). \tag{3.23}$$

They then propose learning a state only reward function so rewards are not shaped and will be disentangled. To do this they modify the discriminator function in a manner which operators on state-action-next state triplets. This is defined as,

$$D_{\theta,\Phi}(s,a,s') = \frac{\exp(f_{\theta,\Phi}(s,a,s'))}{\exp(f_{\theta,\Phi}(s,a,s')) + \pi(a|s)}. \tag{3.24}$$

Recall the definition of Reward Ambiguity (2.5.2), with $f_{\theta,\Phi}$ having reward approximator g_θ with shaping function h_Φ, we can define the function as,

$$f_{\theta,\Phi}(s,a,s') = g_\theta(s,a) + \gamma h_\Phi(s') - h_\Phi(s). \tag{3.25}$$

The introduction of this shaping term mitigates unwanted shaping on the reward approximator. They then prove that they can parameterize the reward approximator only as a function of state, $g_\theta(s)$. We detail this proof in a proof sketch. If a reward approximator is only a function of state, it is also a disentangled reward approximator with respect to transition dynamics.

3.6.3 Proof of State Only Rewards

Here we will show that AIRL recovers a reward function that is state only. Our first step will be to show that our discriminator function recovers the advantage function for each (s,a) pair. This is necessary to rearrange terms correctly in our main theorem. We will also give the definition of decomposability which is required to show relationships between arbitrary functions of state and next state pairs.

Lemma 3.6.1 *$f_{\theta,\omega}(s,a)$ recovers the advantage.*

Proof. It is known that when $\pi = \pi_E$, we have achieved the global min of the discriminator objective. The discriminator must then output 0.5 for all state action pairs. This results in $\exp(f_\theta(s,a)) = \pi_E(a|s)$. Equivalently we have $f^*(s,a) = \log \pi_E(a|s) = A^*(s,a)$. ■

Definition 3.6.1 ***Decomposability condition.*** *We first define 2 states s_1, s_2 as 1-step linked under dynamics $T(s'|s,a)$ if there exists a state s that can reach s_1 and s_2 with non-zero probability in one timestep. The transitivity property holds for the linked relationship. We can say that if s_1 and s_2 and linked, s_2 and s_3 are linked then s_1 and s_3 must also be linked.*

The Decomposability condition for transition dynamics T holds if all states in the MDP are linked with all other states.

Lemma 3.6.2 *For an MDP, where the decomposability condition holds for all dynamics. For arbitrary functions $a(s), b(s), c(s), d(s)$, if for all s and s',*

$$a(s) + b(s') = c(s) + d(s'), \tag{3.26}$$

and for all s,

$$a(s) = c(s) + const_s, \tag{3.27}$$

$$b(s) = d(s) + const_s, \tag{3.28}$$

where $const_s$ is a constant dependent with respect to state s.

Proof. If we rearrange Equation 3.26, we can obtain the quality $a(s) - c(s) = b(s') - d(s')$.

Now we define $f(s) = a(s) - c(s)$. Given our equality, we have $f(s) = a(s) - c(s) = b(s') - d(s')$. This holds for some function dependent on s.

To represent this, $b(s') - d(s')$ must be equal to a constant (with the constant's value dependent on the state s) for all one-step successor states s' from s.

Now, under decomposability, all one step successor states (s') from s must be equal through the transitivity property so $b(s') - d(s')$ must be a constant with respect to state s. Therefore, we can write $a(s) = c(s) + \text{const}_+ s$ for an arbitrary state s and functions b and d.

Substituting this into the Equation 3.26, we can obtain $b(s) = d(s) + \text{const}_s$. Now we will show an Inductive proof that this holds for any successor state.

Let us consider for any MDP and any arbitrary functions $a(\cdot)$, $b(\cdot)$, $c(\cdot)$ and $d(\cdot)$,

$$a(s) + b(S^{(k)}) = c(s) + d(S^{(k)}), \tag{3.29}$$

where $S^{(k)}$ is the k-th successor state reached in k time-steps from the current state. Let us denote by $T^{\pi,(k)}(s, S^{(k)})$ the probability of transitioning from state s to $S^{(k)}$ in k steps using policy π. Then, we can express $T^{\pi,(k)}(s, S^{(k)})$ recursively as,

$$T^{\pi,(k)}(s, S^{(k)}) = \sum_{s' \in \mathcal{S}} T^{\pi,(k-1)}(s, s') T^{\pi}(s', S^{(k)}), \tag{3.30}$$

where $T^{\pi}(s', S^{(k)})$ is the one-step transition probability from state s' to state $S^{(k)}$ (by definition of the Bellman operator).

Denote $P(S^{(k)})$ as the probability of landing in state $S^{(k)}$ in k steps from any current state. We can write $P(S^{(k)})$ using (3.30) as,

$$P(S^{(k)}) \sum_{s \in \mathcal{S}} T^{\pi,(k)}(s, S^{(k)}) \mu(s), \tag{3.31}$$

where μ is the state-distribution.

The unbiased estimator $\hat{s}^{(k)}$ of an unknown successor state $S^{(k)}$ is given by

$$\hat{s}^{(k)} \mathbb{E}(S^{(k)}) = \sum_{s^{(k)} \in \mathcal{S}} s^{(k)} P(S^{(k)}), \tag{3.32}$$

where $P(S^{(k)})$ is given in (3.31).

Now, replacing $S^{(k)}$ in (3.29) with its unbiased estimator $\hat{s}^{(k)}$ as given by (3.32), we have

$$a(s) - c(s) = b(\hat{s}^{(k)}) - d(\hat{s}^{(k)}) \overset{(a)}{=} f(k), \tag{3.33}$$

for some function f, where (a) holds since $\hat{s}^{(k)}$ depends only on k. Thus, we get $a(s) = c(s) +$ const. and $b(s) = d(s) +$ const. where the constant is with respect to the state s. ■

Now we have the shown relationship for arbitrary functions applied to a state and next state are related by a constant dependent on the state. We are now going to show that we can apply this relationship to the discriminator function and the reward estimator to show that the reward estimator is state only.

Theorem 3.6.3 *Suppose we have, for a MDP where the decomposability condition holds, then*

$$f_\theta(s, a, s') = g(s, a) + \gamma h_\Phi(s') - h_\Phi(s), \tag{3.34}$$

where h_Φ is a shaping term from our definition of reward ambiguity. If we obtain the optimal $f^*_\theta(s, a, s')$, with a reward approximator $g^*(s, a)$. Under deterministic dynamics the following holds

$$g^*(s, a) + \gamma h^*_\Phi(s') - h^*_\Phi(s) = r^*(s) + \gamma V^*(s') - V^*(s), \tag{3.35}$$

and,

$$g^*(s) = r^*(s) + const_s. \tag{3.36}$$

Proof. We know $f^*(s, a, s') = A^*(s, a) = Q^*(s, a) - V^*(s) = r^*(s) + \gamma V^*(s') - V^*(s)$ from Lemma 3.6.1. We can substitute the definition of $f^*(s, a, s')$ to obtain

$$g^*(s, a) + \gamma h^*\Phi(s') - h^*\Phi(s) = r^*(s) + \gamma V^*(s') - V^*(s), \tag{3.37}$$

which holds for all s and s'. Now we apply Lemma 3.6.2. We say that $a(s) = g^*(s) - h^*(s), b(s') = \gamma h^*(s'), c(s) = r(s) - V^*(s)$ and $d(s') = \gamma V^*(s')$ and rearrange according to Lemma 3.6.2. We therefore have our results that $g^*(s) = r(s) +$ const and $h^*(s) = V^*(s) +$ const.

Since the reward approximator is state only, the rewards are invariant to environmental transitional dynamics. It is therefore shown in AIRL that there is a state-only reward

approximator $f^*(s, a, s') = r^*(s) + \gamma V^*(s') - V^*(s) = A^*(s, a)$ where the reward is invariant to transition dynamics and is disentangled. ■

It is important to show that the rewards are disentangled to solve a certain set of problems. These problems arise when we learn rewards in one environment and then wish to exploit these rewards in a similar environment with similar goals but different transition dynamics.

It is experimentally shown in AIRL that the rewards learned are highly generalizable and therefore policy optimization is more portable to being used in new environments with similar underlying goals. This is an important problem we wish to solve that will be discussed in greater detail in Chapter 5.

Chapter 4

Robust Inverse Reinforcement Learning with Options

In this chapter, we detail the main method of this thesis. We wish to formulate an IRL method that learns a policy over options with state-only disentangled rewards. As we have stated, state only rewards are disentangled and therefore highly generalizable. A policy over options is also shown to create more generalizable policies. The goal of this method is therefore to learn a state-of-the-art generalizable reward function.

The main steps will include showing an MLE objective for Maximum Entropy hierarchical inverse reinforcement learning (HIRL) with options. We will then define the objective and optimization criteria for the discriminator in GAN (GAN-GCL) based HIRL and show that this is equal to the MLE objective. This shows that our algorithm is equivalent to solving the Maximum Entropy HIRL problem. Finally, we will unify these two objectives in a manner that is disentangled with respect to environmental dynamics.

4.1 Preliminaries

We will assume we have an adversarial IRL framework with a generator that generates trajectories from a policy over options and a discriminator for each option which takes state-action pairs as input.

Let $(s_0, a_0, \ldots s_T, a_T) \in \tau_i^E$ be an expert trajectory of state-action pairs. Define a novice trajectory as $(s_0, a_0, \omega_0 \ldots s_T, a_T, \omega_T) \in \tau_{\pi_{\Theta,t}}$ generated by policy over options $\pi_{\Theta,t}$ of the generator at iteration t.

We also commonly refer to the transition probabilities for a policy over options. We refer to the same formulation as in Subsection 2.6.1.

The policy over options is parameterized by ζ, the intra-option policies by α for each option, the reward approximator by θ, and the option termination probabilities by δ.

4.2 Overview of Derivation Steps

First, we will formulate an MLE objective for inverse reinforcement learning with a policy over options. This will be later used to show that a GAN based objective (based on AIRL) is equivalent to this MLE objective. We will also show the derivative of this objective for later use in optimizing the objective with Gradient Descent.

We will then formulate a discriminator objective or our IRL problem based on a GAN. We will show that this objective is the same as the MLE objective and then give the derivative of it.

Finally, we will show that our discriminator objective recovers state only rewards for each option's reward function and is, therefore, invariant to transition dynamics. Since we have a GAN based objective we can prove this in a way similar to AIRL. This will be followed by the main algorithm that implements this discriminator and an analysis of its' convergence.

4.3 MLE Objective for IRL Over Options

In this section, we will define the MLE objective for our IRL problem in a policy over options framework. With ω_0 selected according to $\pi_\Omega(\omega|s)$, we can define a maximum causal entropy RL problem with objective $\max_\theta \mathbb{E}_{\tau\sim\mathcal{D}}[\log(p_\theta(\tau))]$ where,

$$p_\theta(\tau) \sim p(s_0, \omega_0) \prod_{t=0}^{T-1} P(s_{t+1}, \omega_{t+1}|s_t, \omega_t, a_t) e^{\hat{r}_{\theta,\omega}(s_t,a_t)}. \tag{4.1}$$

4.3.1 Demonstrator Option Inference

We do not know the option trajectories in our expert demonstrations, so they must be inferred. This is a separate step that we must consider, which we will detail next.

Given the expert trajectories $\tau\{(s_t, a_t)\}_{t=0}^T$, the agent needs to infer the option sequence $\{\omega_t\}_{t=0}^T$ for each (s_t, a_t) to generate the augmented trajectories $\tau_\omega\{s_t, \omega_t, a_t)\}_{t=0}^T$. Also, for ease of exposition, let us denote by τ^{AE} the set of augmented expert trajectories. In order to solve the inference problem, we start with a parameterised policy over options, π_Ω and parameterised intra-option policies and termination functions corresponding to an option ω_t, π_{ω_t} and β_{ω_t}. The posterior $P(\omega_t|s_{0:t}, a_{0:t})$ is given by the Bayesian update,

$$P(\omega_t|s_{0:t}, a_{0:t}) \propto \pi_{\omega_t}(a_t|s_t) P(\omega_t|s_{0:t}, a_{0:t-1}). \tag{4.2}$$

The prediction $P(\omega_{t+1}|s_{0:t}, a_{0:t})$ can be computed as,

$$P(\omega_{t+1}|s_{0:t}, a_{0:t}) = \sum_{\omega_t\in\Omega} P(\omega_t|s_{0:t}, a_{0:t}) \tilde{\pi}_\Omega(\omega_{t+1}|\omega_t, s_{t+1}),$$

where $\tilde{\pi}_\Omega(\omega_{t+1}|\omega_t, s_{t+1})$ is given by,

$$\tilde{\pi}_\Omega(\omega_{t+1}|\omega_t, s_{t+1}) = (1 - \beta(s_{t+1}))\delta_{\omega_{t+1},\omega_t} + \beta(s_{t+1})\pi_\Omega(\omega_{t+1}|s_{t+1}). \tag{4.3}$$

Here δ is the Kronecker delta, $\delta_{\omega_{t+1},\omega_t} = 1$ when $\omega_{t+1} = \omega_t$ and $\delta_{\omega_{t+1},\omega_t} = 0$ otherwise.

Now, the MAP (Maximum a Posteriori) estimator of (4.2) is computed as,

$$\log \hat{P}(\omega_t|s_{0:t}, a_{0:t}) \leftarrow \max \log \pi_{\omega_t}(a_t|s_t) + \log \hat{P}(\omega_t|s_{0:t}, a_{0:t-1}), \tag{4.4}$$

where the maximization is done with respect to the parameterized policies in (4.4). Once the augmented trajectories τ_ω are generated, for each given option ω, a *discriminator* D_ω is trained using τ_ω and agent trajectories generated by rolling out the agent policies π_{ω_t}. The generator density function with τ_ω is given as,

$$p_\theta(\tau_\omega) \propto p(s_0, \omega_0) \prod_{t=0}^{T} \hat{P}(\omega_t|s_{0:t}, a_{0:t}) \pi_{\omega_t}(a_t|s_t) e^{\hat{r}_{\theta,\omega_t}(s_t,a_t)}, \tag{4.5}$$

where $\hat{P}(\omega_t|s_{0:t}, a_{0:t-1})$ is the inferred probability obtained by solving (4.4). The inference algorithm is detailed in Algorithm 5.

he actual demonstrations from the expert trajectories. In particular, at any instant t, the cross-entropy for training is

Algorithm 5 Option Inference Estimation

Input: Expert Trajectories: $\{\tau_1^E, \ldots, \tau_n^E\} \in \mathcal{T}_\mathcal{D}$, Policy Parameters:$(\theta, \zeta, \delta, \alpha), \gamma$

for Expert trajectory $\tau_i \in \{\tau_1^E, \ldots, \tau_n^E\}$ **do**
  $\{s_0, a_0, s_1\} \sim \tau_i$
  $\hat{P}(\omega_0|s_{0:0}, a_{0:0}) = \pi_{\Omega,\zeta}(\omega|s_0)$
  $\omega_0 \sim \pi_{\Omega,\zeta}(\omega|s_0)$
  $\tau_i \cup \{\omega_0\}$
  $\hat{\pi}_\Omega(\omega_1|\omega_0, s_1) = (1 - \beta(s_1))\delta_{\omega_1,\omega_0} + \beta(s_1)\pi_{\Omega,\zeta}(\omega_1|s_1)$
  $\hat{P}(\omega_1|s_{0:1}, a_{0:0}) = \sum_{\omega_0 \in \Omega} P(\omega_0|s_{0:0}, a_{0:0})\pi_\Omega(\omega_1|\omega_0, s_1)$
  for step $t = 0, 1, \ldots, T$ **do**
    $\log \hat{P}(\omega_t|s_{0:t}, a_{0:t}) \leftarrow \max \log \pi_{\omega_t}(a_t|s_t) + \log \hat{P}(\omega_t|s_{0:t}, a_{0:t-1})$
    $\hat{\pi}_\Omega(\omega_t|\omega_{t-1}, s_t) = (1 - \beta(s_t))\delta_{\omega_t,\omega_{t-1}} + \beta(s_t)\pi_{\Omega,\zeta}(\omega_t|s_t)$
    $\hat{P}(\omega_t|s_{0:t}, a_{0:t-1}) = \sum_{\omega_{t-1} \in \Omega} P(\omega_{t-1}|s_{0:t}, a_{0:t-1})\pi_\Omega(\omega_t|\omega_{t-1}, s_t)$
    $\omega_t \sim \hat{P}(\omega_t|s_{0:t}, a_{0:t})$
    $\tau_i \cup \{\omega_t\}$
  end for
end for

4.3.2 MLE Objective

From (4.5), we can write the MLE objective for the agent, which is to be maximized (up to a normalizing constant) as,

$$J(\theta) = \mathbb{E}_{\tau_\omega}[\log p_\theta(\tau_\omega)] = \mathbb{E}_\tau \Big[\log p(s_0, \omega_0) + \sum_{t=0}^{T} [\hat{r}_{\theta,\omega_t}(s_t, a_t) + \log \hat{P}(\omega_t | s_{0:t}, a_{0:t-1}) + \log \pi_{\omega_t}(a_t | s_t)] \Big]. \tag{4.6}$$

This is given by expanding the log term. Note that the first term on RHS of (4.6) does not participate in the maximization since it is assumed a-priori and thus not parameterized.

Now we will define a loss in terms of the expected returns. Let us denote by $G_t^{\tau_\omega}$ the discounted return from trajectory τ_ω as,

$$G_{t,\theta}^{\tau_\omega} = \sum_{t'=t}^{T} \gamma^{t'-t} r_{\theta,\omega_{t'}}(s_{t'}, a_{t'}). \tag{4.7}$$

The un-discounted return is

$$R_{t,\theta}^{\tau_\omega} = \sum_{t'=t}^{T} r_{\theta,\omega_{t'}}(s_{t'}, a_{t'}). \tag{4.8}$$

Then a loss similar to (4.6) can be rewritten in terms of $R_t^{\tau_\omega}$ as,

$$J(\theta) = \mathbb{E}_{\tau_\omega}[R_{0,\theta}^{\tau_\omega}] = \int_{\tau_\omega} R_{0,\theta}^{\tau_\omega} p_\theta(\tau_\omega) d\tau_\omega. \tag{4.9}$$

given the expected value of a continuous probability distribution.

In the next steps, we will define a loss function function from (4.6) under deterministic dynamics. For ease of derivations we let

$$\mathbb{T}_\theta(0, \tau_\omega) = R_{0,\theta}^{\tau_\omega} + \sum_{t=0}^{T} \log[\hat{P}(\omega_t | s_{0:t}, a_{0:t-1}) \pi_{\omega_t}(a_t | s_t)], \tag{4.10}$$

and,

$$\nabla_\theta \mathbb{T}_\theta(0, \tau_\omega) = \nabla_\theta R^{\tau_\omega}_{0,\theta}. \tag{4.11}$$

We can write out our MLE objective for our generator ($J(\theta)$). This is defined similarly in [16] and [13]. The full derivation is shown (with generator p_θ) as,

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\tau_\omega \sim \tau^{AE}} \Big[\log(p_\theta(\tau)) \Big] \\ &= \mathbb{E}_{\tau_\omega \sim \tau^{AE}} \Big[\mathbb{T}_\theta(0, \tau_\omega) \Big] - Z_\theta \\ &\approx \mathbb{E}_{\tau_\omega \sim \tau^{AE}} \Big[\mathbb{T}_\theta(0, \tau_\omega) \Big] - \mathbb{E}_{p_\theta} \Big[\mathbb{T}_\theta(0, \tau_\omega) \Big], \end{aligned} \tag{4.12}$$

where Z_θ is a partition function in the formulation of $p(\theta)$ from Equation (2.28).

4.3.3 MLE Derivative

Now that we have an MLE objective, we take the gradient of it with respect to θ. This yields,

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_{\tau_\omega \sim \tau^{AE}} \Big[\nabla_\theta \log(p_\theta(\tau)) \Big] \\ &= \mathbb{E}_{\tau_\omega \sim \tau^{AE}} \Big[\nabla_\theta \mathbb{T}_\theta(0, \tau_\omega) \Big] - \nabla_\theta Z_\theta \\ &\approx \mathbb{E}_{\tau_\omega \sim \tau^{AE}} \Big[\nabla_\theta R^{\tau_\omega}_{0,\theta} \Big] - \mathbb{E}_{p_\theta} \Big[\nabla_\theta R^{\tau_\omega}_{0,\theta} \Big]. \end{aligned} \tag{4.13}$$

We now define $p_{\theta,t}(s_t, a_t) = \int_{s_{t' \neq t}, a_{t' \neq t}} p_\theta(\tau) ds_{t'} da_{t'}$ as the state action marginal at time t. This allows us to examine the trajectory from step t. Consequently, we have

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau_\omega \sim \tau^{AE}} \Big[\nabla_\theta R^{\tau_\omega}_{0,\theta} \Big] - \mathbb{E}_{p_{\theta,t}} \Big[\nabla_\theta R^{\tau_\omega}_{0,\theta} \Big]. \tag{4.14}$$

We perform importance sampling over the hard to estimate generator density. We make an importance sampling distribution $\mu_{t,w}(\tau)$ for option w. We sample a mixture policy $\mu_\omega(a|s)$ defined as $\frac{1}{2}\pi_\omega(a|s) + \frac{1}{2}\hat{p}_\omega(a|s)$ and $\hat{p}_\omega(a|s)$ is a rough density estimate trained

on the demonstrations. We wish to minimize the $D_{KL}(\pi_w(\tau)||p_\omega(\tau))$. KL refers to the Kullback–Leibler divergence metric between two probability distributions. Applying the aforementioned density estimates we can express the gradient of the MLE objective J as,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau_\omega \sim \tau^{AE}}[\nabla_\theta R_{0,\theta}^{\tau_\omega}] - \mathbb{E}_{\mu_t}\left[\frac{p_{\theta,t,\omega}(s_t, a_t)}{\mu_{t,w}(s_t, a_t)} \nabla_\theta R_{0,\theta}^{\tau_\omega}\right]. \tag{4.15}$$

In our policy optimization objective for a single option policy, we take the derivative of $J(\alpha)$ with respect to α. Note we are writing a policy gradient in terms of the discounted return for example (this is not necessarily used in the final algorithm). It is obtained from (4.9) as,

$$\nabla_\theta J(\alpha) = \mathbb{E}_{\tau_\omega}\left[G_{0,\theta}^{\tau_\omega} \sum_{t=0}^{T} \nabla_\alpha \log[\hat{P}(\omega_t|s_{0:t}, a_{0:t-1})\pi_{\omega_t,\alpha}(a_t|s_t)]\right], \tag{4.16}$$

where $G_{0,\theta}^{\tau_\omega}$ is found using our IRL objective for the reward function.

4.4 Discriminator Objective

We formulate the discriminator as the odds ratio between the policy and the exponentiated reward distribution for option ω as in AIRL parameterized by θ. We have a discriminator for each option ω and generator option policy π_ω,

$$D_{\theta,\omega}(s, a) = \frac{\exp(f_{\theta,\omega}(s, a))}{\exp(f_{\theta,\omega}(s, a)) + \pi_w(a|s)}. \tag{4.17}$$

4.4.1 Loss Formulation

For a given option ω, we define the estimated return function $R_{0,\theta}^{\tau_\omega}$, which is to be maximised. We can we let discriminator cost function $f_{\theta,\omega}(s, a)$ act as the state-action reward

function for option ω. We can then think of our estimated return as,

$$\hat{R}^{\tau_\omega}_{t,\theta} = \sum_{t'=t}^{T} f_{\theta,\omega_{t'}}(s_{t'}, a_{t'}). \tag{4.18}$$

We minimize the cross-entropy loss between expert demonstrations and generated examples assuming we have the same number of options in the generated and expert trajectories. We define the discriminator loss function L^D as,

$$L^D = -\mathbb{E}_{\tau\sim\tau^{AE}}[\log(D_{\theta,\omega_t}(s_t, a_t))] - \mathbb{E}_{\pi_{\Theta,t}}[\log(1 - D_{\theta,\omega_t}(s_t, a_t))]. \tag{4.19}$$

The agent wishes to minimize L^D to find its optimal policy. We now write a negative discriminator loss ($-L^D$) to turn our loss minimization problem into a maximization problem. This loss is defined as

$$-L^D = \mathbb{E}_{\tau\sim\tau^{AE}}[\log(D_{\theta,\omega_t}(s_t, a_t))] + \mathbb{E}_{\pi_{\Theta,t}}[\log(1 - D_{\theta,\omega_t}(s_t, a_t))]. \tag{4.20}$$

Since we have set up a maximization problem, we can let cost function inside our discriminator be $\hat{R}^{\tau_\omega}_{t,\theta}$ and we have a discriminator in the form,

$$D_{\theta,\omega}(s_t, a_t) = \frac{\exp(\sum_{t'=t}^{T} f_{\theta,\omega_{t'}}(s_t, a_t))}{\exp(\sum_{t'=t}^{T} f_{\theta,\omega_{t'}}(s_t, a_t)) + \pi_\omega(a_t|s_t)} = \frac{\exp(\hat{R}^{\tau_\omega}_{t,\theta})}{\exp(\hat{R}^{\tau_\omega}_{t,\theta}) + \pi_\omega(a_t|s_t)}. \tag{4.21}$$

4.4.2 Optimization Criteria

Note that θ parameterizes the state-action reward function estimate for option ω. The term $-L^D$ is the negative discriminator loss. We therefore turn our minimization problem into a maximization problem. We define our objective similar to the GAN objective from AIRL.

For a given option ω, we can write the reward (option return) estimator function $\hat{r}_{\omega,\theta}(s,a)$ to be used in our policy optimization step as,

$$\begin{aligned}\hat{r}_{\omega,\theta}(s_t,a_t) &= \log\left(D_{\theta,\omega}(s_t,a_t)\right) - \log\left(1-D_{\theta,\omega}(s_t,a_t)\right). \\ &= \log\left(\frac{\exp\left(\sum_{t'=t}^{T} f_{\theta,\omega_{t'}}(s_t,a_t)\right)}{\exp\left(\sum_{t'=t}^{T} f_{\theta,\omega_{t'}}(s_t,a_t)\right)+\pi_\omega(a_t|s_t)}\right) - \\ &\log\left(\frac{\pi_\omega(a_t|s_t)}{\exp\left(\sum_{t'=t}^{T} f_{\theta,\omega_{t'}}(s_t,a_t)\right)+\pi_\omega(a_t|s_t)}\right) \\ &= \hat{R}^{\tau_\omega}_{t,\theta} - \log(\pi_\omega(a_t|s_t)).\end{aligned} \tag{4.22}$$

With $\sum_{t'=t}^{T} f_{\theta,\omega_{t'}}(s_t,a_t)$ acting as the cost function, the discriminator's objective and can be defined as,

$$\begin{aligned}-L^D &= \mathbb{E}_{\tau\sim\tau^{AE}}\left(\left[\log(D_{\theta,\omega}(s_t,a_t))\right] + \mathbb{E}_{\pi_t}\left[\log(1-D_{\theta,\omega}(s_t,a_t))\right]\right) \\ &= \mathbb{E}_{\tau\sim\tau^{AE}}\left[\log\left(\frac{\exp(\hat{R}^{\tau_\omega}_{t,\theta})}{\exp(\hat{R}^{\tau_\omega}_{t,\theta})+\pi_\omega(a_t|s_t)}\right)\right] \\ &\quad + \mathbb{E}_{\pi_t}\left[\log\left(\frac{\pi_\omega(a_t|s_t)}{\exp(\hat{R}^{\tau_\omega}_{t,\theta})+\pi_\omega(a_t|s_t)}\right)\right] \\ &= \mathbb{E}_{\tau\sim\tau^{AE}}\left[\hat{R}^{\tau_\omega}_{t,\theta}\right] - \mathbb{E}_{\tau\sim\tau^{AE}}\left[\log(\exp(\hat{R}^{\tau_\omega}_{t,\theta})+\pi_\omega(a|s_t))\right] \\ &\quad + \mathbb{E}_{\pi_t}\left[\log(\pi_\omega(a_t|s_t))\right] - \mathbb{E}_{\pi_t}\left[\log(\exp(\hat{R}^{\tau_\omega}_{t,\theta}))+\pi_\omega(a_t|s_t))\right].\end{aligned} \tag{4.23}$$

Now, we set a mixture of experts and novice as $\bar{\mu}$ observations to obtain

$$-L^D = \mathbb{E}_{\tau\sim\tau^{AE}}\left[\hat{R}^{\tau_\omega}_{t,\theta}\right] + \mathbb{E}_{\pi_t}\left[\log(\pi_\omega(a_t|s_t))\right] - 2\,\mathbb{E}_{\bar{\mu}_t}\left[\log\left(\exp(\hat{R}^{\tau_\omega}_{t,\theta})+\pi_\omega(a_t|s_t)\right)\right]. \tag{4.24}$$

We can take the derivative with respect to θ (state-action reward function estimate parameter). The derivation is

$$\nabla_\theta(-L^D) = \mathbb{E}_{\tau\sim\tau^{AE}}\left[\nabla_\theta\hat{R}^{\tau_\omega}_{t,\theta}\right] - \mathbb{E}_{\bar{\mu}_t}\left[\nabla_\theta\left(\frac{\exp(\hat{R}^{\tau_\omega}_{t,\theta})}{\frac{1}{2}\exp(\hat{R}^{\tau_\omega}_{t,\theta})+\frac{1}{2}\pi_\omega(a_t|s_t)}\right)\hat{R}^{\tau_\omega}_{t,\theta}\right]. \tag{4.25}$$

We can multiply the top and bottom of the fraction in the mixture expectation by the state marginal $\pi_\omega(s_t) = \int_{a \in A} \pi_\omega(s_t, a_t)$. This allows us to write $\hat{p}_{\theta,t,\omega}(s_t, a_t) = \exp(L(s_t, \omega_t, a_t))\pi_{\omega,t}(s_t)$. Using this, we can derive an importance sampling distribution in our loss,

$$\nabla_\theta(-L^D) = \mathbb{E}_{\tau \sim \tau^{AE}}\left[\nabla_\theta \hat{R}^{\tau_\omega}_{t,\theta}\right] - \mathbb{E}_{\bar{\mu}_t}\left[\left(\frac{\hat{p}_{\theta,t,\omega}(s_t, a_t)}{\hat{\mu}_{t,\omega}(s_t, a_t)}\right)\nabla_\theta \hat{R}^{\tau_\omega}_{t,\theta}\right]. \tag{4.26}$$

Now we have shown that our discriminator loss (Equation 4.26) is of the same form as our MLE loss (Equation 4.15).

4.5 Learning Disentangled State-only Rewards with Options

In this section, we provide our main algorithm for learning robust rewards with options. We implement this algorithm similarly to AIRL, with a discriminator update that considers the rollouts of a policy over options. We perform this update with (s, a, s') triplets and a discriminator function in the form of $f_{\theta,\omega}(s, a, s')$. This allows us to formulate the discriminator with state-only rewards in terms of option-value function estimates to compute an option-advantage estimate. Since the reward function only requires state, we learn a reward function and corresponding policy that is disentangled from the environmental transition dynamics. The discriminator function is

$$f_{\omega,\theta}(s, a, s') = \hat{r}_{\omega,\theta}(s) + \gamma \hat{V}_\Omega(s') - \hat{V}_\Omega(s) = \hat{A}(s, a, \omega), \tag{4.27}$$

where,

$$Q(s, \omega) = \sum_{a \in \mathcal{A}} \pi_{\omega,\alpha}(a|s)\Big[r_{\omega,\theta}(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a)\Big((1 - \beta_{\delta,\omega}(s'))Q(s', \omega) + \beta_{\delta,\omega}(s')V_\Omega(s')\Big)\Big], \tag{4.28}$$

and $V_\Omega(s) = \sum_{\omega \in \Omega} \pi_{\Omega,\zeta}(\omega|s)Q(s, \omega)$.

We now show that the reward approximator is state only for a policy over options in Theorem 4.5.1. This is important because a state only reward aproximator is disentangled with respect to transition dynamics of the environment.

Theorem 4.5.1 *Suppose we have, for a MDP where the decomposability condition (Definition 3.6.1) holds,*

$$f_{\theta,\omega}(s,a,s') = g_\omega(s,a) + \gamma h_\Phi(s') - h_\Phi(s), \tag{4.29}$$

where h_Φ is a shaping term. If we obtain the optimal $f^*_{\theta,\omega}(s,a,s')$, with a reward approximator $g^*_\omega(s,a)$. Under deterministic dynamics the following holds

$$g^*_\omega(s,a) + \gamma h^*_\Phi(s') - h^*_\Phi(s) = r^*_\omega(s) + \gamma V^*_\Omega(s') - V^*_\Omega(s), \tag{4.30}$$

and,

$$g^*_\omega(s) = r^*_\omega(s) + c_\omega. \tag{4.31}$$

Proof. We know $f^*_\omega(s,a,s') = A^*(s,a,\omega) = Q^*(s,a,\omega) - V^*_\Omega(s) = r^*_\omega(s) + \gamma V^*_\Omega(s') - V^*_\Omega(s)$ from Lemma 3.6.1. We can substitute the definition of $f^*_\omega(s,a,s')$ to obtain our Theorem.

Now we apply Lemma 3.6.2. We say that $a(s) = g^*_\omega(s) - h^*_\Phi(s), b(s') = \gamma h^*_\Phi(s'), c(s) = r(s) - V^*_\Omega(s)$ and $d(s') = \gamma V^*_\Omega(s')$ and rearrange according to Lemma 3.6.2. We therefore have our results that $g^*_\omega(s) = r_\omega(s) + c_\omega$, where c_ω is a constant. ■

Our discriminator model must learn a parameterization of the reward function and the value function for each option, given the total loss function in (4.36). These parameterized models are learned with a multi-layer perceptron. For each option, the termination functions $\beta_{\omega,\delta}$ and option-policies $\pi_{\omega,\alpha}$ are learned using PPOC (a method which performs PPO with a policy over options) [22]. In addition, we have shown that the discriminator objective is defined as the MLE objective for our IRL problem.

4.5.1 Algorithm

Our main algorithm is given by Algorithm 6. In this algorithm, we randomly initialize all our initial parameters for our discriminator network, policy, and termination functions.

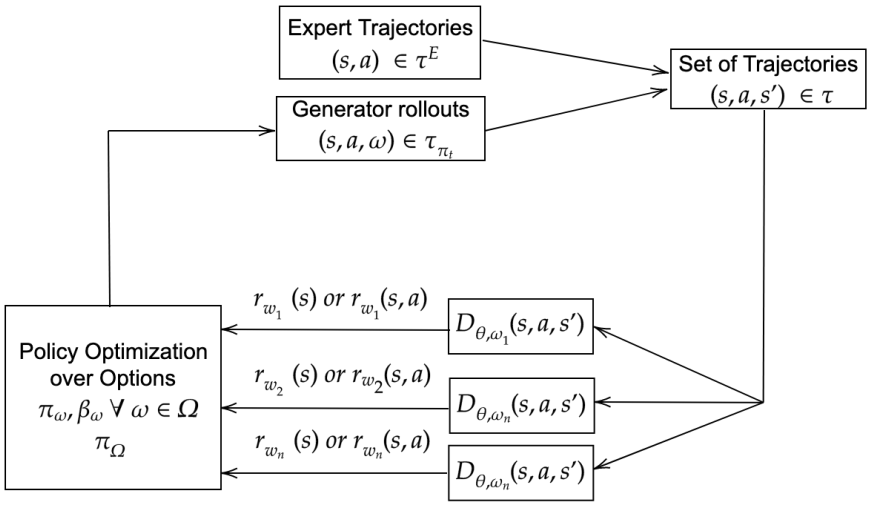


Figure 4.1: Our GAN architecture.

Algorithm 6 IRL Over Options with Robust Rewards (oIRL)

Input: Expert Trajectories: $\{\tau_1^E, \dots, \tau_n^E\} \in \mathcal{T}_\mathcal{D}$, Initial Parameters:$(\theta_0, \zeta_0, \delta_0, \alpha_0), \gamma$

Initialize policies $\pi_{\omega,\alpha_0}, \pi_{\Omega,\zeta_0}$ and discriminators $D_{\theta_0,\omega},$ and $\beta_{\omega,\delta_0} \forall \omega \in \Omega$
for step $t = 0, 1, 2, \dots, T$ **do**
  Learn $\hat{P}(\omega_{k+1}|s_{0:k}, a_{0:k-1})$ according to Algorithm 5 and build τ^{AE}
  Collect trajectories $\tau_i = (s_0, a_0, \omega_0, \dots)$ from $\pi_{\omega,\alpha_t}, \pi_{\Omega,\zeta_t}, \beta_{\omega,\delta_t}$
  Train discriminator $D_{\theta_t,\omega} \forall \omega \in \Omega$
  for step $k = 0, 1, 2, \dots$ **do**
    Sample Any $(s_j, a_j, s_{j+1}, \omega_j) \sim \tau_{i,t}$ s.t. $\omega_j = \omega_k$
    if s'_k not terminal state **then**
      $l_k = -\mathbb{E}_\mathcal{D}[\log(D_{\theta_t,\omega_k}(s_k, a_k, s_{k+1}))] - \mathbb{E}_{\pi_{\Theta,t}}[\log(1 - D_{\theta_t,\omega_k}(s_j, a_j, s_{j+1}))]$
      Optimize model parameters w.r.t.: $L_k = -l_k$
    end if
  end for
  Obtain reward $r_{\theta_t,\omega}(s, a, s') \leftarrow \log(D_{\theta_t,\omega}(s, a, s')) - \log(1 - D_{\theta_t,\omega}(s, a, s'))$
  Update $\pi_{\omega,\alpha_t}, \beta_{\omega,\delta_t} \forall \omega \in \Omega$ and π_{Ω,ζ_t} with any policy optimization method (e.g. PPOC)
end for

We show the adversarial architecture of this algorithm in Figure 4.1.

4.6 Convergence Analysis

In this section, we show an analysis of the convergence of our method.

We have shown in Theorem 4.5.1 that the actual reward function is recovered (up to a constant) by the reward estimators for each option. We will now show that the reward estimator error in an IRL setting is bounded. This will be later used in an analysis of convergence or our IRL algorithm with options.

Definition 4.6.1 ***Reward Approximator Error.*** *From Theorem 4.5.1, we can see that our reward approximator is $g^*_\omega(s) = r_\omega(s) + c_\omega$. We define a reward approximator error over all options as $\delta_r = \sum_{\omega\in\Omega} \pi_\Omega(\omega)|g^*_\omega(s) - r^*(s)|$. This error is bounded by*

$$\delta_r = \sum_{\omega\in\Omega} \pi_\Omega(\omega)|g^*_\omega(s) - r^*(s)| \leq \max_{\omega\in\Omega} c_\omega, \tag{4.32}$$

by definition of $g^*_\omega(s)$.

Using the fact that $g_{\theta,\omega}(s) \to g^*_\omega(s) = r^*(s) + c_\omega$, and by using Cauchy-Schwarz inequality of sup-norm, we prove that the update of the TD-error is a contraction. This is necessary for us to later show that we have convergence of intra-option Q learning in the IRL problem.

Lemma 4.6.1 *The Bellman operator for options in the IRL problem is a contraction.*

Proof. We prove this by Cauchy-Schwarz and the definition of the sup-norm. We must define this inequality in terms of the IRL problem where we have a reward estimator $\hat{g}_{\theta\omega}(s)$

under our learned parameter θ and an optimal reward estimator $r^*(s)$. The derivation is

$$
\begin{aligned}
&||Q_{\pi_{\Omega,t}}(s,\omega) - Q^*(s,\omega)||_\infty \\
&= ||\hat{g}_\theta(s) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a)((1-\beta(s'))Q_{\pi_{\Omega,t}}(s',\omega) + \beta(s') \max_{\omega \in \Omega} Q_{\pi_{\Omega,t}}(s',\omega)) - \\
&r^*(s) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a)((1-\beta(s')Q^*(s',\omega) + \beta(s') \max_{\omega \in \Omega} Q^*(s',\omega))||_\infty \\
&= ||\hat{g}_\theta(s) - r^*(s) + \sum_{s' \in \mathcal{S}} P(s'|s,a)[(1-\beta(s'))(Q_{\pi_{\Omega,t}}(s',\omega) - Q^*(s',\omega))] + \\
&[\beta(s')(\max_{\omega \in \Omega} Q_{\pi_{\Omega,t}}(s',\omega) - \max_{\omega \in \Omega} Q^*(s',\omega))]||_\infty \qquad (4.33) \\
&= ||\sum_{s' \in \mathcal{S}} P(s'|s,a)[(1-\beta(s'))(Q_{\pi_{\Omega,t}}(s',\omega) - Q^*(s',\omega))] + \\
&[\beta(s')(\max_{\omega \in \Omega} Q_{\pi_{\Omega,t}}(s',\omega) - \max_{\omega \in \Omega} Q^*(s',\omega))]||_\infty + \max_{\omega \in \Omega} c_\omega \\
&\leq \sum_{s' \in \mathcal{S}} P(s'|s,a) \max_{s'',\omega''} ||Q_{\pi_{\Omega,t}}(s'',\omega'') - Q^*(s'',\omega'')||_\infty + \max_{\omega \in \Omega} c_\omega \\
&\leq \gamma \max_{s'',\omega''} ||Q_{\pi_{\Omega,t}}(s'',\omega'') - Q^*(s'',\omega'')||_\infty + \max_{\omega \in \Omega} c_\omega.
\end{aligned}
$$

This is given by Definition 4.6.1 and Sutton, Precup, and Singh [35] [Theorem 3].

This gives our result $\max_{s'',\omega''} |Q_{\pi_{\Omega,t}}(s,\omega) - Q^*(s,\omega)| \leq \epsilon + \max_{\omega \in \Omega} c_\omega$ for $\epsilon \in \mathbb{R}_{>0}$. ■

In order to prove asymptotic convergence to the optimal option-value Q^*, we show using the contraction argument that $g_{\theta,\omega}(s) + \gamma Q(s',\omega)$ converges to Q^*.

Theorem 4.6.2 *$g_\theta(s) + \gamma Q(s',\omega)$ converges to Q^*.*

Proof. We know $g_\theta(s) \to g^*_\theta(s) = r^*(s)+$const. Given this we can show by Cauchy-Schwarz that

$$
\begin{aligned}
&|\mathbb{E}[g_\theta(s)] + \gamma\mathbb{E}[Q(s',\omega)|s] - Q^*(s',\omega)| \\
&= |\mathbb{E}[g_\theta(s)] + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)((1-\beta_\omega(s'))Q(s'\omega) + \beta_\omega(s')V_\Omega(s')) \\
&- r^*(s) - \sum_{s'\in\mathcal{S}} P(s'|s,a)((1-\beta_\omega(s')Q^*(s',\omega)) + \beta_\omega(s') \max_{\omega\in\Omega} Q^*(s',\omega)| \\
&= |\mathbb{E}[g_\theta(s)] - r^*(s) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)[\beta_\omega(s')[\max_{\omega\in\Omega} Q(s'\omega) - \max_{\omega\in\Omega} Q^*(s',\omega)] \\
&+ (1-\beta_\omega(s'))[Q(s'\omega) - Q^*(s',\omega)]]| \\
&\overset{(a)}{\leq} (\max_{\omega\in\Omega} c_\omega)|\gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)[\max_{s'',\omega''}||Q(s'',\omega'') - Q^*(s'',\omega'')||]| \\
&\overset{(b)}{\leq} (\max_{\omega\in\Omega} c_\omega)(\epsilon + \max_{\omega\in\Omega} c_\omega)\gamma \sum_{s'\in\mathcal{S}} P(s'|s,a) \\
&\leq (\max_{\omega\in\Omega} c_\omega)(\epsilon + \max_{\omega\in\Omega} c_\omega)\gamma, \qquad (4.34)
\end{aligned}
$$

where (a) follows from Lemma 4.6.1 and (b) holds since $\sum_{s'\in\mathcal{S}} P(s'|s,a) \leq 1$. ■

4.7 Related Work

Hierarchical Inverse Reinforcement Learning methods learn policies with high level temporally extended actions using IRL. OptionGAN [20] provides an adversarial IRL objective function for the discriminator with a policy over options. It is formulated such that L_{reg} defines the regularization terms on the mixture of experts so that they converge to options. The discriminator objective in OptionGAN takes state-only input and is formulated as,

$$L_\Omega = \mathbb{E}_\omega[\pi_{\Omega,\zeta}(\omega|s)(L_{\alpha,\omega})] + L_{reg}, \qquad (4.35)$$

where,

$$L_{\alpha,\omega} = \mathbb{E}_{\tau^N}[\log(r_{\theta,\omega}(s))] + \mathbb{E}_{\tau^E}[\log(1 - r_{\theta,\omega}(s))]. \qquad (4.36)$$

In Directed-Info GAIL [30] implements GAIL in a policy over options framework.

Work such as HIRL [24] solves this hierarchical problem of segmenting expert demonstration transitions by analyzing the changes in *local linearity with respect to a kernel function*. It has been suggested that decomposing the reward function is not enough [20]. Other works have learned the latent dimension along with the policy for this task [19,42]. In this formulation, the latent structure is encoded in an unsupervised manner so that the desired latent variable does not need to be provided. This work parallels many hierarchical IRL methods but with recoverable robust rewards.

Chapter 5

Experiments

In this chapter, we demonstrate the effectiveness of the algorithm presented in Chapter 4 in a variety of simulated reinforcement learning tasks.

Our algorithm can learn completely disentangled reward functions for each option. This should present benefits in terms of reward generalizability and *transfer learning*.

Transfer learning can be described as using information learned by solving one problem and then applying it to a different but related problem. In the RL sense, it means taking a reward function trained on one environment and then using this reward function to optimize a policy to solve a similar task in a different previously unseen environment. An example would be learning to walk along a surface on the earth and then abstracting that knowledge to the task of walking on the moon with very different gravity and dynamics.

We run experiments in different environments to address the following questions:

- Does learning a policy over options with the AIRL framework improve reward robustness in transfer learning tasks (where the environmental dynamics are manipulated)?

- Can the policy over options framework match or exceed benchmarks for imitation learning on complex continuous control tasks?

To answer these questions, we compare our model against AIRL (the current state of the art for transfer learning) in a transfer task by learning in an *Ant* environment and modifying the physical structure of the Ant. We also compare our method on various benchmark IRL continuous control tasks. In addition, we compare against OptionGAN to give a baseline hierarchical RL algorithm. We wish to see if learning disentangled rewards for sub-tasks through the options framework is more portable.

We train a policy using each of the baseline methods and our method on these expert demonstrations for 500 time steps on the *gait* environments and 500 time steps on the hierarchical ones. Then we take the trained reward function and use a policy optimized with respect to this reward function on the transfer environments and observe the reward obtained. Such a method of transferring the reward function is called a *transfer task*.

We will then show that our method performs in line with AIRL in continuous control imitation learning benchmark tasks. There are not transfer tasks. The purpose of these experiments is to show that our method is useful not only for transfer but is also able to perform a wide range of forward RL tasks.

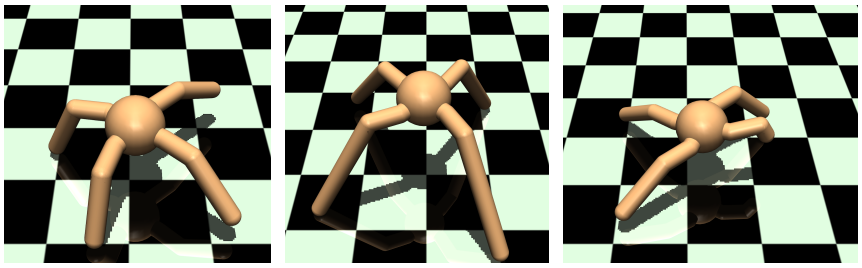
5.1 MuJoCo Experimental Environments

We use the MuJoCo physics simulator [6] for most of the experiments in this work. MuJoCo stands for Multi-Joint dynamics with Contact and has been widely adapted in RL research as a tool to build simulation environments. One of the biggest uses of MuJoCo is to perform robotic simulations using reinforcement learning algorithms as controllers. Tasks such as teaching a robot-like object to walk, hop, and go through a maze are examples of environments that have been created in the engine.

5.1.1 Transfer Learning MuJoCo Environments

For the transfer learning tasks, we use *Transfer Environments for MuJoCo* [8], a set of gym environments for studying potential improvements in transfer learning tasks. The task

involves an *Ant* as an agent which optimizes a gait to crawl across the landscape. The expert demonstrations are obtained from the optimal policy in the basic Ant environment. We disable the agent Ant in two ways for two transfer learning tasks. In *Big Ant* tasks, the length of all legs is doubled, no extra joints are added though. The *Amputated Ant* task modifies the agent by shortening a single leg to disable it. These transfer tasks require the learning of a true disentangled reward of walking sideways instead of directly imitating and learning the reward specific to the gait movements. These manipulations are shown in Figure 5.1.



(a) Ant environment **(b)** Big Ant environment **(c)** Amputated Ant environment

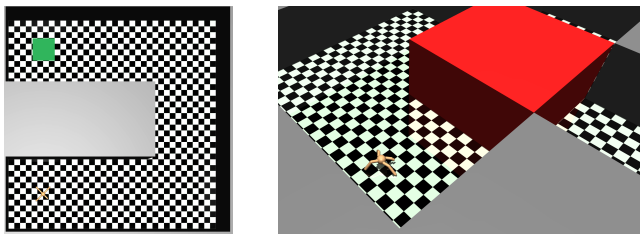
Figure 5.1: MuJoCo Ant Gait transfer learning task environments. When the Ant is disabled, it must position itself correctly to crawl forward. This requires a different initial policy than the original environment where the Ant must only crawl sideways.

We believe that exploiting a hierarchical framework such as the options framework is useful in these tasks because these tasks are in practice hierarchical. The gait, which involves making different motions of each limb in succession could be thought of as subtasks.

In addition, we adopt more complex hierarchical environments that require both locomotion and object interaction. In the first environment, the Ant must interact with a large movable block. This is called the *Ant-Push* environment [11]. To reach the goal, the Ant must complete two successive processes: first, it must move to the left of the block and then push the block right, which clears the path towards the target location. There is

a maximum of 500 timesteps. These can be thought of as hierarchical tasks with *pushing to the left*, *pushing to the right* and *going to the goal* as sub-goals.

We also utilize an Ant-Maze environment [15] where we have a simple maze with a goal at the end. The agent receives a reward of $+1$ if it reaches the goal and 0 elsewhere. More specifically, in the Ant Maze and Push tasks we define a success as being within an L2 distance of 5 from the goal on the last step of the episode. The Ant must learn to make two turns in the maze, the first is down the hallway for one step and then a turn towards the goal. Again, we see hierarchical behavior in this task: we can think of sub-goals consisting of *learning to exit the first hall of the maze*, then *making the turn* and finally *going down the final hall towards the goal*. The two complex environments are shown in Figure 5.2.



(a) Ant-Maze environment

(b) Ant-Push environment

Figure 5.2: MuJoCo Ant Complex Gait transfer learning task environments. We perform these transfer learning tasks with the Big Ant and the Amputated Ant.

5.1.2 MuJoCo Continuous Control Tasks

In this section, we describe the structure of the objects that gait in the continuous control benchmarks and the reward functions. For the transfer learning tasks, we use the same reward function described here for the Ant.

Walker: The walker is a planar biped. There are 7 rigid links comprised of legs and a torso. This includes 6 actuated joints. This task is particularly prone to falling. The state space is of 21 dimensions. The observations in the states include joint angles (θ),

joint velocities (v_x), the center of mass's coordinates. The reward function is $r(s,a) = v_x 0.005||a||_2^2$. The termination condition occurs when $z_{\text{body}} < 0.8, z_{\text{body}} > 2.0$ or $||\theta_y|| > 1.0$.

Half-Cheetah: The half-cheetah is a planar biped also like the Walker. There are 9 rigid links comprised of 9 actuated joints, a leg and a torso. The state space is of 20 dimensions. The observations include joint angles, the center of mass's coordinates, and joint velocities. The reward function is $r(s,a) = v_x 0.05||a||_2^2$. There is no termination condition.

Ant: The Ant has four legs with 13 rigid links in its structure. The legs have 8 actuated joints. The state space is of 125 dimensions. This includes joint angles, joint velocities, coordinates of the center of mass, the rotation matrix for the body, and a vector of contact forces. The function is $r(s,a) = v_x 0.005||a||_2^2 C_{\text{contact}} + 0.05$, where C_{contact} is a penalty for contacts to the ground. This is $5 \times 104||F_{\text{contact}}||_2^2$. The term F_{contact} is the contact force. Its values are clipped to be between 0 and 1. The termination condition occurs $z_{\text{body}} < 0.2$ or $z_{\text{body}} > 1.0$.

5.2 MuJoCo Experimental Setup

In this section, we describe the procedures for each of our experiments in the MuJoCo simulator.

5.2.1 Transfer Tasks

For the transfer tasks, we are given a set of expert demonstrations of our (undeformed) Ant performing the task optimally. As an example, this demonstration could be walking optimally in the case of gait tasks or achieving the goal cells on the Ant-Push tasks. These expert demonstrations are obtained by training a policy on these environments using vanilla PPO until 2 million iterations (20 million on the AntPush and Maze tasks) and then sampling trajectories from this policy.

We train a policy using each of the baseline methods and our method on these expert demonstrations for 500 time steps on the gait environments and 500 timesteps on the complex ones. Then we take the trained reward function and use this reward function on the transfer environments and observe the reward obtained by optimizing a new policy on this reward function. A random seed in these experiments refers to different random values that we use to initialize our input parameters.

5.2.2 Continuous Control Tasks

These tasks involve no transfer learning. We simply follow the same method to obtain demonstrator trajectories as described before. We then train our policy given these demonstrator trajectories on the target environment using our method and the baseline methods. We do not use the disabled Ants in these tasks, only the normal Ant.

5.2.3 Parameters for MuJoCo Experiments

For these experiments, we use PPO to obtain an optimal policy given our ground truth rewards. This is used to obtain expert demonstrations. We sample 50 expert trajectories. PPOC is used for the policy optimization step for the policy over options. We tune the deliberation cost hyper-parameter via cross-validation. The optimal deliberation cost found was 0.1 for PPOC. We also use state-only rewards for the transfer tasks. The hyperparameters for our policy optimization are given in Table 5.1.

Our discriminator is a neural network with the optimal architecture of 2 linear layers of 50 hidden states, each with ReLU activation followed by a single node linear layer for output. We also tried a variety of hidden states including 100 and 25 and tanh activation during our hyperparameter optimization step using cross-validation.

The policy network has 2 layers of 64 hidden states. A batch size of 64 or 32 is used for 1 and any number of options greater than 1 respectively. No mini-batches are used in the discriminator since the recursive loss must be computed. There are 2048 timesteps per

batch. Generalized Advantage Estimation is used to compute advantage estimates. We list additional network parameters in the next section. The output of the policy network gives the Gaussian mean and the standard deviation. This is the same procedure as in [29].

Table 5.1: Policy Optimization parameters for MuJoCo

Parameter	Value
Discr. Adam optimizer learning rate	$1 \cdot 10^{-3}$
Adam ϵ	$1 \cdot 10^{-5}$
PPOC Adam optimizer learning rate	$3 \cdot 10^{-4}$
GAE λ	0.95
Entropy coefficient	10^{-2}
value loss coefficient	0.5
discount	0.99
batch size for PPO	64 or 32
PPO epochs	10
clip parameter	0.2

5.3 MuJoCo Transfer Learning Task Results

For the transfer learning environments, we simply display the mean reward achieved using the policy learned on the transferred reward function. We train the policy multiple times (with different random seeds) with one transferred reward function. This is an average reward obtained after 10 runs of different random seeds.

In our experiments, we refer to our method as oIRL (Optionated Inverse Reinforcement Learning).

Table 5.2 shows the results in terms of reward achieved for the Ant gait transfer tasks. As we can see, in both experiments our algorithm performs better than AIRL. Remark that the ground truth is obtained with PPO after 2 million iterations (therefore much less sample efficient than IRL). We see that the AmputatedAnt performs best with 4 options and the BigAnt with 2 options. We should note that in terms of complexity and training

Table 5.2: The mean reward obtained (higher is better) over 100 runs for the Gait transfer learning tasks. We also show the results of PPO optimizing the ground truth reward. The value after $\pm$ represents one standard deviation.

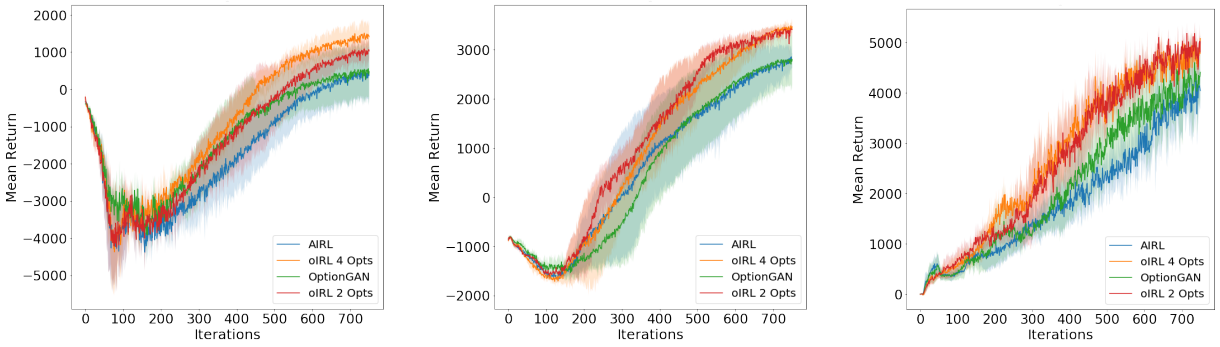
	Big Ant	Amputated Ant
AIRL (Primitive)	-11.6 $\pm$1.1	134.3 $\pm$8.4
2 Opts oIRL	7.4 $\pm$0.4	135.0 $\pm$13.2
4 Opts oIRL	**11.0** $\pm$0.6	**173.2** $\pm$15.2
OptionGAN	-13.3 $\pm$2.0	94.4 $\pm$8.2
Ground Truth	142.9	335.4

time as well as memory required, our algorithm performs far worse than AIRL. This is due to the fact that we computing the recursive loss function we must first store the environment at each step. This is required since we sample different actions depending on if we terminate or continue with the option in our loss computation. In addition, it is an added layer of complexity to obtain these roll-outs.

Table 5.3: The mean reward obtained (higher is better) over 100 runs for the MuJoCo Ant Complex Gait transfer learning tasks. We also show the results of PPO optimizing the ground truth reward. Amp is Amputated.

	Big Ant Maze	Amp Ant Maze	Big Ant Push	Amp Ant Push
AIRL (Primitive)	0.28 $\pm$0.03	0.18 $\pm$0.04	0.22 $\pm$0.00	0.15 $\pm$0.03
2 Opts oIRL	**0.68** $\pm$0.07	0.36 $\pm$0.03	0.62 $\pm$0.10	0.51 $\pm$0.12
4 Opts oIRL	0.63 $\pm$0.08	**0.40** $\pm$0.07	**0.64** $\pm$0.10	**0.52** $\pm$0.11
OptionGAN	0.08 $\pm$0.07	0.09 $\pm$0.02	0.17 $\pm$0.04	0.08 $\pm$0.01
Ground Truth	0.96	0.98	0.90	0.93

Table 5.3 shows that oIRL performs better than AIRL in all of the complex hierarchical transfer tasks. In some tasks such as the Maze environment, AIRL fails to have any or very few successful runs while our method achieves reasonably high reward. In the BigAnt push task, AIRL achieves only very minimal reward where oIRL succeeds to perform the task in some cases.



(a) Ant **(b)** Half Cheetah **(c)** Walker

Figure 5.3: MuJoCo Continuous control locomotion tasks showing the mean reward (higher is better) achieved over 750 iterations of the benchmark algorithms for 10 random seeds. The shaded area represents the standard deviation.

5.4 MuJoCo Continuous Control Benchmarks

We also test our algorithm on a number of robotic continuous control benchmark tasks. These tasks do not involve the transfer of reward functions and in no way test transfer learning. The purpose of these tests is to show that our algorithm is also a viable IRL method for more standard tasks.

We show the plots of the average reward for each iteration during training in Figure 5.3. Achieving a higher reward in fewer iterations is better for these experiments. We examine the Ant, the Half Cheetah, and the Walker MuJoCo gait/locomotion tasks. We train in these experiments with 10 random seeds. The results are quite similar between the benchmarks. Using a policy over options shows reasonable improvements in each task.

5.5 Gym MiniGrid Experiments

In these experiments, we create transfer learning tasks in a 2D GridWorld environment.

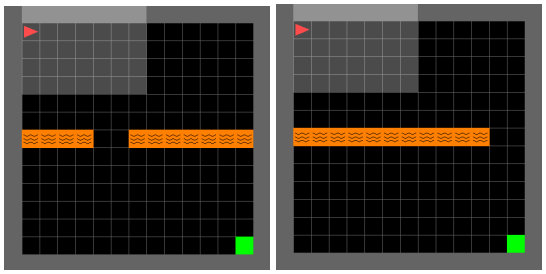
We build our 2D grid-world environments using the Gym "MiniGrid" package [7]. The environments are fully observable and each observation is a (w, h, 3) tensor. At each

timestep, the agent can change its direction, actions are as follows: *turn-left*, *turn-right*, *move-forward*. Facing a wall, the agent will stay in the same state if it moves forward into the wall.

The rewards in these tasks are sparse, we gave a non-zero reward to the agent only when it fully completed the mission, and the magnitude of the reward was $1-0.9\cdot n/n_{\text{max}}$, where n is the length of the successful episode and n_{max} is the maximum number of steps that we allowed for completing the episode, different for each mission. If the agent goes into lava or reaches the maximum number of steps authorized for each episode, the episode ends with 0 rewards.

5.5.1 Grid Transfer Learning Environments

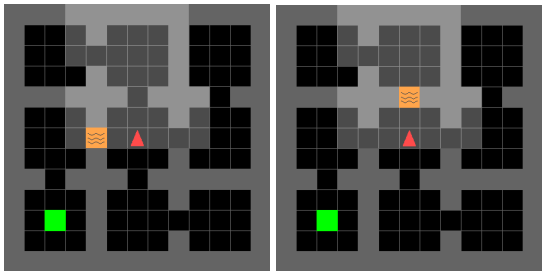
We create transfer learning environments in a 2D Maze environment with lava blockades. The goal of the agent is to go through the opening in a row of lava cells and reach a goal on the other end. For the transfer learning task, we train the agent on an environment where the "crossing" path requires the agent to go through the middle (LavaCrossing-M) and then the rewards are directly transferred and used on a GridWorld of the same size where the crossing is on the right end of the room (LavaCrossing-R). The 2 environments are shown in Figure 5.4. We can think of 2 sub-tasks in this environment, going to the lava crossing and then going to the goal.



(a) LavaCrossing-M MiniGrid Env **(b)** LavaCrossing-R MiniGrid Env

Figure 5.4: The MiniGrid transfer learning task set 1. Here the policy is trained on (a) using our method and the baseline methods and then transferred to be used on environment (b). The green cell is the goal.

In the 2nd transfer learning task, we have a simple maze where the path is blocked by lava to the right of the agent. We train the agent in this environment (FlowerMaze-R) and then transfer the agent's policy to the maze where the top path is blocked (FlowerMaze-T) shown in Figure 5.5.



(a) FlowerMaze-R MiniGrid Env **(b)** FlowerMaze-T MiniGrid Env

Figure 5.5: The MiniGrid transfer learning task set 2. Here the policy is trained on (a) using our method and the baseline methods and then transferred to be used on environment (b).

5.5.2 Grid Experiment Parameters

For experiments, we used the PPOC algorithm with parallelized data collection and GAE. 0.1 is the optimal deliberation cost. Each environment is run with 10 random network

initialization. As before, in Table 5.4, we show some of the policy optimization parameters for MiniGrid Tasks. We rely on an actor-critic network architecture for these tasks. Since the state space is relatively large and spatial features are relevant, we use 3 convolutional layers in the network. The network architecture is detailed in Figure 5.6. The values n and m are defined by the grid dimensions.

The discriminator network is a neural network with the optimal architecture of 3 linear layers of 150 hidden states, each with ReLU activation followed by a single node linear layer for output.

Table 5.4: Policy optimization parameters for benchmark tasks in MiniGrid

Parameter	Value
Adam optimizer learning rate	$7 \cdot 10^{-4}$
Adam ϵ	$1 \cdot 10^{-5}$
value loss coefficient	0.5
discount	0.99
maximum norm of gradient in PPO	0.5
number of PPO epochs	4
batch size for PPO	256
entropy coefficient	10^{-2}
clip parameter	0.2

5.6 Grid Results

Again, we show the mean reward after 10 runs using the policy optimized with the reward function transfers on the environments in Table 5.5. The 4 option oIRL achieved the highest reward on the LavaCrossing tasks. The FlowerMaze task was quite difficult with most algorithms obtaining very low reward. Options still result in a large improvement.

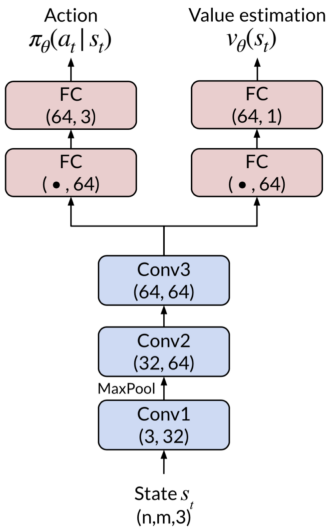


Figure 5.6: Architectures of the actor-critic policies on MiniGrid. Conv is Convolutional Layer and filter sized is described below. FC is a fully connected layer.

Table 5.5: The mean reward obtained (higher is better) over 10 runs for the Maze transfer learning tasks. We also show the results of PPO optimizing the ground truth reward.

	LAVACROSSING	FLOWERMAZE
AIRL (PRIMITIVE)	0.64 ±0.11	0.11 ±0.03
2 OPTS OIRL	0.74 ±0.13	0.33 ±0.06
4 OPTS OIRL	**0.86** ±0.09	**0.36** ±0.05
OPTIONGAN	0.41 ±0.12	0.07 ±0.06
GROUND TRUTH	1.00	1.00

5.7 Interpretation

We have tested a wide variety of transfer tasks involving the disability on a MuJoCo Ant. The increases in performance by using our algorithm against OptionGAN and AIRL indicate 2 main conclusions. The first is that a policy over options with disentangled rewards results in more generalizable rewards in transfer tasks than just a policy over options or primitive disentangled rewards. The results are especially clear on the hierarchical environments, where using a policy over options is beneficial. We also show that this algorithm performs better than AIRL and a standard HRL algorithm (OptionGAN) in

continuous control benchmarks. Our algorithm achieves a higher reward with better sample efficiency. While in these experiments, we only deal with simulation, the algorithm could be useful in more realistic scenarios such as hierarchical robotic transfer tasks. If a robot's structural components are modified in a similar way to how we modify the Ant, we should see similar improvements. Also, as video games (such as the Arcade game environments) are highly hierarchical, our algorithm could potentially perform well in these tasks. We could perform transfer tasks in terms of different 'levels' in a game.

We also note that selecting the number of options appears to be important given the type and complexity of the task we have. For example, 4 options usually performs better than 2 options with the exception of the Big Ant Maze. A potential reason for this is because the maze tasks are slightly less complex than the other environments. It is seen that 4 options performs better in 3/4 of the most *complex* hierarchical tasks.

Chapter 6

Conclusion and future work

This thesis presents Option-Inverse Reinforcement Learning (oIRL), the first hierarchical IRL algorithm able to learn disentangled rewards. We validate oIRL on a wide variety of tasks, including transfer learning tasks, locomotion tasks, complex hierarchical transfer RL environments, and GridWorld transfer navigation tasks, and we compare our results with state-of-the-art algorithms. Combining options with a disentangled IRL framework results in highly portable reward functions. Our empirical studies show clear improvements for transfer learning. The algorithm is also shown to perform well in standard (non-transfer) continuous control benchmark tasks.

6.0.1 Real world implications

While we were unable to benchmark our algorithm on real robotics domains, the performance in simulated continuous control tasks suggests that this work can provide improvements in real transfer simulations with physical robots. MuJoCo is used as a physics simulator to provide some simulation of robotic tasks and the results are promising on these simulations. We could also apply this to a video game learning environment and perhaps learn better transfer-ability of skills between different types of worlds or similar styles of games. Autonomous driving also heavily relies on transferable skills in situations like driving in new areas or different road conditions.

6.0.2 Future work

For future work, we wish to test other sampling methods (e.g., Markov-chain Monte Carlo) to estimate the implicit discriminator-generator pair's distribution in our GAN, such as Metropolis-Hastings GAN [39]. We also wish to explore other methods for inferring option trajectories. Analyzing our algorithm using physical robotics tests for tasks that require multiple sub-tasks would be an interesting, albeit challenging, future course of research.

Bibliography

[1] Pieter Abbeel and Andrew Y. Ng, *Apprenticeship learning via inverse reinforcement learning*, Proceedings of the Twenty-First International Conference on Machine Learning, 2004, pp. 1.

[2] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané, *Concrete problems in ai safety*, arXiv **1606.06565** (2016).

[3] Pierre-Luc Bacon, Jean Harb, and Doina Precup, *The option-critic architecture*, Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, 2017, pp. 1726–1734.

[4] Michael Bain and Claude Sammut, *A framework for behavioural cloning*, Machine intelligence 15, intelligent agents, 1999, pp. 103–129.

[5] Richard Bellman, *Dynamic programming and lagrange multipliers*, Proceedings of the National Academy of Sciences (1956), 767–769.

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba, *OpenAI Gym*, 2016.

[7] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal, *Minimalistic gridworld environment for OpenAI Gym*, GitHub, 2018.

[8] Elizabeth Chu and Sebastien Arnold, *Transfer environments for MuJoCo*, 2018.

[9] Thomas Degris, Martha White, and Richard S. Sutton, *Off-policy actor-critic*, Proceedings of the Twenty-Ninth International Coference on International Conference on Machine Learning, 2012, pp. 179–186.

[10] Dmitriy Drusvyatskiy and Adrian S. Lewis, *Error bounds, quadratic growth, and linear convergence of proximal methods*, Mathematics of Operations Research **43** (2016), no. 3, 919–948.

[11] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel, *Benchmarking deep reinforcement learning for continuous control*, Proceedings of the Thirty-Third International Conference on International Conference on Machine Learning, 2016, pp. 1329–1338.

[12] Oliver Duerr, Beate Sick, and Elvis Murina, *Probabilistic deep learning: With python, keras and tensorflow probability*, Manning Publications, 2020.

[13] Chelsea Finn, Paul Christiano, Pieter Abbeel, and Sergey Levine, *A connection between generative adversarial networks, inverse reinforcement learning, and energy-based models*, in Proceedings of the Thirtieth International Conference on Neural Information Processing Systems Workshop on Adversarial Training (2016).

[14] Chelsea Finn, Sergey Levine, and Pieter Abbeel, *Guided cost learning: Deep inverse optimal control via policy optimization*, Proceedings of the Thirty-Third International Conference on International Conference on Machine Learning, 2016, pp. 49–58.

[15] Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel, *Reverse curriculum generation for reinforcement learning*, Proceedings of Machine Learning Research **78** (2017), 482–495.

[16] Justin Fu, Katie Luo, and Sergey Levine, *Learning robust rewards with adverserial inverse reinforcement learning*, Proceedings of the Sixth International Conference on Learning Representations, 2018.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*, MIT Press, 2016. `http://www.deeplearningbook.org`.

[18] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, *Generative adversarial nets*, Proceedings of the 27th International Conference on Neural Information Processing Systems, pp. 2672–2680.

[19] Karol Hausman, Yevgen Chebotar, Stefan Schaal, Gaurav Sukhatme, and Joseph J. Lim, *Multi-modal imitation learning from unstructured demonstrations using generative adversarial nets*, Proceedings of the 31st International Conference on Neural Information Processing Systems, 2017, pp. 1235–1245.

[20] Peter Henderson, Wei-Di Chang, Pierre-Luc Bacon, David Meger, Joelle Pineau, and Doina Precup, *Optiongan: Learning joint reward-policy options using generative adversarial inverse reinforcement learning*, In proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, 2018, pp. 3199–3206.

[21] Jonathan Ho and Stefano Ermon, *Generative adversarial imitation learning*, Proceedings of the Thirtieth International Conference on Neural Information Processing Systems, 2016, pp. 4572–4580.

[22] Martin Klissarov, Pierre-Luc Bacon, Jean Harb, and Doina Precup, *Learnings options end-to-end for continuous action tasks*, arXiv **1712.00004** (2017).

[23] George Konidaris and Andrew Barto, *Building portable options: Skill transfer in reinforcement learning*, Proceedings of the Twentieth International Joint Conference on Artificial Intelligence, 2007, pp. 895–900.

[24] Sanjay Krishnan, Animesh Garg, Richard Liaw, Lauren Miller, Florian T. Pokorny, and Kenneth Y. Goldberg, *HIRL: hierarchical inverse reinforcement learning for long-horizon tasks with delayed rewards*, ArXiv **1604.06508** (2016).

[25] Oliver Kroemer, Scott Niekum, and George Konidaris, *A review of robot learning for manipulation: Challenges, representations, and algorithms*, ArXiv **1907.03146** (2019).

[26] Andrew Y. Ng, Daishi Harada, and Stuart J. Russell, *Policy invariance under reward transformations: Theory and application to reward shaping*, Proceedings of the Sixteenth International Conference on Machine Learning, 1999, pp. 278–287.

[27] Andrew Y. Ng and Stuart J. Russell, *Algorithms for inverse reinforcement learning*, Proceedings of the Seventeenth International Conference on Machine Learning, 2000, pp. 663–670.

[28] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel, *High-dimensional continuous control using generalized advantage estimation*, Proceedings of the Fourth International Conference of Learning Representations, 2016.

[29] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov, *Proximal policy optimization algorithms*, arXiv **1707.06347** (2017).

[30] Mohit Sharma, Arjun Sharma, Nicholas Rhinehart, and Kris M. Kitani, *Directed-info GAIL: Learning hierarchical policies from unsegmented demonstrations using directed information*, Proceedings of the Seventh International Conference on Learning Representations, year=2019,.

[31] Paul Smaldino and Peter Richerson, *The origins of options*, Frontiers in Neuroscience **6** (2012), 50.

[32] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov, *Dropout: A simple way to prevent neural networks from overfitting*, Journal of Machine Learning Research **15** (2014), no. 1, 1929–1958.

[33] Richard S. Sutton, *Learning to predict by the methods of temporal differences*, Machine Learning **3** (1988), no. 1, 9–44.

[34] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour, *Policy gradient methods for reinforcement learning with function approximation*, Proceedings of the Twelfth International Conference on Neural Information Processing Systems, 1999, pp. 1057–1063.

[35] Richard S. Sutton, Doina Precup, and Satinder Singh, *Between MDPs and Semi-MDPs: A framework for temporal abstraction in reinforcement learning*, Artificial Intelligence **112** (1999), no. 1–2, 181–211.

[36] Matthew E. Taylor and Peter Stone, *Transfer learning for reinforcement learning domains: A survey*, Journal of Machine Learning Research **10** (2009), 1633–1685.

[37] A. N. Tikhonov, A. S. Leonov, and A. G. Yagola, *Nonlinear ill-posed problems*, Proceedings of the First World Congress on World Congress of Nonlinear Analysts, 1996, pp. 505–511.

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

[39] Ryan Turner, Jane Hung, Eric Frank, Yunus Saatchi, and Jason Yosinski, *Metropolis-hastings generative adversarial networks*, Proceedings of the Thirty Sixth International Conference on Machine Learning, 2019, pp. 6345–6353.

[40] David Venuto, Jhelum Chakravorty, Leonard Boussioux, Junhao Wang, Gavin McCracken, and Doina Precup, *oIRL: Robust adversarial inverse reinforcement learning with temporally extended actions*, arXiv **2002.09043** (2020).

[41] T. P. Vogl, J. K. Mangis, A. K. Rigler, W. T. Zink, and D. L. Alkon, *Accelerating the convergence of the back-propagation method*, Biological Cybernetics **59** (1988), 257–263.

[42] Ziyu Wang, Josh Merel, Scott Reed, Greg Wayne, Nando de Freitas, and Nicolas Heess, *Robust imitation of diverse behaviors*, Proceedings of the Thirty First International Conference on Neural Information Processing Systems, 2017, pp. 5326–5335.

[43] Christopher J.C.H. Watkins and Peter Dayan, *Technical note: Q-learning*, Machine Learning **8** (1992), 279–292.

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