

MASTER IN ARTIFICIAL INTELLIGENCE

FINAL MASTER PROJECT
Thesis

Evaluating the Robustness of GAN-Based Inverse Reinforcement Learning Algorithms

Submitted by
Johannes Heidecke

Supervised by
Mario Martin
Dario Garcia-Gasulla

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) - Barcelona Tech

Facultat de Matemàtiques de Barcelona
Universitat de Barcelona (UB)

Escola Tècnica Superior d'Enginyeria
Universitat Rovira i Virgili (URV)

January 15, 2019

Abstract

Providing a suitable reward function is often a key bottleneck when setting up a reinforcement learning algorithm to solve a new task. Inverse reinforcement learning (IRL) aims to automatically learn a reward function based on expert demonstrations, which can then be used to train a reinforcement learning agent. Reward functions learned with IRL are claimed to be more robust and transferable than policies learned with behavioral cloning by imitating the expert.

We evaluate the robustness of learned reward functions when transferred to similar tasks with different transition dynamics. For this purpose, we combine guided cost learning and adversarial inverse reinforcement learning with a soft actor critic reinforcement learning agent. Both of these methods can be seen as special cases of generative adversarial networks. Our results exceed the previously reported state of the art for the Pendulum environment. To the best of our knowledge, we are the first to apply inverse reinforcement learning to the Lunar Lander environment. Some modifications are proposed that achieve faster and more stable training.

We find that adversarial IRL learns reward functions that are more robust than both guided cost learning and the behavioral cloning method GAIL. However, the achieved robustness is not sufficient, as on all explored transfer tasks the learned policies fall short of expert performance.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Contributions	2
1.3	Overview	2
2	Prerequisites	4
2.1	Reinforcement Learning	4
2.1.1	The Reinforcement Learning Problem	4
2.1.2	Value functions and Q-learning	6
2.1.3	Experience Replay	7
2.1.4	Policy Optimization Methods	7
2.1.5	Actor Critic Algorithms	8
2.1.6	Exploration and Maximum Entropy RL	9
2.1.7	Soft Actor Critic	11
2.2	Generative Adversarial Networks	12
2.3	Inverse Reinforcement Learning	12
2.3.1	Motivation	13
2.3.2	Difficulties of IRL	16
2.3.3	Linear Programming Approaches	16
2.3.4	Apprenticeship Learning via IRL	20
2.3.5	Bayesian IRL	21
2.3.6	Maximum Entropy IRL	26
2.3.7	Maximum Causal Entropy IRL	28
2.3.8	Guided Cost Learning	29
2.3.9	Equivalence of IRL and GAN	31
2.3.10	Adversarial IRL	33
2.3.11	Summary	34

3 Methods	36
3.1 IRL Algorithms	36
3.1.1 General IRL Implementation	37
3.1.2 Reinforcement Learning	37
3.1.3 Reward Function Pre-training	38
3.1.4 Reward Magnitude Penalty	40
3.1.5 Mixture Batch	40
3.1.6 GCL Modifications	41
3.1.7 AIRL Modification	42
3.2 RL Environments	43
3.2.1 Pendulum	43
3.2.2 Lunar Lander	44
3.3 Expert Data Collection	45
3.3.1 Pendulum	46
3.3.2 Lunar Lander	46
3.4 Baseline Algorithms	46
3.4.1 True Reward Baseline	48
3.4.2 GAIL Baseline	48
3.4.3 Random Agent Baseline	49
3.5 Robustness Evaluation	49
3.5.1 Pendulum	50
3.5.2 Lunar Lander	50
3.6 Shaped Reward Loss Metric	51
4 Results	55
4.1 Original Task Performance	55
4.1.1 Sampling Policy for Pendulum	56
4.1.2 Fixed Reward Function for Pendulum	58
4.1.3 Sampling Policy for Lunar Lander	60
4.1.4 Fixed Reward Function for Lunar Lander	62
4.2 Robustness of Learned Reward Function	64
4.2.1 Pendulum	65
4.2.2 Lunar Lander	67
4.3 Pre-Trained Reward Function	68
4.4 Expert Data Quantity	70

4.5	Shaped Reward Loss	71
5	Discussion	73
5.1	Conclusions and Contributions	73
5.1.1	Inverse Reinforcement Learning	73
5.1.2	Proposed Modifications on IRL Algorithms	73
5.1.3	Robustness Results	74
5.1.4	Training with Fixed Reward Functions	75
5.2	Future Work	76
	References	76

List of Figures

2.1	Comparison of policies for exploration	10
3.1	Loss of reward function pre-training classifier.	39
3.2	Pendulum environment	43
3.3	Lunar Lander environment	44
3.4	Expert performance on Lunar Lander environment	47
3.5	Random agent performance on Lunar Lander environment	47
3.6	Baselines for the Pendulum environment.	48
3.7	Baselines for the Lunar Lander environment.	48
3.8	Baselines for the Pendulum (max-speed 6) environment.	50
3.9	Baselines for the Pendulum (max-speed 10) environment.	51
3.10	Lunar Lander (modified) environment	51
3.11	Baselines for the Lunar Lander (modified) environment.	52
4.1	Reward Magnitude Penalty with GCL on Pendulum environment . .	56
4.2	Reward Magnitude Penalty and mixture batch with GCL on Pendulum environment	57
4.3	Reward Magnitude Penalty with AIRL on Pendulum environment . .	57
4.4	GCL results on Pendulum environment	58
4.5	AIRL results on Pendulum environment	58
4.6	GCL and AIRL comparison on Pendulum environment	59
4.7	Performance in true reward on Pendulum after training on reward estimate	59
4.8	Visualization of learned reward functions for Pendulum environment.	61
4.9	GCL results on Lunar Lander environment	61
4.10	AIRL results on Lunar Lander environment	62
4.11	GCL and AIRL comparison on Lunar Lander environment	63

4.12	Performance in true reward on Lunar Lander after training on reward estimate	63
4.13	Visualization of learned reward functions for Lunar Lander environment	64
4.14	GCL results on Pendulum (max-speed 6) environment	65
4.15	AIRL results on Pendulum (max-speed 6) environment	66
4.16	GCL results on Pendulum (max-speed 10) environment	66
4.17	AIRL results on Pendulum (max-speed 10) environment	67
4.18	GCL results on Lunar Lander (modified) environment	67
4.19	AIRL results on Lunar Lander (modified) environment	68
4.20	Pre-trained GCL results on Lunar Lander environment	69
4.21	Pre-trained AIRL results on Lunar Lander environment	69
4.22	Shaped reward loss for Pendulum environment	72

List of Tables

2.1	Bellman expectation equations for state and action values	5
2.2	Bellman optimality equations for state and action values	5
2.3	Analogies between IRL and GAN	33
2.4	Summary of the described IRL algorithms	35
3.1	Reinforcement learning hyperparameters used for all experiments . .	38
4.1	Improvement over state of the art for Pendulum environment	60
4.2	Average return with fixed reward functions	63
4.3	Effect of pre-training reward function on early training stages	70
4.4	Effect of expert data quantity for Pendulum environment	70
4.5	Effect of expert data quantity for Lunar Lander environment	71

Acronyms

AIRL adversarial inverse reinforcement learning. 2, 4, 33, 35–38, 40–43, 45, 47–50, 56–58, 60, 62, 63, 65–76

BIRL Bayesian inverse reinforcement learning. 22–26, 52

CIRL cooperative inverse reinforcement learning. 14

DQN deep Q-network. 7, 8

EM expectation maximization. 26

GAIL generative adversarial imitation learning. 33, 46–50, 60, 62, 65–68, 74, 76

GAN generative adversarial network. 2, 4, 12, 31–33, 49, 75

GCL guided cost learning. 2, 4, 31, 33, 35–38, 40–42, 45, 47–50, 56–58, 60, 62, 63, 65, 67, 70–76

i.i.d. independently identically distributed. 22, 42

IRL inverse reinforcement learning. 1–3, 13–16, 20, 22–24, 26–33, 35–40, 43–46, 48, 49, 51, 52, 54, 55, 60, 62, 70, 73, 76

KL divergence Kullback-Leibler divergence. 11

MAP maximum a posteriori. 24, 26

MCMC Markov chain Monte Carlo. 23, 25, 35

MDP Markov decision process. 4–7, 16, 25, 27–29, 31, 33–36, 40, 44, 45, 50, 54, 74–76

POMDP partially observable Markov decision process. 6

PReLU parametric rectified linear units. 37

ReLU rectified linear unit. 37, 39, 53

RL reinforcement learning. 1, 2, 4, 9, 14–16, 20, 27, 30, 40, 46, 50, 53, 55, 56, 58–60, 62, 68, 70, 73, 75, 76

SAC soft actor critic. 2, 4, 11, 30, 37, 38, 42, 46, 48, 50, 59, 60, 63, 73, 76

SGD stochastic gradient descent. 42

SVM support vector machine. 21

TD-learning temporal difference learning. 6

TD3 Twin Delayed Deep Deterministic Policy Gradient. 11

TRPO trust region policy optimization. 37, 49, 59, 60

1 | Introduction

This master's thesis project evaluates the robustness of inverse reinforcement learning algorithms to distributional shift, i.e. a change in transition dynamics between the environment used during training and during operation of a reinforcement learning agent. Different approaches were investigated with the goal of making learned reward functions more generalizable across similar environments.

1.1 Motivation

When setting up a reinforcement learning (RL) system for a specific task, one of the hardest aspects is to specify a reward function that correctly represents the desired goal states and preferences of how to reach them [1], [2], [3]. Misspecified reward functions can lead to undesirable or even dangerous policies [1], [4]. Inverse reinforcement learning (IRL), which is the core concept covered in this thesis, is the problem of learning a reward function estimate for a reinforcement learning problem such that observed expert demonstrations are consistent with said estimate [5], [6]. This reward function estimate can then be used to train reinforcement learning agents in hopes of them performing well under the true (unknown) reward function.

IRL stands in contrast with behavioral cloning methods of directly learning policies from expert demonstrations, e.g. by means of supervised learning [7]. These approaches, while fast and relatively straightforward, often suffer from brittleness and compounding errors once situations are encountered that are not well-covered in the training data [8], [9].

One of the motivations behind using IRL over supervised methods of behavioral cloning is that the learned reward function can, in theory, be used as a succinct, robust, and transferable description of a task [6], even for a different type of agent in a slightly different environment. For example, human demonstrations of assembling a device at their workplace could be used to train robots in various factories to do the same based on the learned reward function. However, this requires the learned reward to be robust to distributional shift: a change in the probabilities of transitioning to new world states between the training and test environments. Therefore, it is often required that the learned reward function transfers well to environments with small variations compared to the training environment.

In this thesis, we evaluate the robustness of transferring reward functions to environments that are slightly different from the one encountered during training. Developing our understanding of inverse reinforcement learning algorithms and their

capability to generalize over different transition dynamics is an important step towards making reinforcement learning more applicable to a wider range of problems for which reward functions could be learned from expert demonstrations.

1.2 Contributions

This thesis makes several contributions that are summarized below.

To the best of our knowledge, this is the first time that inverse reinforcement learning has been combined with the soft actor critic (SAC) reinforcement learning algorithm. This is a well-suited combination as both the inverse and forward part of the implementation are based on the maximum entropy principle [10], [11], [12]. As an off-policy method, SAC maintains an experience replay memory [13] that can readily be used to construct training batches for the training of the two implemented IRL algorithms, guided cost learning (GCL) [14] and adversarial inverse reinforcement learning (AIRL) [15].

For the Pendulum environment, we achieved better results than the previous state of the art for IRL reported in [15]. In addition, as far as we can tell this is the first time that IRL has successfully been applied to the Lunar Lander environment, a high-dimensional problem with a non-trivial reward function that is often hard to learn for RL agents. By evaluating the robustness of learned reward functions when transferred to a testing environment with different transition dynamics, we found that reward functions learned with AIRL generalize better to the new situations, but still fall short of results achieved with the true reward function.

We evaluate the effect of pre-training non-linear reward functions represented by artificial neural networks on the convergence speed during training. This is done based on the provided expert demonstrations and transitions sampled from a random policy. A novel metric for IRL, termed shaped reward loss, is investigated. This metric operates directly on two reward functions to compare their similarity while taking optimality-preserving reward function transformations into account. In addition, this thesis offers a comprehensive review of some of the most important and relevant IRL algorithms.

1.3 Overview

This thesis is split into four main parts. In chapter 2, the theoretical background required for understanding later parts is covered. Section 2.1 sets the broader context of this work and describes the basics of reinforcement learning. This includes general notation and definitions, as well as a description of SAC, the particular RL algorithm used in this thesis. As the two IRL methods that were implemented are essentially equivalent to a certain kind of generative adversarial network (GAN), this family of generative models is briefly introduced in section 2.2. For the largest part of the chapter, section 2.3 surveys the field of inverse reinforcement learning and introduces some of the most important algorithms that have been proposed. This concludes

with a description of the two recent algorithms that were chosen to be implemented: guided cost learning and adversarial inverse reinforcement learning.

Chapter 3 provides detailed explanations of the methods and experiments that were implemented and run for this thesis. This starts with specifics of the two implemented IRL algorithms in section 3.1, including a description of the modifications that make our implementation differ from the original models proposed in [14] and [15]. The two environments, Pendulum and Lunar Lander, are introduced in section 3.2, followed by a description of how expert demonstrations were collected for each of them in section 3.3. Our results are compared against three different baselines which are detailed in section 3.4. To test the robustness of learned reward functions on new environments, we propose modified environments for testing in section 3.5. Section 3.6 introduces a new type of metric to measure the quality of IRL results.

In chapter 4 we present the results of our experiments. First, the performance on the original environments during and after training is presented in section 4.1. This is followed by robustness results of how well the learned reward functions transferred to the modified environments in section 4.2. Section 4.3 shows the effects of reward function pre-training and section 4.4 details the impact of limited expert data on the different methods. The proposed shaped reward loss metric is investigated in section 4.5. At last, chapter 5 discusses the results and puts them into context with both previous work and potential future work.

2 | Prerequisites

For this thesis, a SAC reinforcement learning algorithm is combined with both GCL and AIRL. This chapter summarizes these methods and puts them into the wider context of research on (inverse) reinforcement learning. Section 2.1 gives an overview of RL and introduces important notations, as well as summarizing SAC in section 2.1.7. Given that the implemented algorithms can be seen as a special case of generative adversarial networks, section 2.2 briefly outlines this family of generative models. This is followed by section 2.3, providing a comprehensive review of important inverse reinforcement learning algorithms, culminating in a description of GCL in section 2.3.8 and AIRL in section 2.3.10. The relationship of both methods with GANs is detailed in section 2.3.9.

2.1 Reinforcement Learning

The reinforcement learning (RL) paradigm is a subfield of machine learning research, with the aim of solving problems based on a provided reward signal. In contrast to supervised learning, where examples for training are labeled, RL algorithms have to learn correct behavior solely from feedback on the actions they take. Instead of telling an agent what to do or how to do it, the agent has to explore possible actions and learn which policy maximizes the expected reward received from the environment [1]. Reinforcement learning has successfully been applied to complex tasks such as the game Go [16], Atari video games [17], and robotics [18], [19]. Some recent real-world applications of RL include recommender systems for content on websites [20] and optimizing memory controllers [21], [22].

2.1.1 The Reinforcement Learning Problem

Reinforcement learning is typically formalized as solving a Markov decision process (MDP), which is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, T, \gamma \rangle$ with the following components:

- \mathcal{S} is the state space.
- \mathcal{A} is the action space.

- \mathcal{R} is the reward function, depending on the context either mapping from states to rewards ($\mathcal{R} : \mathcal{S} \rightarrow \mathbb{R}$) or from state action pairs to rewards¹ ($\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$).
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ are the transition probabilities of reaching a state s' given a previous state s and an action a . This will often be denoted as $T(s'|s, a)$.
- γ is the discount factor for future rewards with which lower importance is attributed to rewards occurring later in an episode. It is used in the definition of state values V and action values Q below.

The decision process is Markovian since transitions and rewards only depend on the last state and not on the entire sequence of previous states.

A policy π maps from states to actions. Policies can either be stationary deterministic given a state ($\pi : \mathcal{S} \rightarrow \mathcal{A}$) or stochastic ($\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$), mapping from states to action probabilities. If not stated otherwise, we will assume the policy to be stochastic and denote the probability of picking action a in state s with $\pi(a|s)$.

$V^\pi : \mathcal{S} \rightarrow \mathbb{R}$ is the state value function that describes the expected return of being in state s and following policy π . Similarly, $Q^\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the action value function, corresponding to the expected return of performing action a in state s and then following policy π . They are defined by the so-called Bellman expectation equations in the following table 2.1. Their definitions depends on how the reward function \mathcal{R} of the MDP is defined.

	$\mathcal{R}(s)$	$\mathcal{R}(s, a)$
$V^\pi(s) =$	$\mathbb{E}_{a \sim \pi(a)}[Q^\pi(s, a)]$	$\mathbb{E}_{a \sim \pi(a)}[Q^\pi(s, a)]$
$Q^\pi(s, a) =$	$\mathbb{E}_{s' \sim T(s, a)}[\mathcal{R}(s') + \gamma V^\pi(s')]$	$\mathcal{R}(s, a) + \gamma \mathbb{E}_{s' \sim T(s' s, a)}[V^\pi(s')]$

Table 2.1: Bellman expectation equations for state and action values.

We can also define the optimal value functions for states and actions, V^* and Q^* . As before, they are recursively related and defined by the Bellman optimality equations:

	$\mathcal{R}(s)$	$\mathcal{R}(s, a)$
$V^*(s) =$	$\max_{a \in \mathcal{A}} [Q^\pi(s, a)]$	$\max_{a \in \mathcal{A}} [Q^\pi(s, a)]$
$Q^*(s, a) =$	$\mathbb{E}_{s' \sim T(s, a)}[\mathcal{R}(s') + \gamma V^*(s')]$	$\mathcal{R}(s, a) + \gamma \mathbb{E}_{s' \sim T(s' s, a)}[V^*(s')]$

Table 2.2: Bellman optimality equations for state and action values

¹A more general type of reward functions is $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$, such as defined in [1]. This is less frequently used in the literature however.

The optimal policy always picks an action with maximum Q-value, i.e. $\pi(a|s) = 1$ if and only if $a = \arg \max_{a' \in \mathcal{A}} Q^*(s, a')$ for stochastic policies and $\pi(s) = \arg \max_{a' \in \mathcal{A}} Q^*(s, a')$ for stationary deterministic policies.

We define the return J of a policy π as the expected sum of rewards for trajectories $\zeta = (s_1, a_1, s_2, a_2, \dots)$ where the actions are drawn from π :

$$J(\pi) = \mathbb{E}_{\zeta \sim \pi} [\mathcal{R}(\zeta)].$$

Extensions exist to MDPs such as the partially observable Markov decision process (POMDP) [23], which will not be covered in this thesis.

2.1.2 Value functions and Q-learning

The methods used in this thesis are based on the concept of temporal difference learning (TD-learning). This type of learning uses the error between predicted value and observed rewards from single transitions between states to update the Q -values. Starting from any random initialization of Q -values, this method is guaranteed to converge to the optimal ones under some assumptions (e.g. decreasing learning rate). The general update rule is

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$$

for a transition (s, a, r, s', a') where $r + \gamma Q(s', a') - Q(s, a)$ is the temporal difference error between the predicted value $Q(s, a)$ and the target value, which uses real information about the achieved reward r and the predicted future value $Q(s', a')$. As we can see from the Bellman expectation equations (table 2.1), this error is zero when correct values have been learned. The learning rate α defines the step size of each update. Learning Q -values in this way is called SARSA [24], [1] and is an on-policy method where updates are learned based on information sampled from the current policy.

An off-policy version of TD-learning is Q -learning. Here, the temporal difference error is modeled based on a greedy argmax-policy that might differ from the actual policy used during training (section 2.1.6 on exploration policies). This corresponds to the Bellman optimality equations (table 2.2) and yields a slightly different update rule

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a' \in \mathcal{A}} (Q(s', a')) - Q(s, a)).$$

This update rule can be used on stored transitions (s, a, r, s') and as an off-policy method it does not use information about the next action a' taken after s' . Q is often modeled as an artificial neural network. The use of the maximum operator instead of the policy action introduces a bias in the early stages of the learning process when values have not converged close to their true values yet. This bias can be alleviated by using double Q -learning: a second network is kept to estimate the values of the maximum value action selected by the first network. The networks Q_1 and Q_2 are not updated at the same time and thus can often avoid developing the same bias. In some versions of this algorithm, the second network is copied from the first network every fixed number of steps. The double Q -learning update rule is

$$Q_1(s, a) \leftarrow Q_1(s, a) + \alpha(r + \gamma Q_2(s', \arg \max_{a' \in \mathcal{A}} Q_1(s', a')) - Q_1(s, a)).$$

For MDPs with a small number of states, optimal V and Q -values can be computed with dynamic programming, a tabular method where one entry is kept in memory for each possible state or state action pair in \mathcal{S} and $\mathcal{S} \times \mathcal{A}$. For large or continuous environments, this quickly becomes infeasible from a computational point of view. This calls for the use of a more compact, parameterized function representation [1].

Deep neural networks can be applied as a non-tabular version of Q -learning. A deep Q-network (DQN) uses the temporal difference error defined above in its loss function and updates the parameters based on the gradient of this loss. This can be done on high-dimensional sensory input such as raw pixel data without the need of manually specifying features [17]. A double Q -learning version using neural networks has been proposed in [25].

Reinforcement learning agents based on Q -learning operate in the environment with an argmax-policy, picking the action with the highest value estimate in each state. In this way, the learned Q -values indirectly define the behavior of the agent.

2.1.3 Experience Replay

As described in [26], function approximation for temporal difference learning often suffers from correlations in a sequence of observations (i.e. nearby observations are often similar) and from the fact that even small changes in Q can change the policy and therefore the distribution of incoming data significantly.

For this reason, transitions (s, a, r, s') or (s, a, r, s', a') from the agent's interaction with the environment are often stored to a memory and re-used for training. When sampling transitions to update the Q -values (and indirectly the policy), transitions are drawn randomly, removing temporal correlation and yielding a more diverse set of training examples [13]. This procedure is called "experience replay".

Faster convergence to good results is often achieved by using prioritized experienced replay where transitions that led to a high error in the past are prioritized for learning over others [27].

2.1.4 Policy Optimization Methods

While Q -learning indirectly optimizes the policy by finding better estimates of action values and then picking actions with high Q -values, another family of reinforcement learning algorithms, called policy optimization or policy gradient algorithms, directly optimize the metric we care about: expected return of a policy $J(\pi)$.

The basic idea of these methods is to update the parameters of the policy, which is often a neural network, by performing gradient ascent on the current policy return [1]. The policy is a parameterized function on states which does not require any information on value functions. Value functions can still be used to learn the policy, but are not needed to select actions given a state.

One rather simple gradient of policy parameters ψ based on the return $J(\pi_\psi)$ was introduced in the REINFORCE algorithm [28]:

$$\begin{aligned}\nabla_\psi J(\pi_\psi) &= \mathbb{E}_{\zeta \sim \pi_\psi} \left[\sum_{a_t, s_t \in \zeta} \nabla_\psi \log (\pi_\psi(a_t|s_t)) \cdot \mathcal{R}(\zeta) \right] \\ &= \mathbb{E}_{\zeta \sim \pi_\psi} \left[\sum_{a_t, s_t \in \zeta} \frac{\nabla_\psi \pi_\psi(a_t|s_t)}{\pi_\psi(a_t|s_t)} \cdot \mathcal{R}(\zeta) \right].\end{aligned}$$

We can see above that the gradient of policy parameters is proportional to the trajectory reward sum $\mathcal{R}(\zeta)$, times the gradient of the parameters for taking a certain action, divided by the probability of taking this action (this avoids updating actions with high probability too strongly). This gradient can be used along with sampled trajectories to update the policy parameters.

Lower variance estimates of the gradient can be achieved by adding a baseline function over states (without any dependence on actions) to the expectation and by only considering rewards that occur after taking an action

$$\nabla_\psi J(\pi_\psi) = \mathbb{E}_{\zeta \sim \pi_\psi} \left[\sum_{a_t, s_t \in \zeta} \frac{\nabla_\psi \pi_\psi(a|s)}{\pi_\psi(a|s)} \cdot \sum_{t'=t}^T (\mathcal{R}(s_{t'}, a_{t'})) - V(s_t) \right],$$

where we use the state value function V as a baseline to reduce variance. This kind of gradient has empirically been shown to exhibit lower variance [1]. Intuitively, we make the gradient proportional to the relative difference between actions given a state and ignore if this state has very high or low value by subtracting the state value [28], [1]. For any action, we now only consider rewards that occur later in the trajectory, which further reduces variance.

2.1.5 Actor Critic Algorithms

Actor critic methods can be seen as a combination of temporal difference learning and direct policy optimization. As described in section 2.1.2, we can train a network to predict the Q -value of an action given a state. Instead of using the return over entire trajectories, we can then formulate policy parameter updates based on single steps taken by the policy π_ψ (the actor) in the environment, using the trained DQN Q_ϕ (the critic) [1]:

$$\nabla_\psi J(\pi_\psi) = \mathbb{E}_{(s,a) \sim \pi_\psi} \left[\frac{\nabla_\psi \pi_\psi(a|s)}{\pi_\psi(a|s)} \cdot Q_\phi(s, a) \right].$$

As before, we can use the state value function as a baseline for variance reduction, which makes the gradient of the actor parameters proportional to the advantage $A(s, a)$ of sampled actions

$$\begin{aligned}\nabla_\psi J(\pi_\psi) &= \mathbb{E}_{(s,a) \sim \pi_\psi} \left[\frac{\nabla_\psi \pi_\psi(a|s)}{\pi_\psi(a|s)} \cdot (Q_\phi(s, a) - V(s)) \right] \\ &= \mathbb{E}_{(s,a) \sim \pi_\psi} \left[\frac{\nabla_\psi \pi_\psi(a|s)}{\pi_\psi(a|s)} \cdot A(s, a) \right].\end{aligned}$$

Note that $A(s, a) = Q_\phi(s, a) - V(s) = r + \gamma V(s') - V(s)$, i.e. the advantage is equal to the temporal difference error for our V -function. Instead of doing Q -learning, we can learn a state value function V and use the temporal difference error for sampled transitions (s, a, r, s') to update both the actor and the critic. Actor critic methods come with the advantage of being easily applicable to problems with continuous action spaces where finding the argmax of all Q -values for a given state is very costly.

2.1.6 Exploration and Maximum Entropy RL

Reinforcement learning requires the agent to explore the environment sufficiently in order to allow it to learn an optimal policy. In fact, convergence proofs of reinforcement learning algorithms often require each state to be visited an infinite number of times [1].

One common way of exploring is to use an ϵ -greedy policy: the agent picks the best action by default, but with probability ϵ picks a random action uniformly. In the limit, this ensures that each action will be sampled an infinite number of times [1]. The exploration hyperparameter ϵ is often scheduled and decreased over the duration of training.

Another strategy to encourage exploration is to apply the principle of maximum entropy to the policy optimization and replace the common objective of maximizing expected return by a new one which also incentivizes high policy entropy $\mathcal{H}(\pi(\cdot|s))$

$$\begin{aligned} J_{\mathcal{H}}(\pi) &= J(\pi) + \beta \cdot \mathcal{H}(\pi) = \mathbb{E}_{\zeta \sim \pi} [\mathcal{R}(\zeta)] + \beta \cdot \mathbb{E}_{\zeta \sim \pi} \left[\sum_{s \in \zeta} \mathcal{H}(\pi(\cdot|s)) \right] \\ &= \mathbb{E}_{\zeta \sim \pi} \left[\sum_{s \in \zeta} \mathcal{R}(s) + \beta \cdot \mathcal{H}(\pi(\cdot|s)) \right] \\ &= \mathbb{E}_{\zeta \sim \pi} \left[\sum_{s \in \zeta} \mathcal{R}(s) + \beta \cdot \sum_{a \in \mathcal{A}} -\pi(a|s) \log (\pi(a|s)) \right]. \end{aligned}$$

The importance of the entropy is controlled with a temperature parameter β . Adding the entropy to the policy's objective can lead to improved exploration, since the policy is incentivized to distribute probability mass on several actions as long as it does not decrease performance strongly [29]². This becomes clearer when looking at the optimal policy for this objective

$$\pi_{\mathcal{H}}^*(a|s) = \exp \left[\frac{1}{\beta} (Q^{\text{soft}}(s, a) - V^{\text{soft}}(s)) \right],$$

where Q^{soft} is composed of the immediate reward plus the expected soft state value of the next state

$$Q^{\text{soft}}(s, a) = \mathcal{R}(s, a) + \gamma \mathbb{E}_{s' \sim T} [V^{\text{soft}}(s')]$$

and V^{soft} is a smooth (differentiable) approximation of the maximum operator

$$V^{\text{soft}}(s) = \beta \log \sum_a \exp \left[\frac{1}{\beta} Q^{\text{soft}}(s, a) \right].$$

²The beneficial effects of maximum entropy RL on exploration are also presented in this blog-post: <https://bair.berkeley.edu/blog/2017/10/06/soft-q-learning/>

Note that for $\beta \rightarrow 0$ the softmax operator in $V^{\text{soft}}(s)$ is equivalent to the maximum function.

Looking at the optimal maximum entropy policy $\pi_{\mathcal{H}}^*(a|s)$, we can see that actions with higher Q -value will be picked exponentially more often. However, if two actions in a state have very similar values, both of them will be picked with roughly the same probability. This allows for multi-modal behavior, which can be superior to greedily picking the best action and uniformly picking random actions with probability ϵ . The temperature β has to be tuned to trade-off between exploration and exploitation. This is illustrated in figure 2.1c.

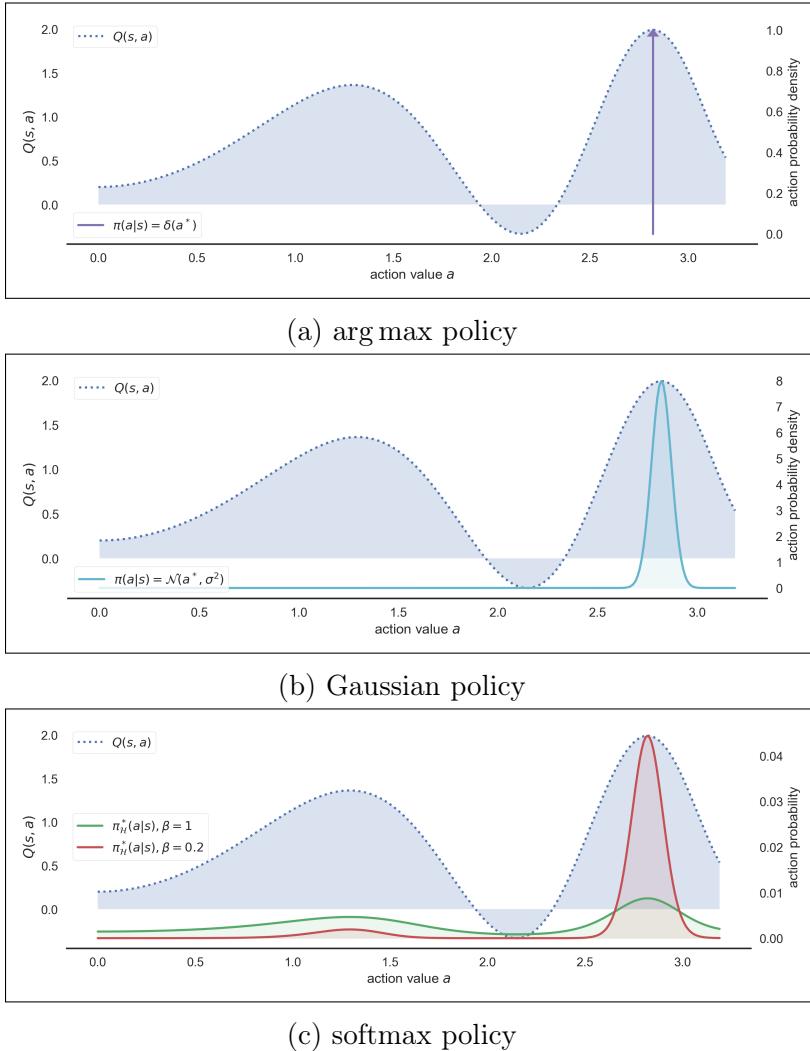


Figure 2.1: Comparison of policies for exploration. There is no exploration with the argmax policy in (a) which greedily picks the best action a^* . More exploration with Gaussian policy (b) that adds action noise in the vicinity of a^* . More exploration can be achieved by increasing variance σ^2 , but only in the neighborhood of the optimal action. The softmax policy in (c) can express multimodal behavior. Lower temperature β leads to more greedy behavior. Figure inspired by: <https://bair.berkeley.edu/blog/2017/10/06/soft-q-learning/>

2.1.7 Soft Actor Critic

Soft actor critic (SAC) [12] is an off-policy actor critic algorithm that makes use of maximum entropy reinforcement learning (section 2.1.6). It optimizes the policy w.r.t. the entropy-regularized return $J_{\mathcal{H}}$ which leads to slightly different formulations of state values V and action values Q :

$$Q_{\mathcal{H}}(s, a) = \mathcal{R}(s, a) + \gamma \mathbb{E}_{s' \sim T(s'|s, a)} [V_{\mathcal{H}}(s')],$$

$$V_{\mathcal{H}}(s) = \mathbb{E}_{a \sim \pi} [Q_{\mathcal{H}}(s, a)] + \beta \cdot \mathcal{H}(\pi(\cdot|s)).$$

There are three neural networks involved: one for state values V_{ψ} to approximate $V_{\mathcal{H}}(s)$ with parameters ψ , one for action values Q_{θ} to approximate $Q_{\mathcal{H}}(s, a)$ with parameters θ , and one for the policy π_{ϕ} with parameters ϕ . While in principle it is not required to model state and action values separately (as they depend on each other recursively), the authors of SAC claim that maintaining two separate networks improves stability of training [12].

The Q -value network can be trained by minimizing

$$L_Q(\theta) = \mathbb{E}_{(s, a, r, s')} \left[(Q_{\theta}(s, a) - (r + \gamma V_{\psi^-}(s')))^2 \right].$$

Instead of using the current state value network V_{ψ} , a second target network V_{ψ^-} is maintained with an exponential moving average of past ψ as parameters ψ^- , which improves training stability [17].

Similar to the Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm [30], two separate Q -networks, Q_{θ_1} and Q_{θ_2} , are used to avoid policy brittleness due to over-estimation of Q -values. The minimum estimate of both networks is used in the value function loss L_V and policy loss L_{π} (see below).

The value function is trained to minimize the squared residual error

$$L_V(\psi) = \mathbb{E}_s \left[(V_{\psi}(s) - \mathbb{E}_{a \sim \pi_{\phi}} [\min(Q_{\theta_1}(s, a), Q_{\theta_2}(s, a)) - \log(\pi_{\phi}(a|s))])^2 \right].$$

The policy is trained by minimizing the Kullback-Leibler divergence (KL divergence) between the current policy and the softmax policy based on the more pessimistic action value estimate³ from Q_{θ_1} and Q_{θ_2} .

$$L_{\pi}(\phi) = \mathbb{E}_s \left[D_{\text{KL}} \left(\pi_{\phi}(\cdot|s) \middle\| \frac{\exp [\min(Q_{\theta_1}(s, \cdot), Q_{\theta_2}(s, \cdot))]}{Z_{\theta}(s)} \right) \right].$$

To reduce variance, a reparameterization trick is used on the policy. For any given state, two parameters are deterministically computed: a mean $\mu_{\theta}(s)$ and a standard deviation $\sigma_{\theta}(s)$. The policy then draws from a Gaussian distribution (figure 2.1b) with these computed parameters [12]. The standard deviation σ^2 is learned according to the the maximum entropy objective $J_{\mathcal{H}}$, but it is not possible to express multimodal behavior (see figure 2.1).

SAC is an off-policy method, i.e. the parameters are updated based on transitions that might come from different (past) policies. We maintain an experience replay memory (section 2.1.3) and use randomly drawn transitions for training.

³The min operator is element-wise for all actions here.

2.2 Generative Adversarial Networks

The implemented methods of inverse reinforcement learning, which are described in sections 2.3.8 through 2.3.10, are analogous processes to a certain kind of generative adversarial network. This section will provide the necessary background on this type of generative models.

A generative adversarial network (GAN) [31] consists of two artificial neural networks, termed generator and discriminator, which are trained together in an adversarial way to create a generative model of some data distribution.

Provided an underlying true data distribution $p(x)$, the discriminator D is trained to correctly classify data as either coming from the true distribution p or from the generator G . The generator is trained to generate data which is similar enough to the true data distribution that it will be mis-classified by the discriminator. This becomes evident when looking at the loss functions of the respective networks and how the networks depend on each other.

The discriminator's loss is the expected log-probability assigned to the correct classification:

$$L_D(\theta) = \mathbb{E}_{x \sim p}[-\log D(x)] + \mathbb{E}_{x \sim G}[-\log(1 - D(x))],$$

where θ are the parameters of the discriminator.

The generator's loss could just be defined as the opposite of the discriminator's loss, but a more robust training signal is often achieved by also using the logarithm of the discriminators confusion [9]

$$L_G(\psi) = \mathbb{E}_{x \sim G}[-\log D(x)] + \mathbb{E}_{x \sim G}[\log(1 - D(x))],$$

where ψ are the parameters of the generator.

This structure of two networks “battling” against each other to reduce their respective losses and resulting in a strong generative model will arise again when looking at reinforcement policies agents as generative models and reward function estimates as part of the opposing discriminator, see section 2.3.9.

2.3 Inverse Reinforcement Learning

Especially in complex reinforcement learning tasks, there is often no natural source for the reward function. Instead, it has to be hand-crafted and shaped by humans to accurately represent the task while still being easily learnable [1]. This requires domain knowledge [32] and in practice often relies on an “informal trial-and-error search for a signal that produces acceptable results” [1].

A better way of finding a well-fitting reward function for some objective might be to observe a (human) expert performing the task, in order to learn corresponding rewards based on these observations [5]. This learned reward function can then be used by any reinforcement learning agent to optimize its policy. As opposed to reinforcement learning, where we are given a reward function \mathcal{R} and look for an

optimal policy π mapping from states to actions, we are now given some policy (or histories of state action pairs sampled from this policy) and look for a reward function for which the observed behavior is optimal or nearly optimal. Since this is effectively the opposite of reinforcement learning, this approach is called inverse reinforcement learning (IRL) [5].

In addition to the symbols already used in the previous section 2.1 on reinforcement learning, we introduce the following notation:

- $\hat{\mathcal{R}}$ is the estimated reward function which is the output of the IRL algorithm. The estimated reward function can be parameterized by some parameters θ , in this case it is written as $\hat{\mathcal{R}}_\theta$.
- ζ is a trajectory through the environment, an ordered list of states and actions $\langle (s_0, a_0), (s_1, a_1), \dots \rangle$.
- \mathcal{D} is the data set consisting of all trajectories ζ_1, ζ_2, \dots collected from the expert.
- π_E is the expert's policy while $\hat{\pi}$ is a policy behaving optimally for the reward estimate $\hat{\mathcal{R}}$.
- \hat{V}^{π_E} and \hat{Q}^{π_E} are the value functions for following the expert policy π_E while receiving the estimated reward $\hat{\mathcal{R}}$ instead of the real reward \mathcal{R} .
- \hat{V}^* and \hat{Q}^* are the state and action value functions which are optimal w.r.t. the current reward estimate $\hat{\mathcal{R}}$, i.e. $\hat{\pi}(a|s) = 1 \Leftrightarrow a = \arg \max_a \hat{Q}^*(s, a)$.

2.3.1 Motivation

The problem of inverse reinforcement learning was introduced⁴ by Stuart Russell in 1998 in an extended abstract [5] with two motivations: firstly, it can be used to learn and model objectives of other agents, especially humans and animals, and secondly it can be a way of learning a task based on expert examples in a robust, transferable, and succinct way [5], [6].

Modeling an agent's objectives

One important motivation of IRL is to learn about an agent's objectives. This can be important for research in psychology, neuroscience, economics, or biology, but also for AI agents that need to model friendly and adversarial agents in their environment.

Models of reinforcement learning are popular to describe operant conditioning and learning in animals and humans [34]. However, as [5] remarks upon, they generally assume a known and fixed reward function based on which behavior is optimized. It can be argued that we have to treat an agent's reward function as an empirical

⁴A similar problem setting was already mentioned in 1964 in the field of optimal control under the name *inverse optimal control* [33].

hypothesis that can potentially be falsified by observing the agent’s behavior. Simply assuming a known reward function *a priori* is especially questionable when dealing with multi-attribute functions where different features have to be combined and weighted. For this reason, IRL can be applied first to deduce a reward function from observed behavior [5].

IRL can also be used to create models of other agent’s motivations within an artificial intelligence system. These models can be of great use both for modeling adversarial agents or friendly agents aiming to solve a problem in cooperation with other robots or even humans. For example, in the cooperative inverse reinforcement learning (CIRL) setting, a robot tries to maximize an unknown human reward function in cooperation with the human [35].

Learning a task from experts

A good reward function is crucial for reinforcement learning algorithms to be successful. As Sutton and Barto note in their seminal book on reinforcement learning, “the success of a reinforcement learning application often strongly depends on how well the reward signal frames the problem and how well it assesses progress in solving it” [1]. When using a simplified or misspecified reward function as a proxy for the true one, RL-agents often discover surprising ways of exploiting the environment for this proxy reward, and the found solutions can be undesirable or even dangerous [1], [4]. The generation of a reward function is a commonly faced challenge in reinforcement learning [4] and some consider it to be the most difficult part of setting up an RL system [2] and a key bottleneck [3]. In many cases, finding a reward function limits the applicability of RL to real-world problems [32], especially if the problem is highly complex and requires a high level of autonomy [36].

One approach that might seem obvious at first sight is to simply provide a binary reward function with value 1 if the goal is achieved, e.g. when a game of chess is won. However, this often comes with the issue of the reward function not exhibiting enough variance for the robot to learn the desired behavior in acceptable training time [4]. Before reaching the desired goal state, the reward signal does not provide any information⁵. This leads to the necessity of shaping the reward [37] and conveying more information than a simple binary signal. Reward design is difficult since reward functions need to be easily learnable for RL agents.

Another aspect making the search for a good reward function difficult is that there are often several features within the environment that matter. The trade-off between them is often essential, but not necessarily obvious [4]. For example in the case of designing a reward function for a self-driving car, it is not necessarily clear how to weigh features such as safety, speed, or fuel consumption. There are other aspects that make reward design difficult, for example [38] found that in practice, when using neural networks for function approximation, multiplying reward functions by a scalar can strongly change the learned behavior.

⁵Imagine how hard it would be for humans to learn chess if their only feedback was to be informed whenever a game ends whether they won or lost. Finding out what the objective of chess is would take a large number of random trials.

This problem has been phrased as the optimal reward problem with the focus on finding reward functions for agents that are computationally bounded [39], [40]. This is a meta-optimization problem of finding reward functions that are both learnable for computationally bounded RL agents with limited resources for exploration and parametric learning, while still representing the desired behavior adequately.

If for the given problem an expert is available whose behavior can be observed, a special family of methods, called imitation learning, learning from demonstration, or apprenticeship learning, can be used to learn a good policy [1]. The expert demonstrations are used to avoid the optimal reward problem and manual specification of a reward signal.

There are two different types of these methods; the first directly learns a policy from given expert demonstrations with supervised learning, and the second is inverse reinforcement learning: learning a reward function which then is optimized to get a policy.

We start with looking at the supervised learning family of methods, which we will call *behavioral cloning*. In this case, we can take state action pairs of expert demonstrations as training data and train a model which takes states of the environment as input and predicts an action that the expert would have taken, using supervised machine learning. An early example of this method was ALVINN, an implementation of a self-driving car from 1989 which learned to imitate human steering behavior in differently curved roads using a simple artificial neural network [41]. Behavioral cloning has since been used for a variety of tasks [7].

While supervised imitation learning can be an effective choice especially for small problems [9], it comes with several drawbacks and issues. Most importantly, imitation learning relies on the learned policy generalizing sufficiently over the provided training data to produce robust behavior even in previously unseen situations. In more complex settings, policies obtained via imitation learning often exhibit brittleness when put in situations with slightly changed task dynamics (e.g. an airplane flying with slight wind turbulences) [8] [42]. In these complex tasks, imitation learning often results in ineffective or even dangerous behavior due to compounding errors [9].

Learning how to do a task by copying expert actions in known situations requires the environment in which the agent will be deployed to be highly similar to the one in which demonstrations were collected. Otherwise, the learned policy will fail as it is unable to properly generalize. The second type of imitation learning aims to avoid this problem of lacking robustness to changes in the environment. As Sutton and Barto summarized in [1], “the reward signal is [the] way of communicating to the robot what you want it to achieve, not how you want it achieved”. If a reward function can be learned from expert demonstrations, it can be used to train agents even in different environments. From this perspective, inverse reinforcement learning provides a “solution to the problem of generalization in imitation learning” [3].

To summarize, we want to use IRL to learn a definition of some task which can then robustly be learned by other agents with reinforcement learning algorithms. This definition is robust, since it does not rely on simply copying observed expert actions. In particular, it allows for suboptimality of the expert without necessarily “copying” the mistakes.

2.3.2 Difficulties of IRL

While inverse reinforcement learning brings the promise of learning a robust and transferable representation of a task, it also comes with difficulties.

In general, IRL is a heavily under-defined problem: for any given set of expert demonstrations, there are many (degenerate) reward functions which perfectly explain the behavior but are clearly not suitable solutions [6], e.g. the reward function that assigns zero value to all states and makes any behavior optimal. As was shown in [37], there is an equivalence class of reward transformations for which the optimal policy remains the same. This means that for any expert policy the solution set will have a large or even infinite number of reward functions that are consistent with expert behavior. Different approaches have been proposed for picking one particular reward function from the solution set, including principled methods such as maximum causal entropy IRL (section 2.3.7).

Algorithms for IRL generally require solving an entire MDP with RL in each iteration to update the reward function estimate, such as in [6], [43], [44], [45], [11], [46], [47], [48]. More modern algorithms re-use past policies and do not solve the MDP from scratch in each iteration [14], but they still require reinforcement learning algorithms. All difficulties of RL such as exploration and lacking robustness of results [38] are also present in IRL.

Another difficulty of IRL lies in the evaluation of obtained results, especially if no ground truth reward function is known. Since many different reward functions can lead to the same desired behavior, directly comparing the true and learned reward function is often not a reliable metric. Similarly, comparing the expert policy with the learned policy is not a good metric since even small differences in a few states can have a huge impact on the expected outcome, and the expert might make mistakes which we do not want copied. So far, the literature has not converged on one single metric to evaluate IRL methods, partially due to the variety of problems.

As shown in [49], it is impossible to deduce correct rewards from observed behavior without a model of the expert’s planning capabilities and rationality. Early IRL methods (implicitly) assumed the expert to always act optimally, later methods allowed for “soft” sub-optimality where better actions are chosen exponentially more often. However, in many domains human mental capabilities might be more limited than these assumptions and more realistic models of the experts planning algorithm will be needed.

Especially in problems with a large state space, the expert demonstrations might only cover a relatively small amount of possible states. This limited information makes it difficult to generalize the reward function to situations that were not seen during demonstrations [47].

2.3.3 Linear Programming Approaches

Arguably the first concrete methods for IRL were introduced in [6] for three different scenarios; small state spaces, large or infinite state spaces, and for cases where the behavior is only observed as example trajectories rather than the exact policy. We

will have a brief look at all three of them before continuing to more recent approaches since they provide valuable insights into what IRL entails and why it is difficult.

Small state space and known policy

The first method from [6] is suitable for small state spaces where the expert's policy π_E is stationary deterministic, fully known and perfectly optimal w.r.t. the true reward function $\mathcal{R} : \mathcal{S} \rightarrow \mathbb{R}$.

The inverse reinforcement learning problem requires the observed policy to be optimal under the estimated reward and thus also for the corresponding state values

$$\forall s : \pi_E(s) \in \arg \max_a \sum_{s' \in \mathcal{S}} T(s'|s, a) \hat{V}^{\pi_E}(s'), \quad (2.1)$$

where the sum of $T(s'|s, a) \hat{V}^{\pi_E}(s')$ is the Q -value of being in state s and taking action a . This is equivalent to

$$\forall s, \forall a : \sum_{s' \in \mathcal{S}} T(s'|s, \pi_E(s)) \hat{V}^{\pi_E}(s') \geq \sum_{s' \in \mathcal{S}} T(s'|s, a) \hat{V}^{\pi_E}(s'). \quad (2.2)$$

The method proposed in [6] for the setting described above is based on linear programming and can succinctly be formulated in matrix and vector notation. Let \mathbf{T}^{π_E} be the policy transition matrix of size $n \times n$ containing state transition probabilities for always choosing $\pi(s)$ in all n states

$$\mathbf{T}_{s,s'}^{\pi_E} = T(s'|s, \pi_E(s)).$$

For each state, there are $k - 1$ actions that are not picked by the policy ($a \neq \pi_E(s)$). We can order these actions arbitrarily for each state and construct $k - 1$ non-policy transition matrices $\mathbf{T}^{\neg\pi_E} = \{\mathbf{T}^1, \dots, \mathbf{T}^{k-1}\}$. $\mathbf{T}^i \in \mathbf{T}^{\neg\pi_E}$ is a $n \times n$ transition matrix containing state transition probabilities for always choosing the i -th non-policy action in any state. The estimated state values \hat{V}^{π_E} and estimated rewards $\hat{R}(s)$ can be expressed as a vectors $\hat{\mathbf{V}}^{\pi_E}$ and $\hat{\mathbf{R}}$ with n elements. Let \succeq denote vectorial inequality, i.e. $\mathbf{x} \succeq \mathbf{y}$ if and only if $\forall i : \mathbf{x}_i \geq \mathbf{y}_i$. Then we can express inequality (2.2) in matrix notation as

$$\forall \mathbf{T}^i \in \mathbf{T}^{\neg\pi_E} : \mathbf{T}^{\pi_E} \hat{\mathbf{V}}^{\pi_E} \succeq \mathbf{T}^i \hat{\mathbf{V}}^{\pi_E}. \quad (2.3)$$

Since

$$\hat{\mathbf{V}}^{\pi_E} = \hat{\mathbf{R}} + \gamma \mathbf{T}^{\pi_E} \hat{\mathbf{V}}^{\pi_E} \Leftrightarrow \hat{\mathbf{V}}^{\pi_E} = (\mathbf{I}_n - \gamma \mathbf{T}^{\pi_E})^{-1} \hat{\mathbf{R}},$$

with \mathbf{I}_n being the $n \times n$ identity matrix, we can simplify the solution set defined in (2.3) to

$$\forall \mathbf{T}^i \in \mathbf{T}^{\neg\pi_E} : (\mathbf{T}^{\pi_E} - \mathbf{T}^i)(\mathbf{I}_n - \gamma \mathbf{T}^{\pi_E})^{-1} \hat{\mathbf{R}} \succeq 0. \quad (2.4)$$

The definition of the solution set in equation (2.4) provides $(k - 1) \cdot n$ linear inequalities with n unknowns which can be fed to any linear programming solver to find a solution. However, it is obvious that the solution of $\hat{\mathbf{R}} = 0$, as well as all

other constant reward functions and many other degenerate solutions, are part of the solution set, many of them not at all similar to the true reward function \mathcal{R} . Thus, the problem is under-defined. Another big drawback of this formulation is that the solution set can become empty when the observations contain sub-optimal, inconsistent examples.

In order to remove degenerate solutions from the solution set and to pick one that is reasonable good, two heuristics are proposed in [6] that can easily be added to the linear programming problem.

The first heuristic aims to maximize the distance between the value of the policy action and second-best action for each state:

$$\text{maximize: } \sum_{s \in \mathcal{S}} \hat{Q}^{\pi_E}(s, \pi_E(s)) - \max_{a \in \mathcal{A} \setminus \{\pi_E(s)\}} \hat{Q}^{\pi_E}(s, a). \quad (2.5)$$

While heuristic (2.5) will remove many degenerate solutions such as the reward function assigning zero reward to all states, it also makes quite a strong assumption of often having a large difference between the values of the best and second-best action. In section 2.3.6 we will see a more principled way of selecting one reward function without using this strong assumption.

As a second heuristic, [6] introduces a penalty on large rewards, similar to weight decay regularization [50] in other machine learning techniques. For this, one can use the L1-norm or L2-norm, weighted by some hyperparameter λ , and modify the maximization term to

$$\text{maximize: } -\lambda \|\hat{\mathbf{R}}\|_1 + \sum_{s \in \mathcal{S}} \hat{Q}^{\pi_E}(s, \pi_E(s)) - \max_{a \in \mathcal{A} \setminus \{\pi_E(s)\}} \hat{Q}^{\pi_E}(s, a). \quad (2.6)$$

In addition to the constraints from (2.4), an upper bound for the values of the reward function is enforced:

$$\text{s. t. } \|\hat{\mathbf{R}}\|_\infty \leq R_{\max}. \quad (2.7)$$

Large or infinite state space and known policy

The previous method runs into problems of tractability when applied to larger problems, especially with large state spaces. This is both due to the required matrix inversion becoming too costly and the set of constraints for the linear solver becoming very large.

Instead, the second method proposed in [6] resorts to function approximation and expresses the estimated reward function as a linear combination of d fixed, known, and bounded basis functions⁶ $\phi_i : \mathcal{S} \rightarrow \mathbb{R}$:

$$\hat{\mathcal{R}}_\theta(s) = \hat{\theta}_1 \cdot \phi_1(s) + \dots + \hat{\theta}_d \cdot \phi_d(s), \quad (2.8)$$

where $\hat{\theta}_i$ are the estimated coefficients of the reward function. The state value function \hat{V}^{π_E} can be expressed in terms of these basis functions. We can compute

⁶These basis functions are often called *feature functions* as they map from the large or high-dimensional state space to features for which the reward function is a linear combination.

the value function for each basis function, $\hat{V}_i^{\pi_E}$, pretending that $\hat{\mathcal{R}}_\theta(s) = \phi_i$. By the linearity of expectations, the value with the full estimated reward function will then be

$$\hat{V}^{\pi_E} = \hat{\theta}_1 \hat{V}_1^{\pi_E} + \dots + \hat{\theta}_d \hat{V}_d^{\pi_E}.$$

As the state space is very large or even infinite in this scenario, the solution set cannot be expressed with matrices as we did in the previous approach. Instead a subset of states $\mathcal{S}_0 \subset \mathcal{S}$ is sampled to construct constraints enforcing that the value of policy actions for these sampled states be equal to or higher than the value of non-policy actions

$$\forall s \in \mathcal{S}_0, \forall a \in \mathcal{A} \setminus \{\pi(s)\} : \mathbb{E}_{T(s'|s, \pi_E(s))} [\hat{V}^{\pi_E}(s')] \geq \mathbb{E}_{T(s'|s, a)} [\hat{V}^{\pi_E}(s')]. \quad (2.9)$$

Since we know that \hat{V}^{π_E} is linear in the parameters $\hat{\theta}_i$, (2.9) provides a set of linear equations based on which we can estimate fitting parameter values. However, the real reward function might not be representable as a linear combination of our selected basis functions ϕ_i . This could lead to our solution set being empty as soon as two or more constraints are not fulfillable with the choice of any values for the parameters $\hat{\theta}_i$. To resolve this issue, we relax our linear constraints and simply penalize whenever they are violated. For this we introduce some penalization function p given by $p(x) = x$ if $x \geq 0$, and $p(x) = c \cdot x$ otherwise. The hyperparameter c defines how much constraint violations should be penalized.

Similarly to the distance maximization heuristic (2.5) before, the linear programming solver is asked to maximize the distance between the policy action value and the second best action value. In addition, the reward parameters $\hat{\theta}_i$ are forced to lie in the interval $[-1, 1]$. The formulation of the problem then becomes

$$\begin{aligned} & \text{maximize} \sum_{s \in \mathcal{S}_0} \min_{a \in \mathcal{A} \setminus \{\pi(s)\}} p\left(\mathbb{E}_{T(s'|s, \pi_E(s))} [\hat{V}^{\pi_E}(s')] - \mathbb{E}_{T(s'|s, a)} [\hat{V}^{\pi_E}(s')]\right), \\ & \text{s. t. } |\hat{\theta}_i| \leq 1, i = 1, \dots, d. \end{aligned}$$

Gaussian basis functions which are spaced evenly over the state space were used for the experiments in [6].

Based on example trajectories

In many real world scenarios, the expert's policy π_E is not fully known. Instead, behavior is observed as expert trajectories $\zeta_i \in \mathcal{D}$ consisting of alternating observations of states and actions. The third method proposed in [6] deals with this scenario.

As in the previous algorithm, we will use a linear combination of known, fixed, and bounded basis functions ϕ_i to approximate the true reward function. However, now we want to calculate the empirical value of a trajectory $\hat{V}(\zeta)$, not the expected value of a policy.

To do this, we compute the corresponding future discounted value estimate $\hat{V}_i(\zeta)$ for each basis function

$$\hat{V}_i(\zeta) = \sum_{s_j \in \zeta} \gamma^j \phi_i(s_j).$$

Since we defined $\hat{\mathcal{R}}_\theta$ to be a linear combination of the basis function, the overall empirical value of a trajectory is

$$\hat{V}(\zeta) = \hat{\theta}_1 \hat{V}_1(\zeta) + \dots + \hat{\theta}_d \hat{V}_d(\zeta).$$

Our goal is to find reward parameters θ_i such that the mean trajectory values of all expert demonstrations $\zeta_i \in \mathcal{D}$ is greater than than the mean trajectory value of other trajectories \mathcal{D}_s , sampled from some other policy π_o

$$\mathbb{E}_{\zeta_i \in \mathcal{D}} [\hat{V}(\zeta_i)] \geq \mathbb{E}_{\zeta_i \in \mathcal{D}_s} [\hat{V}(\zeta_i)] \quad (2.10)$$

Based on the inequality (2.10) we can now use linear programming as in the approach before to find fitting parameters $\hat{\theta}_i$. However, comparing the expert trajectories with only one arbitrary other policy π_o is likely to be insufficient. Optimally, we want to compare the expert trajectories with many more trajectories and especially with trajectories that are more “competitive” than random policies. This is where the inductive step of the method comes in. After each new estimation of the reward function, we run an RL algorithm to find a policy that is optimal for the current reward estimate. This policy will most likely be better than the expert trajectories for the current reward estimate and if we use it for additional constraints it drives the estimate closer to the actual reward function.

For each new policy π_i generated this way, we will generate corresponding trajectories. This algorithm then runs for some large number of iterations until a satisfactory reward function estimate is found. For this algorithm to work we have to make two important assumptions: Firstly, given any reward function, we can generate a policy that is optimal for this reward function. Secondly, trajectories can be generated for policies obtained in this way, e.g. via a simulator.

As in the approach before, we might run into cases where the true reward function cannot be expressed as a linear combination of the fixed basis functions. To still find a solution that is at least close to the real one, we again use the penalization function p , which penalizes violated constraints more heavily. Also, just as in the algorithm before, we limit the maximum value of the parameters $\hat{\theta}_i$.

2.3.4 Apprenticeship Learning via IRL

Another important early approach to the IRL problem was introduced in [43] for the apprenticeship learning setting, which is the task of learning behavior from an expert. Similarly to the previous two methods, the assumption of this paper is that there is some function $\phi : \mathcal{S} \rightarrow [0, 1]^k$ mapping from states to k -dimensional feature vectors such that the true reward function \mathcal{R} can be expressed as a linear combination of the features and a vector of reward parameters θ : $\mathcal{R} = \theta \cdot \phi(s)$. The assumption of $\|\theta\|_1 \leq 1$ is added to ensure that rewards are bounded by 1.

We define the feature expectation of a policy π as

$$\mu(\pi) = \mathbb{E}_{s \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t \phi(s_t) \right].$$

Since we assume the reward function to be a linear combination of the features ϕ , the expected return of a policy is $\theta \cdot \mu(\pi)$. The expert's empirical feature expectation μ_E is calculated based on all observed trajectories in the dataset

$$\mu_E = \frac{1}{|\mathcal{D}|} \sum_{\zeta \in \mathcal{D}} \sum_{s \in \zeta} \gamma^t \phi(s) \approx \mu(\pi_E).$$

The approach proposed in [43] is based on finding reward function parameters $\hat{\theta}$ which lead to a policy $\hat{\pi}$ with very similar feature counts as the expert's behavior

$$|\mu_E - \mu(\hat{\pi})| \leq \epsilon,$$

which, given that the true reward function is assumed to be a linear combination of the features ϕ , guarantees an expected true return of the learned policy which is close to the expert's return:

$$\begin{aligned} & \left| \mathbb{E}_{s_t \sim \pi_E} \left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t) \right] - \mathbb{E}_{s_t \sim \hat{\pi}} \left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t) \right] \right| \\ & \leq \|\theta\|_2 \cdot \|\mu_E - \mu(\hat{\pi})\|_2 \\ & \leq 1 \cdot \epsilon = \epsilon, \end{aligned}$$

since $\|\theta\|_2 \leq \|\theta\|_1$ and $\|\theta\|_1$ is assumed to be no larger than 1.

The estimated reward parameters θ are found with an algorithm that is essentially based on a simple linear support vector machine (SVM). The positive class only contains a single point μ_E , labeled with +1. The negative class is iteratively constructed and contains feature expectation vectors $\mu(\hat{\pi}_i)$ for new policies $\hat{\pi}_i$ which are optimal to the respective reward estimate at the current step. After new points are added to the negative class, the reward parameters $\hat{\theta}$ are updated using the SVM. Under the assumptions of the paper, $\hat{\theta}$ is the normal vector of the maximal margin separating hyperplane. The intuition here is that newly found policies will be optimal for the current reward estimate and will thus force the reward estimate to update closer to the real one in order to keep the newly found feature expectation in the negative class. Since μ_E is assumed to be optimal behavior for the real reward function, the problem will always be linearly separable (except for the exceptional but desirable case of $\mu(\hat{\pi}_i) = \mu_E$).

Overall this methods strongly resembles the linear methods described in section 2.3.3 from [6]. The biggest difference is that instead of using a linear solver, the well-established methods of SVM are applied.

A second, projection based method, which is similar to the one describe above, was proposed in the same paper as a faster procedure.

2.3.5 Bayesian IRL

Earlier approaches such as [6] or [43] come with two limitations: firstly, they rely on the assumption of the expert demonstrations being optimal, and secondly they rely on heuristics to pick one candidate from the solution set to avoid ending up

with a degenerate reward function. These heuristics induce a bias of which reward function will be selected as the best estimate. The Bayesian inverse reinforcement learning (BIRL) approach [45] aims to avoid these two drawbacks by framing the IRL problem as a problem of Bayesian inference. Instead of heuristically picking one single reward function that fits to observed behavior, this approach estimates the posterior probability distribution over all possible reward functions given expert demonstrations and a prior over reward functions. In this way the uncertainty about which reward function is the correct one can be modeled.

We can apply Bayes theorem to model the posterior probability $Pr(\hat{\mathcal{R}}|\mathcal{D})$ of a reward function given observed trajectories \mathcal{D} as evidence and a prior over reward functions $Pr(\hat{\mathcal{R}})$:

$$Pr(\hat{\mathcal{R}}|\mathcal{D}) = \frac{Pr(\mathcal{D}|\hat{\mathcal{R}}) \cdot Pr(\hat{\mathcal{R}})}{Pr(\mathcal{D})}. \quad (2.11)$$

The numerator in equation (2.11) consists of two parts: the likelihood $Pr(\mathcal{D}|\hat{\mathcal{R}})$ of observing trajectories \mathcal{D} given a certain reward $\hat{\mathcal{R}}$, and the prior over reward functions $Pr(\hat{\mathcal{R}})$.

The likelihood $Pr(\mathcal{D}|\hat{\mathcal{R}})$ is based on a model of how the observed agent makes choices given a reward function. A fully rational and optimal agent would always pick the action which corresponds to the highest expected future return

$$\arg \max_a \hat{Q}^*(s, a).$$

However, [45] aims to allow imperfect agents that sometimes pick suboptimal actions. In this way BIRL avoids the problem of the methods described in the previous subsection where the solution set can become empty if the expert is not fully optimal. The likelihood of picking an action should increase with the action's \hat{Q}^* -value. The likelihood of picking an action with high \hat{Q}^* -value should also depend on the agent's capability of acting rationally, which they model with a parameter α where high α corresponds to very rational agents. The entire likelihood of picking action a in state s can then be modeled by a Boltzmann distribution with energy $-\hat{Q}^*(s, a)$:

$$Pr(a|s, \hat{\mathcal{R}}) = \frac{\exp [\alpha \cdot \hat{Q}^*(s, a)]}{Z}. \quad (2.12)$$

The likelihood of observing a data set \mathcal{D} of trajectories with state-action pairs then is

$$Pr(\mathcal{D}|\hat{\mathcal{R}}) = \frac{\exp [\alpha \cdot (\sum_{\zeta \in \mathcal{D}} \sum_{(s,a) \in \zeta} \hat{Q}^*(s, a))] }{Z}. \quad (2.13)$$

The prior over reward functions $Pr(\hat{\mathcal{R}})$ can be used to encode prior and external information about the model, or for combining evidence from multiple experts. If not specified otherwise, the rewards of different states are assumed to be independently identically distributed (i.i.d.). There are different kinds of priors proposed in [45], among them:

- A **uniform** prior if one is completely agnostic about the reward function.

- A **Gaussian** or **Laplacian** prior with zero mean if one suspects most states have negligible rewards.
- A **beta distribution** for scenarios where most states have low rewards and only a few states have high rewards.

The normalization term $Pr(\mathcal{D})$ in the denominator of equation 2.11 is in most cases intractable to compute. Instead, [45] make use of a Markov chain Monte Carlo (MCMC) method to approximate the posterior distribution.

There are two possible scenarios for which we might want to use IRL: reward estimation or apprenticeship learning. The former is the inferential problem of finding a reward function estimate $\hat{\mathcal{R}}$ that is as close as possible to the original \mathcal{R} by minimizing a loss function, e.g. the L2 loss⁷, for the vectorized reward function estimate $\hat{\mathbf{R}}$ and vectorized ground truth \mathbf{R} ,

$$\|\mathbf{R} - \hat{\mathbf{R}}\|_2.$$

The latter problem of apprenticeship learning is about learning a policy $\hat{\pi}$ from expert demonstrations with the objective of performing as well as possible under the true reward function. A loss can be introduced judging how close the learned policy is to the optimal one.

Both reward learning and apprenticeship learning require $\mathbb{E}_{Pr(\hat{\mathcal{R}}|\mathcal{D})}[\hat{\mathcal{R}}]$, the mean of the posterior distribution (2.11) over reward functions, to find optimal solutions, as is shown in [45].

The mean of the posterior is estimated via a MCMC method called policy walk in order to avoid the expensive computation of the denominator $Pr(\mathcal{D})$. This method works as follows⁸:

1. Pick a random reward vector as initial reward estimate vector $\hat{\mathbf{R}}$.
2. Compute optimal policy and action values $\hat{\pi}$, \hat{Q}^* w.r.t $\hat{\mathbf{R}}$.
3. Repeat n times:
 - (a) Given current reward function sample $\hat{\mathbf{R}}$, sample new proposal $\hat{\mathbf{R}}_{\text{prop}}$ from neighbors of $\hat{\mathbf{R}}$ on a grid of length δ . $\hat{\mathbf{R}} = \hat{\mathbf{R}}_{\text{prop}}$ except for one element corresponding to some state s where $\hat{\mathbf{R}}(s) = \hat{\mathbf{R}}_{\text{prop}}(s) \pm \delta$.
 - (b) Compute $\hat{\pi}_{\text{prop}}$, \hat{Q}_{prop}^* (e.g. with policy iteration initialized with $\hat{\pi}$).

⁷Using a loss function directly on reward functions can be a bad metric as for each function there is a large equivalence class of reward functions leading to the same behavior. Equivalent functions can have large distances. More on this in section 3.6. BIRL restricts the class of allowed reward functions to partially avoid this problem.

⁸The description of the policy walk algorithm differs slightly from the original pseudocode in [45]. This is mostly due to their definition of $P(\mathbf{R}, \pi)$ not being entirely clear. In subsequent papers on Bayesian inverse reinforcement learning such as [47] and [51] the posterior distribution (or likelihood combined with uniform prior) is used in this to compute the acceptance probability. We adopted this interpretation of how the policy walk works.

- (c) With probability $\min(1, \frac{Pr(\hat{\mathbf{R}}_{\text{prop}}|\mathcal{D})}{Pr(\hat{\mathbf{R}}|\mathcal{D})})$:
Set $\hat{\mathbf{R}} := \hat{\mathbf{R}}_{\text{prop}}$, $\hat{\pi} := \hat{\pi}_{\text{prop}}$, and $\hat{Q}^* := \hat{Q}_{\text{prop}}^*$

4. Return $\hat{\mathbf{R}}$

The policy walk algorithm described above requires computing $\hat{\pi}_{\text{prop}}$ and \hat{Q}_{prop}^* in each iteration of the sampling process. While previous results of Q -values and policies can be used as initial guesses in each iteration to get faster convergence to new results, this is still a very costly process. In [45] it is shown that in the special case of a uniform prior the Markov chain is rapidly mixing, requiring a polynomially bounded number of steps in the policy walk algorithm. Each policy walk yields one reward function sample from the posterior distribution. In order to perform reward inference or apprenticeship learning, the mean of several reward function samples is computed.

For numerical stability in the implementation of the acceptance probability in part (3.c) above, one can simplify

$$\begin{aligned} \frac{Pr(\hat{\mathbf{R}}_{\text{prop}}|\mathcal{D})}{Pr(\hat{\mathbf{R}}|\mathcal{D})} &= \frac{Pr(\mathcal{D}|\hat{\mathbf{R}}_{\text{prop}})}{Pr(\mathcal{D}|\hat{\mathbf{R}})} = \frac{\exp[\alpha \cdot (\sum_{\zeta \in \mathcal{D}} \sum_{(s,a) \in \zeta} \hat{Q}_{\text{prop}}^*(s,a))]}{\exp[\alpha \cdot (\sum_{\zeta \in \mathcal{D}} \sum_{(s,a) \in \zeta} \hat{Q}^*(s,a))]} \cdot \frac{Z}{Z_{\text{prop}}} \\ &= \exp[\alpha \cdot (\sum_{\zeta \in \mathcal{D}} \sum_{(s,a) \in \zeta} \hat{Q}_{\text{prop}}^*(s,a) - \hat{Q}^*(s,a))] \cdot \frac{Z}{Z_{\text{prop}}} \end{aligned}$$

MAP BIRL

The posterior mean is a typical choice for BIRL, since it can be shown to minimize the mean squared loss L_2 between true reward and estimated reward. However, an example case is pointed out in [52] in which this posterior mean can result in sub-optimal policies w.r.t the true reward while a maximum a posteriori (MAP) estimate yields a reward estimate closer to the ground truth. This is due to the posterior mean integrating over the entire reward space and the possibility of the mean ending up in a region with very low posterior probability.

Many non-Bayesian IRL methods can be framed as MAP search problem where the regularization terms are encoded into the prior and the compatibility score between the reward estimate and demonstrated behavior is encoded into the likelihood function [52].

Improved BIRL

As stated in [47], the BIRL approach to IRL suffers from several inefficiencies, even for moderate problem sizes. For this reason, [47] proposes modifications to the original algorithm in order to increase tractability for larger problems, especially when the expert demonstrations do not cover the full state space.

The reward function in [45] is modeled as a vector with n elements where n is the size of the state space. A large number of iterations is needed for the policy walk algorithm to mix and output a good sample. Since each iteration requires solving an MDP, the algorithm quickly becomes impractical with increasing problem size.

Another issue arises when expert demonstrations do not provide information about parts of the state space. As was observed empirically in [47], reward estimates of states far away from expert demonstrations tend to “wander” around which slows down the convergence to good samples. They argue that it is “naive” to perform inference over states that lie far away from any expert demonstrations.

To overcome the two issues mentioned before, [47] propose two changes to the original BIRL algorithm:

Kernel-based relevance functions are used to focus the policy walk proposals to add δ mostly in states that are similar to ones observed in expert demonstrations. For this, a feature function $\phi : \mathcal{S} \rightarrow \mathbb{R}^d$ is introduced to build feature vectors for each state which can be given to a kernel function k (e.g. the dot product) to judge the pairwise similarity of states. We can define the state relevance function ρ that serves as a measure of how similar some state is to the set of states given by expert demonstrations

$$\rho(s) = \frac{\sum_{s' \in \mathcal{D}} k(\phi(s), \phi(s'))}{Z}, \quad (2.14)$$

with

$$Z = \max_{s \in \mathcal{S}} \sum_{s' \in \mathcal{D}} k(\phi(s), \phi(s')).$$

When constructing $\hat{\mathbf{R}}_{\text{prop}}$ during policy walk, the probability of the state for which the reward gets changed is chosen proportionally to $\rho(s)$. In this way, rewards for states that are far away from expert demonstrations get updated less frequently and stay closer to the prior. Some states might never get picked by the algorithm to adapt the reward value. Because of this it is important that the initial reward estimate $\hat{\mathbf{R}}$ gets initialized meaningfully, e.g. to the maximum of the prior so that rewards for rarely updated states have reasonable default values.

Simulated annealing is proposed as a way to introduce an explicit exploration-exploitation trade-off. The acceptance probability is adapted with a decreasing temperature T_i at iteration i according to some temperature schedule. An alternative interpretation of mean estimation via MCMC is used in [47] where the output of one single policy walk is said to correspond to the mean⁹. Instead of aggregating several samples from several policy walks into one mean value, they use one single policy walk where accepting a reward estimate proposal which is worse than the current estimate becomes less and less likely over time. This leads to early iterations leaning more towards exploration while later iterations will put more focus on maximizing the posterior. One might suspect that this converges to some local

⁹In the original BIRL paper [45], the policy walk algorithm is used to find a single sample from the posterior distribution. Several samples are generated in order to aggregate to the sample mean which estimates the true mean of the distribution. However, in [47] only one run of the policy walk is performed and the single obtained sample is returned, claiming that this sample is a good estimate of the posterior mean.

maximum of the posterior distribution instead of the mean, in many cases this might be equivalent to MAP search.

Robust BIRL

Another modification to the classical BIRL algorithm is introduced in [53] with a more elaborate model of the expert’s choice behavior in the likelihood term. They call their approach robust BIRL in the sense of being robust to suboptimal expert demonstrations. As opposed to the softmax rationality model (2.12) used in [45], where each observation is assumed to have the same reliability, this paper models experts as picking actions with sparse noise. With sparse noise, most actions are assumed to be optimal, while only a few, sparse observations come from sub-optimal decisions. This is modeled by latent variables $\alpha_1, \alpha_2, \dots, \alpha_M$ for each of the M observations (s, a) in the dataset. Each α_i has a prior $P(\alpha)$ which is constructed based on a Laplace distribution such that most α_i will have a value close to one, while only a few will be close to zero. In this way, most observations will be assumed to be near optimal, while a few can be considered as outliers with low reliability. In this way, the inference of the true reward function becomes more robust to a compromised data set. In order to infer both the reward function $\hat{\mathcal{R}}$ and the latent variables $\alpha_1, \alpha_2, \dots, \alpha_M$, an expectation maximization (EM) algorithm is used.

2.3.6 Maximum Entropy IRL

Instead of modeling the distribution of reward functions via Bayesian inference as in section 2.3.5, the approach called maximum entropy IRL introduced in [11] offers another way of reducing the solution set to one sensible reward function candidate in a principled way¹⁰. As in the Bayesian approaches before, maximum entropy methods can deal both with sub-optimal or noisy example trajectories and do not use biased maximum margin heuristics to pick one of the many possible reward functions.

The principle applied in this approach is the one of maximum entropy, introduced in 1957 by Jaynes [10]. It was originally proposed for the domain of statistical mechanics but can be seen as a general principle for any scientific domain when constructing probability distributions based on partial information. Proscribing epistemic modesty, the principle states that from all possible distributions that conform with given partial information, one should pick the one that commits the least beyond this given information. The distribution that should be picked is the one that is least predictable, i.e. the one with maximum entropy. This principle yields the least biased estimate. The maximum entropy distribution always assigns non-zero probability to all possibilities that were not explicitly excluded by the given information.

As already proposed in [43], a reward function estimate that leads to policies with true return similar to the expert’s return can be found by matching the features of the expert demonstrations with features of policies based on the current reward

¹⁰As we will see in section 2.3.7, this leads to a probabilistic model which is essentially equivalent to finding a maximum of the Bayesian posterior.

estimate. When trying to match the features, it is often the case that many different policies lead to the same feature expectation. In [43] this problem is not really addressed and the first found reward estimate and policy with sufficiently matching feature counts are returned as a solution.

The IRL approach proposed in [11] applies the principle of maximum entropy to pick a distribution over possible trajectories through the MDP that does not commit to any assumptions besides the trajectories' feature counts matching the empirical expert feature counts¹¹.

For deterministic MDPs, the resulting maximum entropy distribution is parameterized by the reward parameters θ (assuming that \mathcal{R} is again a linear combination of θ and state features $\phi(s)$):

$$Pr(\zeta|\theta) \propto \exp [\theta \cdot \phi(\zeta)] = \exp \left[\sum_{s \in \zeta} \theta \cdot \phi(s) \right] = \exp \left[\sum_{s \in \zeta} \mathcal{R}(s) \right].$$

This distribution over trajectories makes paths with high reward exponentially more likely while not overly committing to any specific one.

In the case of stochastic MDPs, the randomness of transition dynamics between states has to be taken into account. In [11] an approximation of the trajectory likelihood is proposed:

$$Pr(\zeta|\theta, T) \approx \frac{1}{Z(\theta, T)} \exp [\theta \cdot \phi(\zeta)] \prod_{s_t, a_t, s_{t+1} \in \zeta} T(s_{t+1}|s_t, a_t). \quad (2.15)$$

This approximation is meant for scenarios where “randomness has limited effect on behavior” [11] since it does not adequately reflect the uncertainty of an agent when picking a stochastic action and ignores causality. This approximation was later replaced by the author with a formulation that adequately takes transition probabilities and causality into account [46], as described in section 2.3.7.

The objective in maximum entropy IRL is to maximize the (log-)likelihood of observed expert demonstrations while modeling the expert behavior with a maximum entropy distribution. The reward parameters corresponding to the maximum likelihood then provide the estimate for the true reward function

$$\hat{\theta} = \arg \max_{\theta} \mathcal{L}(\theta),$$

with

$$\mathcal{L}(\theta) = \prod_{\zeta \in \mathcal{D}} Pr(\zeta|\theta, T).$$

Replacing the likelihood \mathcal{L} with the log-likelihood, and using the approximation from equation 2.15, we arrive at the objective

$$\arg \max_{\theta} \frac{1}{|\mathcal{D}|} \sum_{\zeta_d \in \mathcal{D}} \mathcal{R}_{\theta}(\zeta_d) - \log \left[\sum_{\zeta} \exp [\mathcal{R}_{\theta}(\zeta)] \right]. \quad (2.16)$$

¹¹The maximum entropy principle has also been applied in RL to make policies more explorative, combinable, and robust [29], see section 2.1.6.

Note that the left term in equation 2.16 corresponds to the empirical estimated reward of expert demonstrations. The right term contains a log-sum-exp function which is a smooth approximation of the maximum function. It will return a value that is close to the maximal estimated reward of all possible trajectories. Thus the objective amounts to a smooth version of maximizing the smallest difference of expert trajectory rewards and the reward of any possible trajectory, ensuring that expert demonstrations will achieve near-optimal reward when the objective is maximized.

The gradient w.r.t. the reward parameters θ of this log-likelihood objective is

$$\nabla_{\theta} \log \mathcal{L}(\theta) = \frac{1}{|\mathcal{D}|} \sum_{\zeta_d \in \mathcal{D}} \phi(\zeta_d) - \sum_{\zeta} Pr(\zeta|\theta, T)\phi(\zeta). \quad (2.17)$$

The gradient above is the difference between the empirical feature count from expert demonstrations and the expected feature count given the current reward parameter estimate θ . Note that the empirical feature counts only have to be calculated once and do not change, but the expected feature counts given the current reward estimate have to be re-calculated whenever θ changes.

It is intractable in larger MDPs to exactly calculate $\sum_{\zeta} Pr(\zeta|\theta, T)\phi(\zeta)$. For this reason, a dynamic programming approach is proposed in [11] which more efficiently calculates state visitation frequencies and uses those to calculate expected feature counts. This dynamic programming method requires both knowledge of the transition dynamics and of a policy $\hat{\pi}$ optimal w.r.t. the current $\hat{\theta}$.

So far we have assumed the reward function to be a linear combination of some state features ϕ . The maximum entropy IRL approach is straightforward to extend to (differentiable) non-linear reward functions such as deep neural networks, as done in [48]. In this way, manual feature specification can be avoided.

2.3.7 Maximum Causal Entropy IRL

The maximum entropy method described in [11] offers an approximate probability distribution of trajectories in stochastic environments (see equation 2.15). However, as soon as the randomness of transition dynamics significantly influences the decision making of the observed expert, this approximation is not suitable anymore. Essentially, the approach proposed in [11] models the agent as choosing between trajectories, not single actions. However, in stochastic MDPs, the outcome of actions is not known when making a choice. Agents only have information about action outcomes from the past, not the future, they are not able to make choices over an entire trajectory a priori. For example, this is problematic when the MDP contains a state that is extremely hard to reach given the transition dynamics, but yields a relatively high reward compared to other states. If we model the agent as proposed in [11], the relatively high reward contributes exponentially to the likelihood of any trajectory containing this path, while the low probability of reaching it is only taken into account linearly. This makes trajectories containing that state unreasonably likely. In practice however, a rational agent would not pick trajectories leading to the relatively high reward if the probability of reaching it is not high enough. This

flaw in modeling the trajectory likelihood will lead to bad reward estimates whenever transition dynamics play an important role in the problem.

To overcome this issue, the same author proposed an extension to the maximum entropy IRL approach, called maximum causal IRL [46]. Crucially, it maximizes the *causal* entropy that only takes into account information from previous steps in a trajectory and thus does not violate causality. For sequences of states $S_{1:T}$ and actions $A_{1:T}$, causal entropy is defined as

$$\mathcal{H}(A_{1:T}||S_{1:T}) = \sum_{t=1}^T \mathcal{H}(A_t|S_{1:t}, A_{1:t-1}),$$

which is the sum of action choice entropy at all time steps up to some time horizon T given only previous states and actions. Applying this causal entropy while trying to match features of likely transitions with the feature counts observed empirically from the expert, one arrives at a distribution over actions given a state which resembles the one from the non-causal maximum entropy approach but factors in transition probabilities adequately by using Q-values in the exponent

$$\pi_\theta(a|s) \propto \exp [Q_\theta^{\text{soft}}(s, a)], \quad (2.18)$$

which is essentially the same likelihood model as in Bayesian IRL but uses a softmax operator instead of a hard maximum for the value of the next state. Q^{soft} and V^{soft} are defined in section 2.1.6 on maximum entropy reinforcement learning. We can express the exact probability of an action [54] as

$$\pi_\theta(a|s) = \exp [Q^{\text{soft}}(s, a) - V^{\text{soft}}(s)] = \exp [A^{\text{soft}}(s, a)], \quad (2.19)$$

which is the exponentiated soft advantage function A^{soft} of an state action pair.

The probability of a trajectory then is:

$$Pr(\zeta|\theta, T) = \prod_{s_t, a_t, s_{t+1} \in \zeta} \exp [Q^{\text{soft}}(s, a) - V^{\text{soft}}(s)] \cdot T(s_{t+1}|s_t, a_t).$$

This probability distribution over trajectories can be used to maximize the likelihood of expert trajectories as seen in the previous section 2.3.6. The transition dynamics in the distribution do not depend on the reward parameters θ and disappear in the gradient.

2.3.8 Guided Cost Learning

Early approaches to maximum entropy based IRL made use of dynamic programming to calculate the partition function for the distribution of trajectories [11] (section 2.3.6). However, this requires knowledge of the MDP's transition dynamics, a requirement which is often not given, especially in real-world robotics tasks.

As already done in relative entropy IRL [3], the guided cost learning method [14] uses importance sampling to estimate the partition function. Given a proposal distribution for trajectories $q(\zeta)$ which consists of the (unknown) transition dynamics and a proposal policy π_q

$$q(\zeta) = \prod_{s_t, a_t, s_{t+1} \in \zeta} \pi_q(a_t|s_t) \cdot T(s_{t+1}|s_t, a_t),$$

we can estimate the partition function Z based on samples from q :

$$Z = \sum_{\zeta} \exp [\mathcal{R}(\zeta)] = \sum_{\zeta} \frac{\exp [\mathcal{R}(\zeta)]}{q(\zeta)} q(\zeta) = \mathbb{E}_q \left[\frac{\exp [\mathcal{R}(\zeta)]}{q(\zeta)} \right]. \quad (2.20)$$

Based on a sample set \mathcal{D}_q of trajectories, the estimate then is

$$Z \simeq \frac{1}{|\mathcal{D}_q|} \sum_{\zeta} \frac{\exp [\mathcal{R}(\zeta)]}{q(\zeta)}. \quad (2.21)$$

In order to obtain low-variance estimates even with few samples, the proposal distribution q should match the estimated distribution as well as possible [55]. The optimal q is $q^*(\zeta) \propto \exp[\mathcal{R}(\zeta)]$ [14], a distribution that mostly samples trajectories with high reward, since these trajectories have the highest impact on the partition function.

The key idea in guided cost learning is to learn a good proposal distribution q . This is done by basing q on a learned policy π_q that optimizes the current reward function estimate $\hat{\mathcal{R}}$. Instead of solving an entire MDP in the inner loop of an IRL algorithm, the “inverse” step of updating the reward function estimate is now done in the inner loop of a sample-based RL algorithm.

In each iteration of the RL algorithm, we sample a set of trajectories \mathcal{D}_q from π_q and then use those trajectories both to update π_q to maximize $\hat{\mathcal{R}}_\theta$, and to update the reward parameters θ to maximize the log-likelihood of the expert demonstrations

$$\log \mathcal{L}(\theta) = \left(\frac{1}{|\mathcal{D}|} \sum_{\zeta_e \in \mathcal{D}} \hat{\mathcal{R}}_\theta(\zeta_e) \right) - \log \left[\frac{1}{|\mathcal{D}_q|} \sum_{\zeta_q \in \mathcal{D}_q} \frac{\exp [\hat{\mathcal{R}}_\theta(\zeta_q)]}{q(\zeta_q)} \right], \quad (2.22)$$

where the left hand side of the objective is the average estimated expert return and the right hand side is a softmax approximation of the highest possible trajectory return. This is a sampling based estimate of the maximum entropy log-likelihood introduced in section 2.3.6.

In the original paper, guided policy search [56] is used to update the policy π_q w.r.t. the current reward estimate. In theory, any other reinforcement learning method can be used here. In fact, for this thesis we applied SAC, a more modern RL algorithm (section 2.1.7).

Guided cost learning uses non-linear reward functions that do not require a feature function ϕ . Instead, the reward function $\hat{\mathcal{R}}_\theta$ is an artificial neural network which directly takes states and actions as input and for which gradients can be calculated easily, an approach that was previously introduced by [48]. This allows for complex reward functions without the need for manual feature design.

The method is called guided cost learning since the learned policy “guides” the sampling process into areas with high rewards which produce low variance estimates for the gradient steps of the cost learning process.

The algorithm can be summarized as follows:

1. Initialize reward function $\hat{\mathcal{R}}_\theta$ randomly, e.g. as a deep neural network with random weights. Initialize reinforcement learning algorithm which produces sampling policy π_q .
2. Repeat:
 - (a) Sample trajectories \mathcal{D}_q from current sampling policy π_q . Sampled trajectories contain rewards from $\hat{\mathcal{R}}_\theta$.
 - (b) Use \mathcal{D}_q to optimize π_q w.r.t the sampled rewards.
 - (c) Use \mathcal{D}_q for an estimate of the reward parameter likelihood $\mathcal{L}(\theta)$ and update reward parameters with a gradient step to increase the likelihood of expert data.
3. Return $\hat{\mathcal{R}}_\theta, \pi_q$.

When using stochastic gradient descent for updates on the reward function parameters, the authors of GCL found that mixing the sampled trajectories \mathcal{D}_q with some expert trajectories \mathcal{D} improved the stability of results.

Two regularization terms on the reward function are proposed. The first one encourages the reward of trajectories to change locally at a constant rate (lcr) [14] and penalizes high second derivatives over time:

$$g_{\text{lcr}}(\zeta) = \sum_{s_t \in \zeta} [(\hat{\mathcal{R}}_\theta(s_{t+1}) - \hat{\mathcal{R}}_\theta(s_t)) - (\hat{\mathcal{R}}_\theta(s_t) - \hat{\mathcal{R}}_\theta(s_{t-1}))]^2.$$

The second regularization term is only meant for episodic tasks where the expert demonstrations steadily get closer to the goal state with each step. In these cases, forcing the reward function to increase steadily over time in expert trajectories can be encouraged:

$$g_{\text{mono}}(\zeta) = \sum_{s_t \in \zeta} \left[\min(0, \hat{\mathcal{R}}_\theta(s_{t+1}) - \hat{\mathcal{R}}_\theta(s_t)) \right]^2.$$

The likelihood model in GCL uses the maximum entropy principle and not the maximum *causal* entropy one. This means, that it is only a correct model for deterministic environments and only approximates the distribution of trajectories from stochastic MDPs.

2.3.9 Equivalence of IRL and GAN

Interestingly, any sample-based maximum entropy IRL algorithm such as GCL can be seen both as a special case of a generative adversarial network (section 2.2) and an energy based model [9]. We will focus on the former equivalence here and show how guided cost learning can be expressed as a GAN.

If we call the distribution from which expert demonstrations are sampled p , we can express the log-likelihood of reward function parameters (equation 2.22) as

$$\log \mathcal{L}(\theta) = \mathbb{E}_{\zeta \in p} [\mathcal{R}(\zeta)] - \log \left(\mathbb{E}_{\zeta \in q} \left[\frac{\exp[-\mathcal{R}(\zeta)]}{q(\zeta)} \right] \right).$$

To reduce the variance if the sampling distribution q fails to cover trajectories with high reward. This coverage problem can be addressed by using a mixture distribution which consists of both samples from p and q : $\mu = \frac{1}{2}p + \frac{1}{2}q$. In this case the likelihood term becomes

$$\log \mathcal{L}(\theta) = \mathbb{E}_{\zeta \in p} [\mathcal{R}(\zeta)] + \log \left(\mathbb{E}_{\zeta \in \mu} \left[\frac{\exp[-\mathcal{R}(\zeta)]}{\frac{1}{2}p(\zeta) + \frac{1}{2}q(\zeta)} \right] \right).$$

The corresponding loss function is the negative log-likelihood $L(\theta) = -\log \mathcal{L}(\theta)$.

In the context of GAN, the optimal discriminator for a (unknown) generator distribution q is

$$D^*(\zeta) = \frac{p(\zeta)}{p(\zeta) + q(\zeta)}.$$

This discriminator will output the probability that a given trajectory ζ comes from the distribution p of expert demonstrations. If we want to discriminate between expert demonstrations and other demonstrations, we use our model of expert demonstration probabilities being proportional to $\exp[\mathcal{R}(\zeta)]$. The discriminator then depends on the reward parameters θ in the following way:

$$D_\theta(\zeta) = \frac{\frac{1}{Z} \exp [\hat{\mathcal{R}}_\theta(\zeta)]}{\frac{1}{Z} \exp [\hat{\mathcal{R}}_\theta(\zeta)] + q(\zeta)}.$$

Note that this is a special case of a discriminator where the sampling distribution q is known. The loss for the discriminator as defined in section 2.2 is the cross-entropy loss

$$L_D(\theta) = \mathbb{E}_{\zeta \sim p} \left[-\log \left(\frac{\frac{1}{Z} \exp [\hat{\mathcal{R}}_\theta(\zeta)]}{\frac{1}{Z} \exp [\hat{\mathcal{R}}_\theta(\zeta)] + q(\zeta)} \right) \right] + \mathbb{E}_{\zeta \sim q} \left[-\log \left(\frac{q(\zeta)}{\frac{1}{Z} \exp [\hat{\mathcal{R}}_\theta(\zeta)] + q(\zeta)} \right) \right].$$

As is shown in [9], optimizing this discriminator loss L_D with known generator distribution q is equivalent to optimizing the reward function loss function $L(\theta) = -\log \mathcal{L}(\theta)$ from guided cost learning.

At the same time of learning the reward function, the sampling policy π_q is trained to maximize expected reward and its entropy (to be consistent with the maximum entropy principle) with the loss function

$$L_q(\pi_q) = \mathbb{E}_{\zeta \in q} [\mathcal{R}(\zeta)] + \mathbb{E}_{\zeta \in q} [-\log q(\zeta)].$$

It was shown in [9] that the generator loss L_G can be expressed as

$$L_G(\pi_q) = \log Z + \mathbb{E}_{\zeta \in q} [\mathcal{R}(\zeta)] + \mathbb{E}_{\zeta \in q} [-\log q(\zeta)] = \log Z + L_q(\pi_q).$$

Since $\log Z$ depends purely on the discriminator's parameters that are held fixed while training the generator, it becomes obvious that training the generator is equivalent to training the sampling distribution q .

The analogies of IRL and GAN are summarized in the following table:

IRL	GAN
trajectory ζ	sample x
sample distribution q based on π_q	generator G
reward function \mathcal{R}_θ	discriminator D
expert demonstrations \mathcal{D}	training set from true data distribution p

Table 2.3: Analogies between IRL and GAN.

Notably, generative adversarial imitation learning (GAIL) [57], a behavioral cloning method that is also based on GAN, uses a similar structure as the adversarial perspective of GCL. In fact, the policy it learns based on expert demonstrations will converge to one that would also be recovered by maximum entropy IRL. However, since it is using a typical discriminator that does not have access to the generator’s distribution, the reward function remains implicit and cannot be recovered from the algorithm [9]. In the end, it only returns a policy that matches expert behavior as well as possible.

Interestingly, there are also some similarities of GAN and actor critic methods, highlighted in [58].

2.3.10 Adversarial IRL

Two shortcomings of GCL were noted in [15]: modeling distributions of entire trajectories exhibits high variance when estimating the partition function, and using reward functions over state action pairs “entangles” the reward function with the transition dynamics of the MDP used during training and makes the learned function less robust when it is deployed in test settings with different transition dynamics.

The objective of adversarial inverse reinforcement learning (AIRL) is to obtain a reward function that is independent of the training environment’s transition dynamics. This implies the function can be used with variations of the original MDP with different T . They call this kind of learned reward function “disentangled reward”.

They use the principle of maximum causal entropy¹² (section 2.3.7) and define the discriminator on single state action pairs instead of entire trajectories. This yields the optimal discriminator

$$D_\theta(s, a) = \frac{\exp[A^{\text{soft}}(s, a)]}{\exp[A^{\text{soft}}(s, a)] + \pi_q(a|s)}, \quad (2.23)$$

where $A^{\text{soft}}(s, a)$ is the soft advantage function, equivalent to the definition of action probabilities under the maximum causal entropy principle defined in equation 2.19. The discriminator above will output the probability of the state action pair (s, a) coming from the expert demonstrations as opposed to coming from the sampling distribution q .

¹²An improvement over only applying the non-causal version of the principle in GCL

When using reward functions with state-action pairs as domain, the learned reward will be entangled with the transition dynamics of the environment. This becomes more apparent when looking at the equation

$$\mathcal{R}(s, a) = \mathbb{E}_{s' \sim T(\cdot|s, a)} [\mathcal{R}(s')].$$

The expected reward of an action depends on the transition dynamics as they define which future states will be reached with which probability. If we are interested in learning a reward function that also works for different transition dynamics T' , it is thus required¹³ to use a reward function merely over states instead.

This is done by de-composing the soft advantage function A^{soft} into a reward function over states $\mathcal{R}(s)$ and a soft state value function $V^{\text{soft}}(s)$:

$$A^{\text{soft}}(s, a, s') = Q^{\text{soft}}(s, a) - V^{\text{soft}}(s) = \mathcal{R}(s) + \gamma \cdot V^{\text{soft}}(s') - V^{\text{soft}}(s).$$

We can train a discriminator based on this de-composed advantage function which is based on function approximation both for the reward function and state value function

$$D_{\theta, \psi}(s, s') = \frac{\exp [\mathcal{R}_\theta(s) + \gamma \cdot V_\psi^{\text{soft}}(s') - V_\psi^{\text{soft}}(s)]}{\exp [\mathcal{R}_\theta(s) + \gamma \cdot V_\psi^{\text{soft}}(s') - V_\psi^{\text{soft}}(s)] + \pi_q(a|s)}.$$

The sampling distribution q is trained as in guided cost learning by optimizing for the reward function estimate $\hat{\mathcal{R}}_\theta$ with an additional term incentivizing policy entropy.

2.3.11 Summary

In the previous subsections we have seen an overview of some of the most important algorithms for inverse reinforcement learning. This section provides an overview of their core differences and similarities in table 2.4. All methods introduced in previous sections are compared concerning their requirements for expert optimality, the type of reward functions used, the way of picking single solutions from the solution set, the required frequency of solving an MDP, and whether they require knowledge of the transition dynamics T .

¹³In theory it is also possible to convert from $\mathcal{R}(s, a)$ to $\mathcal{R}(s)$ by solving a system of linear equations based on the transition dynamics. However, this is only possible if the dynamics are known.

Method		Expert optimality	Reward function	Picking solution	Frequency of solving MDP	Requires known T
Linear Programming [6] section 2.3.3		fully optimal	linear in features	distance heuristic	each IRL iteration	yes
Quadratic Programming [43] section 2.3.4		fully optimal	linear in features	distance heuristic	each IRL iteration	no
Bayesian IRL [45] section 2.3.5	softmax optimal	tabular		mean of Bayesian posterior	each MCMC step in each IRL iteration	no
MAP Bayesian IRL [52] section 2.3.5	softmax optimal	tabular		maximum of Bayesian posterior	each IRL iteration	no
Improved Bayesian IRL [47] section 2.3.5	softmax optimal	tabular		single sample from MCMC	each MCMC step	no
Max. Entropy IRL [11] section 2.3.6	softmax optimal	linear in features		maximum entropy principle	each IRL iteration	yes
Max. Causal Entropy IRL [46] section 2.3.7	softmax optimal	linear in features		maximum causal entropy principle	each IRL iteration	yes
Guided Cost Learning [14] section 2.3.8	softmax optimal	nonlinear		maximum entropy principle	once	no
Adversarial IRL [15] section 2.3.10	softmax optimal	nonlinear		maximum causal entropy principle	once	no

Table 2.4: Summary of important features of the described IRL algorithms. Most algorithms assume the expert to be softmax optimal, i.e. the probability of picking an action is proportional to the exponentiated reward or action value, e.g. $\pi(a|s) \propto \mathcal{R}(s, a)$. Besides GCL and AIRL, all covered methods require either given features or a tabular representation of the reward function where each possible reward is stored as an entry in a vector or table. Both GCL and AIRL only solve the MDP once and update the reward function in the inner loop of the reinforcement learning algorithm. They are also the only covered methods that use nonlinear reward functions.

3 | Methods

As discussed in chapter 2, the two chosen methods, GCL and AIRL, have some desirable properties based on which they were chosen to be implemented for this thesis:

- Both methods work with nonlinear reward functions implemented as (deep) neural networks and thus do not require manual specification of features.
- Both methods work with unknown transition dynamics T and can deal with large and continuous state and action spaces.
- Both solve an MDP only once and update the reward function at the same time, instead of repeatedly solving many MDPs and updating the reward function only in between. This comes with large speed improvements.
- Both are based on the maximum entropy principle (section 2.3.6), AIRL even on the maximum causal entropy principle (section 2.3.7), making it more suitable for environments with high stochasticity. They use a principled way of selecting one reward function out of the many possible ones that are consistent with expert demonstrations.

This chapter contains descriptions of the methods and algorithms used for the experiments in this thesis. Section 3.1 provides details on the implementation of the IRL algorithms. While the implementations are highly similar to the methods described in the prerequisites section 2.3, some small modifications were added which empirically improved performance. This is followed by a description of the environments used in section 3.2 and how expert data was obtained in section 3.3. Section 3.4 describes the implemented baselines against which the performance of IRL methods was compared in the experiments. The metrics used to evaluate results are described in the following sections; evaluation of robustness to distributional shift on the test environment is covered in section 3.5 and a novel metric to directly compare two reward functions is introduced in section 3.6.

3.1 IRL Algorithms

For both algorithms implemented in this thesis (GCL and AIRL), modifications were developed and tested experimentally. Changes made to both algorithms are described in sections 3.1.2 through 3.1.5. This is followed by descriptions of changes specific to GCL in section 3.1.6 and specific to AIRL in section 3.1.7.

3.1.1 General IRL Implementation

Reward functions are implemented as artificial neural networks, consisting of two hidden layers with 64 units each and parametric rectified linear units (PReLU) [59] as activation functions. A batch normalization layer [60] is applied between the hidden layers to facilitate training. The input layer directly takes state variables, or, if the reward function operates on state action pairs, also action variables. The output layer is a single unit without an activation function, computing a weighted sum of the “features” generated by the last hidden layer. The same architecture is used for the state value network in AIRL.

The reward function parameters are updated in the inner loop of the SAC algorithm. After each 1000 environment interactions, the respective algorithm is applied to learn better reward parameters θ . For this purpose, an Adam optimizer [61] with a learning rate of 0.001 was used. If not stated otherwise, the IRL algorithms are provided with 1,000 expert demonstrations.

3.1.2 Reinforcement Learning

Instead of using guided policy search [56] or trust region policy optimization (TRPO) as in GCL and AIRL respectively, the soft actor critic algorithm (section 2.1.7) has been chosen for this project as the reinforcement learning method. SAC is a natural choice for three reasons:

- Firstly, it is a maximum entropy method, which fits nicely to the model of the expert demonstrations likelihood defined in maximum causal entropy IRL (section 2.3.7).
- Secondly, it has empirically exhibited stronger performance and more robust training compared to other state of the art reinforcement learning methods [12].
- Thirdly, as an off-policy method it comes with an experience replay memory that can be used as a cheap source for the samples needed to update the reward function.

All policy and value networks used in SAC have the same architecture: two hidden layers with 64 units and rectified linear unit (ReLU) activations [62]. The following hyperparameters were used to tune the reinforcement learning part of the experiments:

Parameter	Description	Pendulum	Lunar Lander
β	importance of $\mathcal{H}(\pi)$	0.02	0.05
n_{train}	number of steps	50,000	250,000
n_{explore}	number of exploration steps	10,000	20,000
T	max steps per episode	200	1000
γ	future reward discount factor	0.99	0.99
ρ	exponential parameter averaging	0.95	0.95
α	learning rate	10^{-3}	10^{-3}
	replay memory capacity	10^6	10^6
	batch size	100	100

Table 3.1: Reinforcement learning hyperparameters used for all experiments.

The SAC networks are trained after each episode. The training runs for n_{train} environment steps, out of which the first n_{explore} are run with an additional ϵ -greedy exploration strategy (section 2.1.6) with ϵ decreasing linearly from 1 to 0. The replay memory is adapted in a way that rewards are replaced by their current estimate whenever they are sampled. This slows down the training process slightly.

3.1.3 Reward Function Pre-training

One big advantage of the two implemented IRL methods is that they do not require manually specified features for which the reward function can be expressed as a linear combination. Instead, the use of neural networks allows us to approximate a non-linear reward function that operates directly on states or state action pairs. In a way, the hidden layers of the reward network are trained to find suitable features automatically, based on the provided data. The output layer then computes a linear combination of these features to yield the reward estimate for a state.

The early iterations of GCL and AIRL are started with a randomly initialized reward network¹. Only after the network has been updated sufficiently, the computed rewards will start to guide the sampling policy into regions that are more similar to the expert demonstrations. This could be observed especially in complex problems such as Lunar Lander (section 3.5.2) where the true reward achieved in the first iterations of training was often strongly negative and substantially worse than a random agent baseline.

In order to reduce the time needed for the reward network to produce a strong signal of good performance for the reinforcement learning algorithm, we add an additional step of pre-training the first layer of the reward network with supervised learning based on the expert demonstrations and data collected from a random agent. Notably, this does not require any reinforcement learning steps and can be done once for all future training runs. We are not aware of previous methods using pre-training on the reward function. Pre-training empirically has positive effects on the early stages of training for both GCL and AIRL (section 4.3).

¹Specifically, we use Kaiming He initialization for the network parameters [59].

We train a classifier to distinguish state action pairs as coming either from a data set of expert demonstrations or from a set of random agent demonstrations. The classifier is a simple neural network with two hidden layers and ReLU activations. Both hidden layers have 64 units, the same as the reward and value networks described in section 3.1.1. The second hidden layer is preceded by a batch normalization layer [60] for faster training and succeeded by a dropout layer [63] with a 50% chance of deactivating units to avoid overfitting. The output layer is a single neuron with a tanh activation function to predict values of -1 for random data and $+1$ for expert data.

The loss of this classifier is the mean squared error between true label $y \in \{-1, 1\}$ and the classifier’s output $C(x)$ over a batch of example states or state action pairs \mathcal{D}_x :

$$L_C = \frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}_x} (y - C(x))^2.$$

The number of units in the classifier’s hidden layer corresponds to the number of units in the first layer of the reward network. Before running the IRL algorithm, we replace the first hidden layer of the reward network with the weights of the classifier’s corresponding layer. This provides the reward net with strong initial low-level features of what distinguishes expert demonstrations from random behavior, which can be combined non-linearly in higher layers to predict the reward.

In our experiments, the classifier was trained on 100,000 transitions each from expert and random trajectories (Pendulum: 500 trajectories each, Lunar Lander: 521 expert trajectories and 921 random trajectories). The data was split 60:20:20 into a training set, a validation set, and a test set. The classifier’s loss for both environments is shown in figure 3.1. The classifier achieved 95.2% test accuracy for the Pendulum environment and 98.2% for Lunar Lander (for a description of the environments see section 3.2).

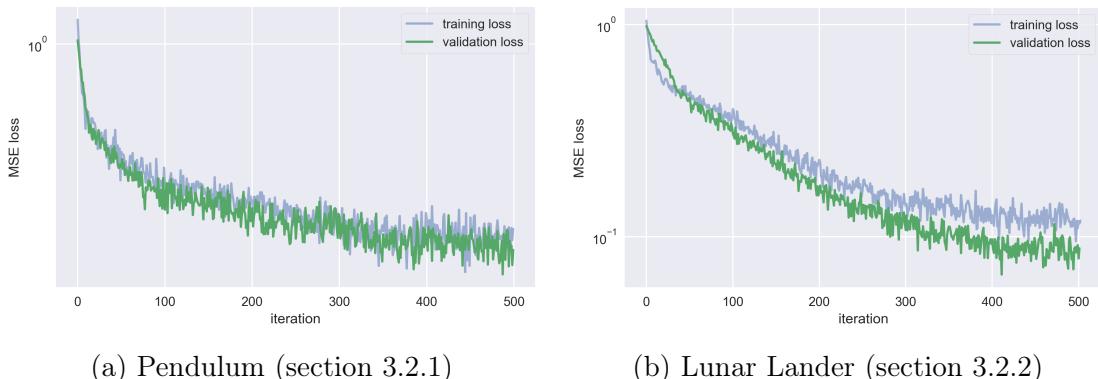


Figure 3.1: Loss of the classifier C used for pre-training of the reward network. Dropout was activated for training loss and deactivated for validation loss. Convergence to good results is achieved for both data sets after about 500 training iterations.

3.1.4 Reward Magnitude Penalty

Both GCL and AIRL can be seen as energy based models where the reward or advantage of a state action pair is the negative energy for the model of action probabilities. However, using rewards in the exponent of the model can lead to numeric instability if we deal with very large reward values. For this reason, it can be desirable that the reward function outputs small absolute values most of the time.

In theory, any positive affine transformation of the reward function (i.e. adding positive scalars and multiplying by positive scalars) preserves the optimal policy [37]. This entails that the optimal policy does not change when scaling our reward function down by multiplying it with a small positive value. For numerical stability of the algorithm, we prefer reward functions with smaller rewards.

We incentivize a small magnitude of rewards by adding a regularization term to the objective, multiplied by a hyperparameter λ to trade-off between maximizing the log-likelihood of expert demonstrations and reward magnitude

$$L'(\theta) = L(\theta) + \lambda \mathbb{E}_{s,a \in \mathcal{D}_s} [\hat{\mathcal{R}}_\theta(s, a)^2].$$

Bounding the learned reward function to a range of small values is not unheard of [6], [43], [45]. In [43], the rewards are bounded to be no larger than 1 which is required for convergence proofs of their algorithm. Our approach does not set a hard boundary for the magnitude of reward values, but instead allows a trade-off between this objective and the likelihood objective, controlled by λ .

While in theory reward functions can be scaled down or up (by multiplying with positive constants) without changing the optimal policy [37], deep reinforcement learning with neural network based function approximation uses finite memory and resources and is not guaranteed to converge to an optimal solution for any scaled version of a reward function. In fact, reward scaling has been applied to make problems more learnable. It has been observed that reward scaling and clipping can have large effects, but those effects are inconsistent across different MDPs [38]. An adaptive normalization of reward magnitude was proposed in [64], applying this to IRL is left open for future research.

We can conclude that in practice an MDP might become harder to solve if the reward function is scaled down too much. Thus, λ has to be carefully chosen for different problem settings to both ensure numerical stability while not strongly reducing the capabilities of the used RL algorithm.

3.1.5 Mixture Batch

Both GCL and AIRL use importance sampling to estimate the partition function of the likelihood term they are maximizing. This estimate potentially suffers from high variance if the sampling distribution q does not cover trajectories or state action pairs with high reward [14], [9], [15]. For this reason, they propose to mix expert demonstrations with those generated by q to form \mathcal{D}_q given that expert demonstrations have high true reward.

On the other hand, in our experiments we have found that including expert demonstrations in the importance sampling estimate can lead to weaker gradients and a significantly slower training process (section 4.1.1). The problem here is that under some circumstances the trajectories sampled from π_q can become almost irrelevant in the gradient as both terms involved become dominated by the expert demonstrations.

As we can see in equation 2.22 (repeated below), the left summand of the expert demonstrations log-likelihood depends solely on data collected from the expert. The right summand is the partition function estimate:

$$\log \mathcal{L}(\theta) = \left(\frac{1}{|\mathcal{D}|} \sum_{\zeta_e \in \mathcal{D}} \hat{\mathcal{R}}_\theta(\zeta_e) \right) - \log \left[\frac{1}{|\mathcal{D}_q|} \sum_{\zeta_q \in \mathcal{D}_q} \frac{\exp[\hat{\mathcal{R}}_\theta(\zeta_q)]}{q(\zeta_q)} \right].$$

The right summand of the log-likelihood contains a log-sum-exp function, which is basically equivalent to the maximum function if one summand is significantly higher than the others. If \mathcal{D}_q now contains both expert demonstrations and trajectories from π_q , the training process can stall if at least one expert demonstration has significantly higher rewards than the sampled trajectories. In this case, the log-likelihood essentially turns into the objective of minimizing the difference between average expert return and maximum expert return. The sampled trajectories then become irrelevant and the reward function is no longer updated in a direction that improves the true performance of π_q .

For our experiments we have implemented both variants: \mathcal{D}_q containing only samples from π_q or mixing them with expert samples.

3.1.6 GCL Modifications

The implementation of guided cost learning suffers from two related issues: high variance and numerical instability. Both arise from the importance sampling estimate of the log-likelihood partition function. This estimate depends mostly on the largest value over all sampled trajectories for $\exp[\hat{\mathcal{R}}_\theta(\zeta_q)]$, divided by $q(\zeta_q)$. The exponentiated return of a trajectory is often a very large number, while $q(\zeta_q)$ (the product of all action probabilities along a trajectory) is often very close to zero. Dividing a very large number by a very small number requires high precision in the calculations and can yield results that are too big to be expressed adequately by common programming languages.

The log-sum-exp operator in the log-likelihood term of GCL and AIRL is prone to numerical instability as it involves taking a sum over extremely large numbers. We use the simple trick of subtracting the highest value in each exponent and adding it outside of the log-sum-exp operator. In this way, the exponents become smaller while the final results remains the same:

$$\log \left(\sum_{x \in \mathcal{X}} \exp [x] \right)$$

$$\begin{aligned}
&= \log \left(\sum_{x \in \mathcal{X}} \exp [x] \cdot \frac{\exp [\max(\mathcal{X})]}{\exp [\max(\mathcal{X})]} \right) \\
&= \log \left(\sum_{x \in \mathcal{X}} \exp [x - \max(\mathcal{X})] \cdot \exp [\max(\mathcal{X})] \right) \\
&= \log \left(\exp [\max(\mathcal{X})] \cdot \sum_{x \in \mathcal{X}} \exp [x - \max(\mathcal{X})] \right) \\
&= \max(\mathcal{X}) + \log \left(\sum_{x \in \mathcal{X}} \exp [x - \max(\mathcal{X})] \right).
\end{aligned}$$

This was applied to our implementation of GCL in the log-likelihood objective (equation 2.22) by subtracting the maximum value of all the importance weights in each summand

$$\max_{\zeta_q} \frac{\exp [\hat{\mathcal{R}}_\theta(\zeta_q)]}{q(\zeta_q)}.$$

Especially in early stages of the training, when the sampling distribution q is still of low quality, the estimate of the likelihood’s partition function often exhibits extremely high variance. For this reason, we applied *truncated importance sampling* [65], a method that reduces variance while introducing a bias in the estimate, the trade-off at the core of machine learning. We use a slightly modified version of truncation, where we truncate the value of $q(\zeta_q)$ in the likelihood to be larger than some small ϵ . As q does not depend on θ , the introduced bias does not vary strongly with θ and empirically did not lead to an influential bias of the gradient. At the same time, especially during early stages of training, the gradient variance can greatly be reduced with this technique. Truncated importance sampling can still be beneficial in later stages of the training process, as it allows “the successful use of some proposal distributions [with] poor approximations to the tails of the target distribution” [65].

Using these two adaptions in our GCL implementation, the obtained results were more stable and numerical errors could be avoided.

3.1.7 AIRL Modification

The main difference in our implementation of AIRL is the way in which training batches for updating θ are obtained. In the original implementation, full trajectories from the last 20 iterations are used to construct batches for stochastic gradient descent (SGD) updates. However, one of the motivations for using SAC is that samples are readily available in the replay memory. For this reason, we construct batches directly from the replay memory which, depending on its capacity, can contain transitions from much earlier stages of training.

This approach comes with both advantages and disadvantages. On the one hand, reusing the replay memory reduces the required memory space and can provide diverse samples that are less correlated as they come from a higher number of different policies and trajectories and are closer to an i.i.d. distribution. This makes the gradients computed with SGD have lower variance and enables faster training as it can avoid oscillations or even divergence of the parameters [17]. On the other hand,

by not putting entire trajectories into batches, some batches might not contain any important goal states. This is especially problematic in episodic tasks where only one or a few states in the end of an episode are relevant with high reward.

3.2 RL Environments

The two environments used for this thesis are both part of the OpenAI gym framework [66]: Pendulum and Lunar Lander. Both of them have continuous state and action spaces and are challenging problems for the reward learning task. While Pendulum has been used before as an evaluation environment for IRL algorithms (e.g. for AIRL [15]), to our knowledge inverse reinforcement learning has not successfully been applied to the openAI Lunar Lander environment before².

3.2.1 Pendulum

The Pendulum environment (also often termed “Inverted Pendulum”) is a typical problem in the control literature [66]. The pendulum is fixed at one point and starts out in a random position. The objective is to swing it upright and to keep it in a vertical position by accelerating in one of the two possible directions. Figure 3.2 shows two example states of the game.

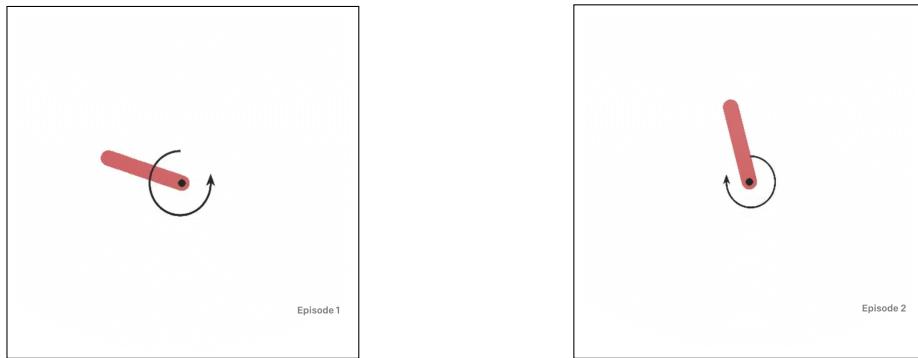


Figure 3.2: Pendulum environment. The pendulum starts hanging down and has to be controlled in a way that makes it stand upright vertically. Source: <https://gym.openai.com/envs/Pendulum-v0/>

The reward function is defined as a weighted sum of: the (normalized) angle ρ of the pendulum (0 if standing upright), the velocity $\dot{\rho}$ of the pendulum, and the acceleration $\ddot{\rho}$:

$$\mathcal{R} = -n(\rho)^2 - 0.1 \cdot \dot{\rho}^2 - 0.01 \cdot \ddot{\rho}^2.$$

where $n(\rho)$ normalizes³ the angle to be in the interval $[-\pi, \pi]$. Rewards are always negative and highest when the pendulum is standing upright without any velocity and acceleration. As $\ddot{\rho}$ is given by the agent’s action, the reward function is defined

²The Lunar Lander environment has been used for experiments of shared control between humans and robots [67]. A different, but similar version of the game has been used with imitation learning in [68].

³ $n(\rho) = ((\rho + \pi) \bmod 2\pi) - \pi$

over state action pairs. Episodes are run for 200 time steps. The best possible return is 0, the worst case is about -2140.

The state variable Pendulum MDP is 3-dimensional and defined as

$$[\cos(\rho), \sin(\rho), \dot{\rho}].$$

The action defines the desired acceleration $\ddot{\rho}$. Both $\dot{\rho}$ and $\ddot{\rho}$ are constrained to not exceed some constants for maximal velocity and acceleration.

The true reward function can easily be expressed as a linear combination of three features (angle, velocity, and acceleration), which can be computed from the state variables and action variable. This makes this environment suitable to be used with IRL methods that require given features. However, the methods tested in this thesis do not require features.

3.2.2 Lunar Lander

In the Lunar Lander environment, a space ship has to be navigated to land smoothly on a landing pad on the surface of the moon. The vehicle has one main engine to move upwards and two side engines to rotate left and right. The action space is continuous⁴ with three degrees of freedom. The state space is a subset of \mathbb{R}^8 . The first two state variables specify the vehicle's position, the next two the velocity, followed by angle, angular velocity, and two indicators for ground contact of the two legs. The environment can be rendered as shown in figure 3.3.

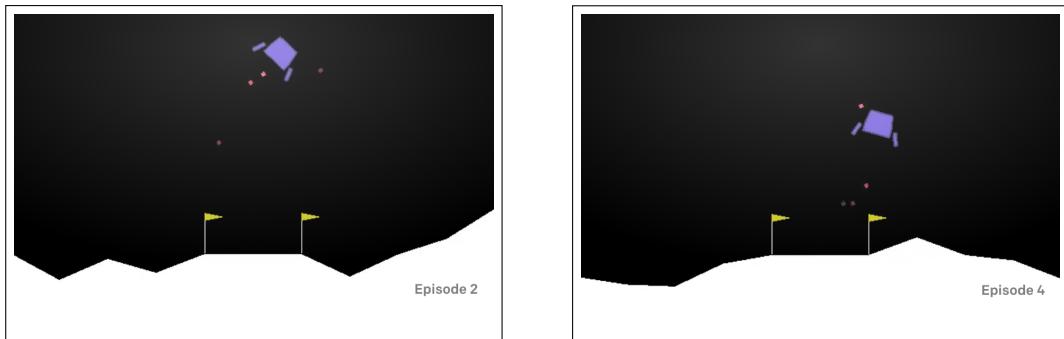


Figure 3.3: Lunar Lander environment. The lander starts in the top of the screen and has to be navigated to land smoothly between the two yellow flags. Source: <https://gym.openai.com/envs/LunarLander-v2/>

The true reward function in Lunar Lander depends on information that is not present in state and action variables. Technically this makes the reward function partially observable. The state-based reward r_t at time t for a state action pair (s_t, a_t) is the sum of the following elements:

- The squared euclidean distance to the goal point, multiplied by 100. This encourages being closer to the goal position.

⁴There is also a discrete version of the game with binary values for the engine being on or off, based on Pontryagin's maximum principle according to which it is optimal to either fire an engine full throttle or to turn it off.

- The squared speed of the vehicle, multiplied by 100. This encourages flying at low speed.
- The absolute value of the angle, multiplied by 100. This encourages staying in a vertical position.
- 10, if the left leg has ground contact
- 10, if the right leg has ground contact

The state based part of the reward r_t is shaped by subtracting the state based reward value of the last step r_{t-1} . This constitutes the shaped part \mathcal{R}_S of the reward function $\mathcal{R}_S = r_t - r_{t-1}$, r_0 is defined to be 0.

If the game has ended in the current step t , 100 points are added to the reward if the landing was successful, or 100 points⁵ subtracted if the lander crashed. This information is not present in the state variables and taken from the internal physics engine. The power consumption of the main engine action a_m and side engine action a_s is factored in with a cost of 0.3 and 0.03, respectively. The combined reward function for the input (s_t, a_t) is

$$\mathcal{R}(s_t, a_t) = \mathcal{R}_S - 0.3 * a_m - 0.03 * a_s + 100 \cdot \mathbb{I}_{\{\text{game done success}\}} - 100 \cdot \mathbb{I}_{\{\text{game done crash}\}}.$$

The ground truth reward function for Lunar Lander is shaped and depends on information that is not present directly in the state variables. This makes it very difficult to manually specify features for which the reward function is a linear combination. Extensive experiments trying to learn good features with supervised machine learning based on sampled transitions were not successful. Other methods have been proposed to construct features suitable for IRL [69], but they still require manual human work. The difficulty of finding good features for the Lunar Lander environment makes this MDP hard to solve with many of the older IRL algorithms such as [43], [45], or [11]. While in theory more effort could have been put into finding a good feature function, we picked GCL and AIRL specifically as they implicitly construct features during training as part of the non-linear reward functions and thus avoid this step of manual search. Given the high complexity of the original Lunar Lander reward function, solving it with IRL is a challenging task and, to the best of our knowledge, no previous publication has reported to have learned good reward functions for this environments.

3.3 Expert Data Collection

Both the Pendulum and the Lunar Lander game are hard for humans to play optimally as they require fast reaction time and are very sensitive to even small overhangs of acceleration which are hard to correct. The author of this thesis was not able to match the performance of computer-controlled agents for either of the two games.

⁵For comparison, more than 90% of rewards during training range from -2 to 4.5 and only about 0.5% transitions have a reward of 100. The end of each episode has the highest reward in magnitude.

For Lunar Lander, [67] have reported that most humans crash the lander in more than 80% of trials.

On the other hand, computer agents trained with reinforcement learning are able to perform actions in much faster succession and are able to quickly correct small deviations from the desired course. For this reason, RL agents trained on the true reward function were used to collect expert demonstrations. Previous methods have also used RL agents trained on the true reward function for expert demonstrations [6], [43], [45], [48], [14], [15]. In this section the used agents and their training process are described.

3.3.1 Pendulum

The Pendulum environment can be solved using SAC rather robustly. A hyperparameter search led to good results with the values specified in table 3.1. Training 50 experts with different random seeds, the worst average return over 100 episodes achieved after training for 50,000 steps was -163.8 . On average, experts achieved -150.5 return. The best expert was chosen to collect demonstrations for IRL. In total, 1,000 trajectories were stored from this selected expert agent. The worst trajectory in the set of expert demonstrations has a return of -293.8 and no trajectory was discarded due to low performance.

3.3.2 Lunar Lander

As before, 50 expert agents were trained with different random seeds with hyperparameters according to table 3.1, with the only difference being that experts were trained for 500,000 steps to ensure that all of them converge to good results. The worst average return over 100 episodes achieved after training for 500,000 steps was 183.1. On average, experts achieved 262.8 return. 1,000 trajectories were collected from the best expert agent. The worst trajectory in the set of expert demonstrations has a return of -123.2 and no trajectory was discarded due to low performance.

Figure 3.4 and 3.5 illustrate the behavior of expert agents and random agents respectively. We can see that the expert agents learn a strategy of staying close to the middle, descending quickly first and then becoming slower as they approach the goal position.

3.4 Baseline Algorithms

The results obtained with IRL are compared to three baselines: A reinforcement learning agent based on SAC with access to the true reward function, GAIL, and a random agent. The true reward function baseline tells us how well the same algorithm would perform if inverse reinforcement learning was not required. Notably, the learned reward function can in theory lead to better results than the true reward function, e.g. if it is better shaped and thus more learnable. The random agent baseline can be seen as a worst case that results should never fall short of. GAIL

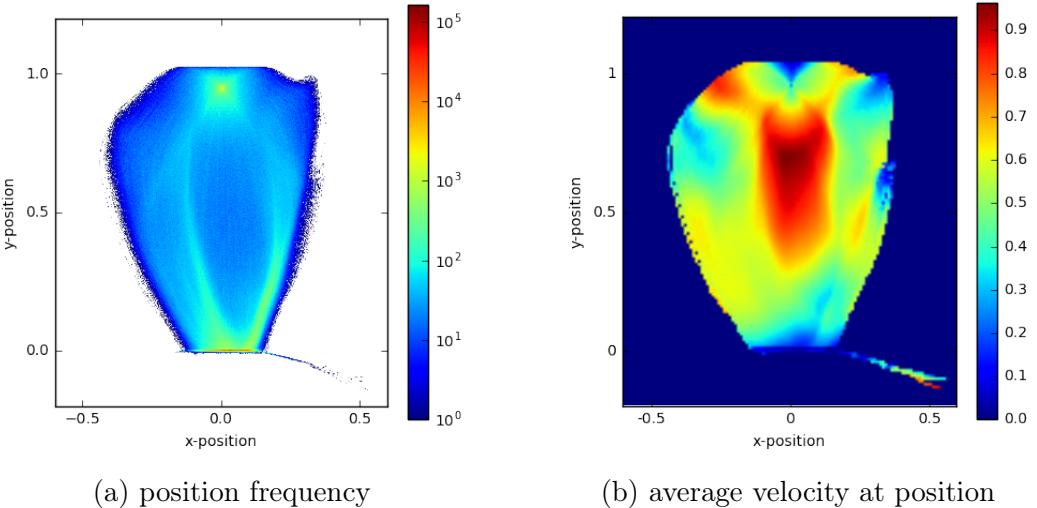


Figure 3.4: Expert performance on Lunar Lander environment. In (a) we can see that the agent mostly stays in the center part of the map and descends with a slight curve towards the bottom. Note that frequencies are plotted on a logarithmic color scale. In (b) we can see that the agent descends with the highest speed when in a central position and not too close to the ground. The closer it gets to the goal position, the more its velocity is reduced. Just above the goal position the speed approaches zero.

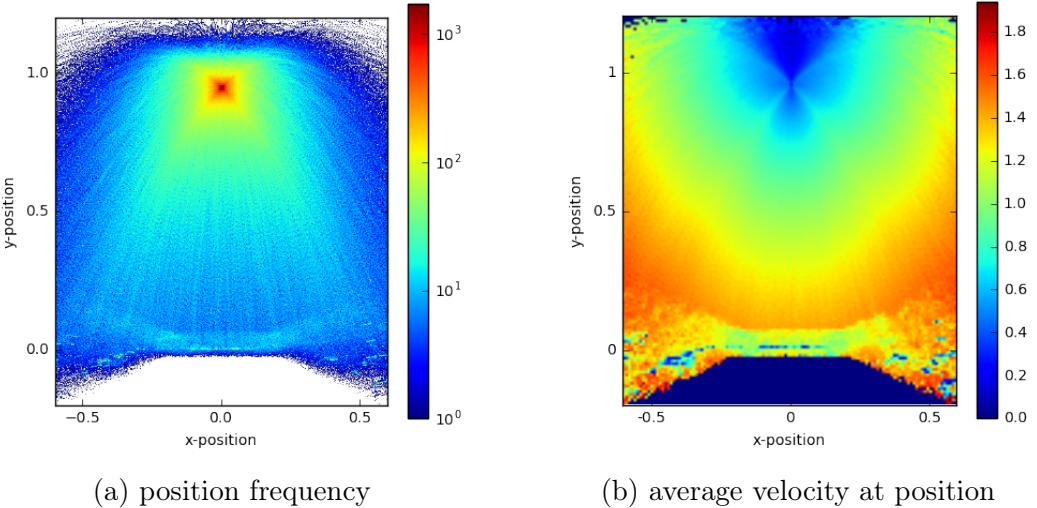


Figure 3.5: Random agent performance on Lunar Lander environment. The random agent paths are more spread out and in most cases do not end in the goal position. In (b) we can see that without directed use of the engines, the agent gains speed through gravity and crashes on the ground in most cases.

is similar to GCL and AIRL but does not construct an explicit and usable reward function. In [15] it was found that policies learned with GAIL are often not very robust if applied to transfer learning tasks.

In figure 3.6 we can see the baseline performances on the Pendulum environment. All performance is measured in the true episode return, even if the true reward function was not known (GAIL) or not used (random agent). Figure 3.7 shows baseline performance for Lunar Lander. The baselines are described in more detail in the following subsections.

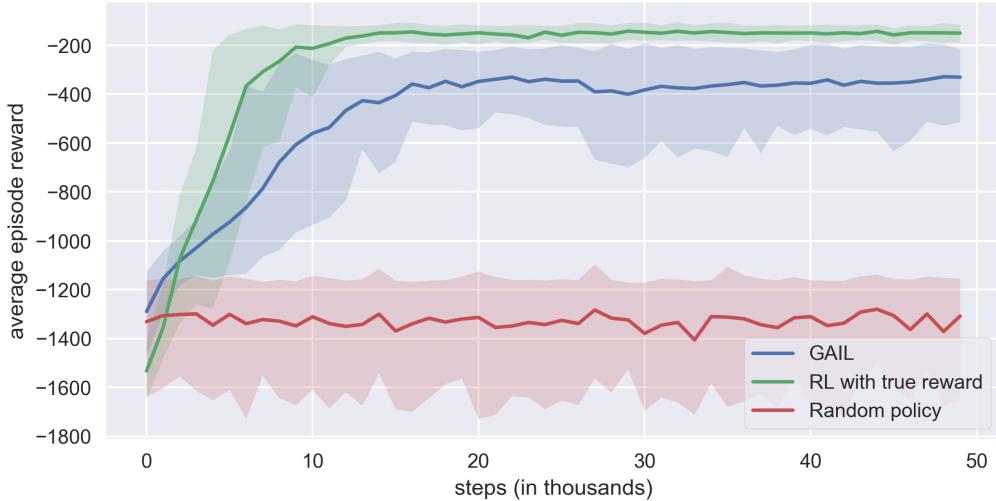


Figure 3.6: Baselines for the Pendulum environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

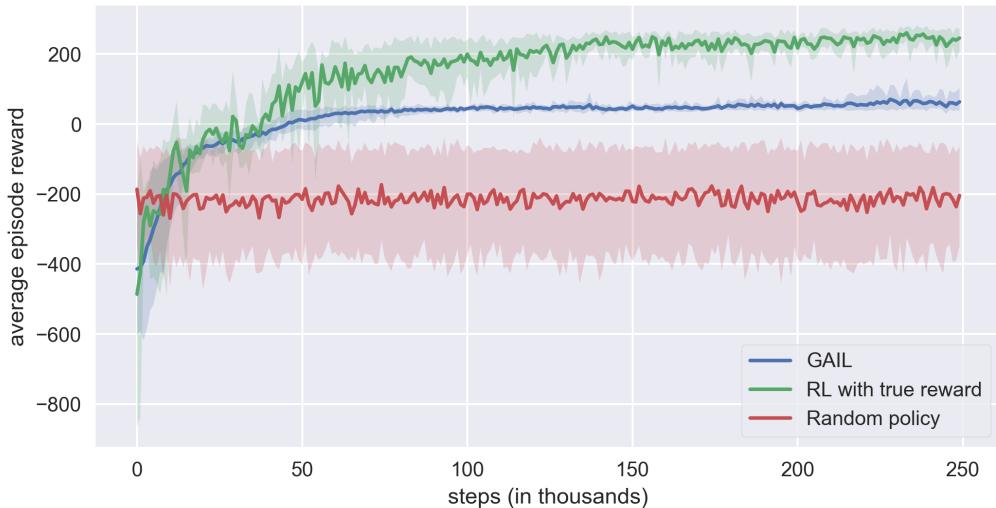


Figure 3.7: Baselines for the Lunar Lander environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

3.4.1 True Reward Baseline

The true reward baseline is a SAC, trained in exactly the same way as the IRL algorithms, but without using a reward function estimate. Instead, it is provided the true reward function and can learn to optimize the true objective directly. Hyperparameters are set to the same values as defined in table 3.1 for the IRL algorithms.

3.4.2 GAIL Baseline

As GCL and AIRL, GAIL is based on maximum causal entropy inverse reinforcement learning. However, for performance reasons it does not involve explicitly learning a reward function and instead only learns a policy by matching state action pair occupancy measures with empirical expert demonstrations (behavioral cloning). IRL

can be seen as learning a reward function that induces a policy matching the expert’s feature occupancy measure. Thus, they argue that the expensive step of repeatedly solving the IRL problem can be skipped by directly training a policy that would be induced by IRL [57]. This is done by using a GAN where the discriminator is trained to distinguish between expert state action pairs and other state action pairs, and the policy is trained with the discriminator as reward function. However, if the policy reaches optimality, the discriminator will output 0.5 for all samples which makes it unsuited to be used as a reward function [15].

In conclusion, GAIL is highly related to GCL and AIRL, but since it does not recover an explicit reward function from expert demonstrations, it is less portable to transfer tasks that are similar but differ in some way [15]. If applying GAIL to a similar problem, one can re-use the learned policy, but not train a new one based on a learned reward function.

The GAIL baseline was trained in a similar way as the two IRL algorithms on 1000 trajectories of expert demonstrations. The reinforcement learning part is using TRPO [70] and an existing implementation from the authors of [15] was used. Even after some tuning of hyperparameters, we did not succeed to match the reported average return of -226 from [15] on the Pendulum environment, with our results converging to -356 instead. To our knowledge, no results on the Lunar Lander environment have been reported for GAIL so far.

3.4.3 Random Agent Baseline

The random agent baseline functions as a worst case lower bound: if performance of our methods is worse than acting randomly, they should never be used. The data for this baseline was collected by sampling uniformly randomly from the action space without learning a policy. The performance depicted in figures 3.6 and 3.7 are based on 50 repetitions to get a more stable estimate of average performance.

3.5 Robustness Evaluation

The focus of this thesis lies on evaluating how robust the learned reward functions obtained with GCL and AIRL are to distributional shift of the transition dynamics, i.e. how well they generalize to similar problems with the same state and action space but with different probabilities of moving between states⁶. For this purpose, we created modified version of both the Pendulum and the Lunar Lander environment to test if the learned reward functions can successfully be applied on those modified problems to learn high return policies. With one of the main claims of AIRL being that the learned reward function $\hat{\mathcal{R}}$ is *disentangled* from transition dynamics and hence more robust (section 2.3.10), we put special emphasis on testing the robustness of this method. At the same time, we evaluate a state-only version of guided cost learning, which we hypothesize to be more robust than the original version.

The general procedure of robustness experiments is as follows:

⁶This is a particular case of transfer learning in reinforcement learning. For more information we refer to the survey [71].

- Learn reward function $\hat{\mathcal{R}}_\theta$ based on expert demonstration \mathcal{D} using GCL or AIRL on the original Pendulum or Lunar Lander MDP.
- Use the learned reward function on the modified MDP with different transition dynamics T to learn a policy $\hat{\pi}$ using SAC.
- Evaluate the learned policy $\hat{\pi}$ on the modified MDP using the true reward function \mathcal{R} to estimate average true return.

3.5.1 Pendulum

For the Pendulum environment, we introduce two basic modifications that are both based on changing the maximum allowed velocity. In the original Pendulum environment, the pendulum cannot move faster than 8 units per step. The two new variants come with maximum speed of 6 and 10.

The baselines for the Pendulum (max-speed 6) environment are depicted in figure 3.8. We can see that both the RL agent trained on the true reward and the GAIL agent achieve higher return than in the original task. The true reward RL agent converges to the best possible return of about 0 in less than 5,000 steps.

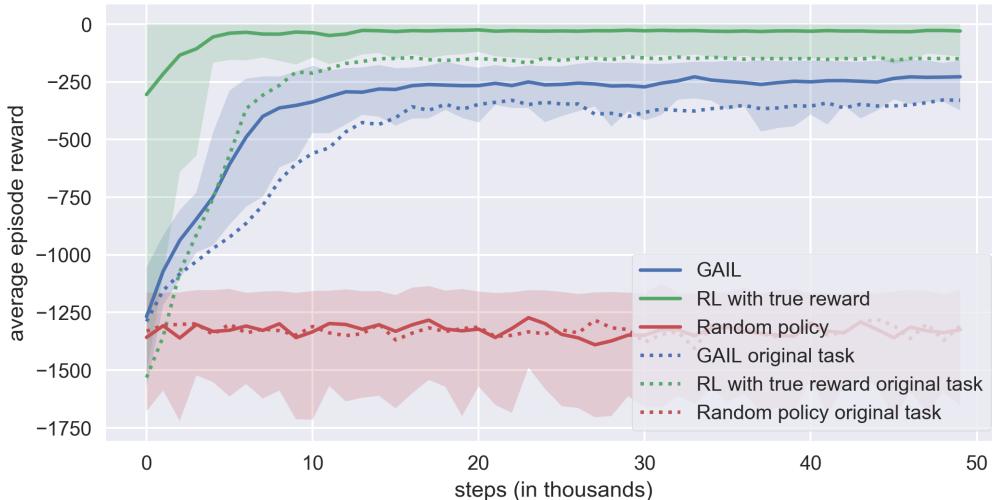


Figure 3.8: Baselines for the Pendulum (max-speed 6) environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

While still easier for the true reward RL agent, with a max speed of 10 the GAIL agent drops in performance as we can see in figure 3.9.

3.5.2 Lunar Lander

The following environment parameters were changed to create the Lunar Lander (modified) environment:

- Main engine power reduced from 13 to 5
- Side engine power reduced from 0.6 to 0.5

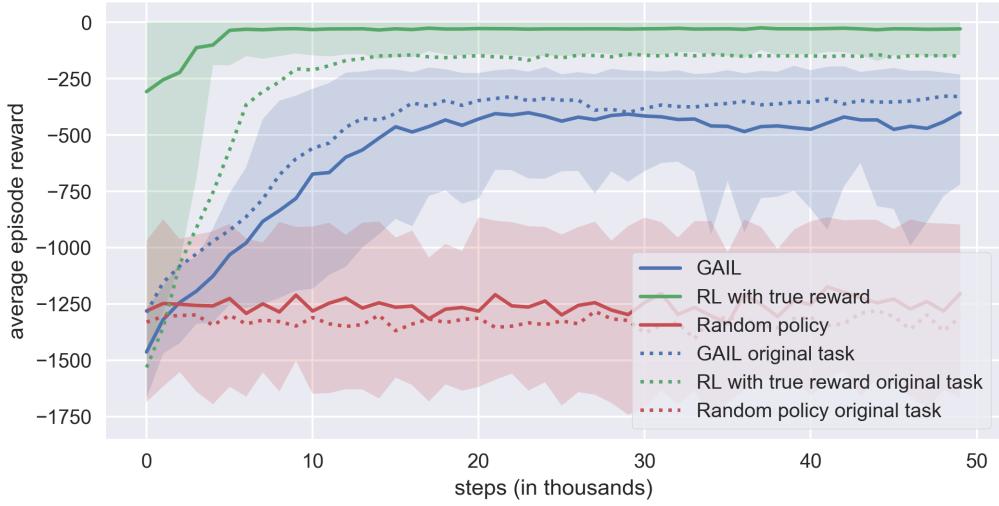


Figure 3.9: Baselines for the Pendulum (max-speed 10) environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

- Lander body polygon shape changed (see figure 3.10)
- Leg width increased from 2 to 3 (see figure 3.10)

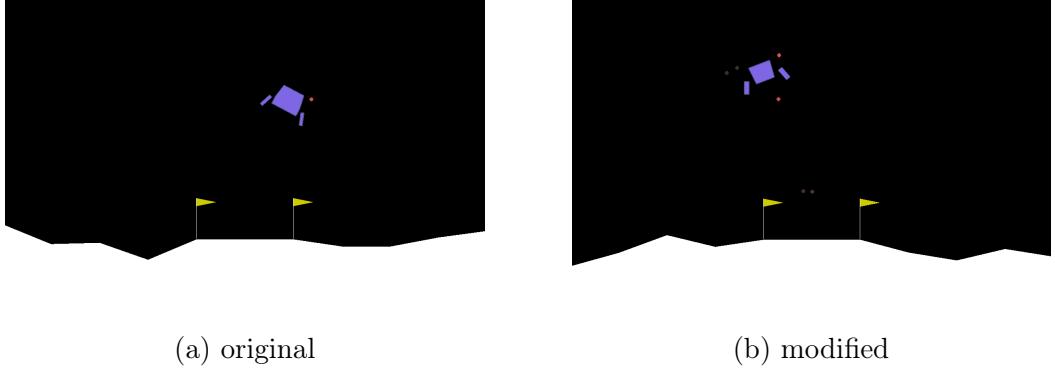


Figure 3.10: Lunar Lander (modified) environment. The legs of the lander were made thicker and are positioned further away from the body. The body shape now has more mass in the top and less in the bottom.

In figure 3.11 we can see the performance of the three baseline agents on the modified Lunar Lander environment. The average return dropped significantly for all three baselines. The environment has become significantly more difficult, mostly due to a decrease in the power of the main engine, making it harder to slow down against the pull of gravity.

3.6 Shaped Reward Loss Metric

In many publications on IRL, new algorithms are evaluated indirectly by looking at the true return of a policy that was trained on the reward function estimate, instead of directly evaluating the learned reward function [6], [43], [45], [48], [15],

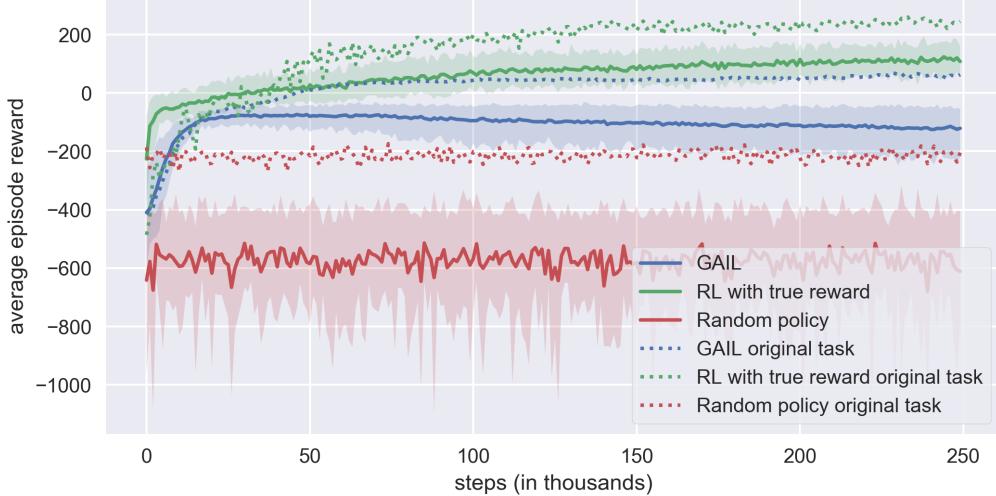


Figure 3.11: Baselines for the Lunar Lander (modified) environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

[14]. Metrics that directly compare the learned reward function with the true one are desirable as they do not require costly reinforcement learning and do not depend on the reinforcement learning algorithm to actually find the optimal solution for a given reward function.

The authors of BIRL [45] use a direct L2 loss between $\hat{\mathcal{R}}$ and \mathcal{R} after regularizing their values by setting an upper bound for reward values. There are two problems with this metric: very different reward functions can lead to the same optimal behavior [37] and very similar reward functions can lead to different optimal policies⁷. Especially the former issue makes the direct L2 loss a bad metric: even if we learn a policy that is essentially equivalent to the true reward as it leads to the same optimal behavior, it could get assigned a high reward loss.

We⁸ propose a new metric that directly compares two reward functions while (at least partially) avoiding the problem of assigning high loss to seemingly different reward functions that actually lead to the same optimal policy. To achieve this, we take into account the findings of [37] about certain types of transformations on reward functions under which the optimal policy stays invariant. Those are positive affine transformations

$$\mathcal{R}' = b \cdot \mathcal{R} + c, \quad (3.1)$$

with $b, c \in \mathbb{R}, b > 0$, and potential based transformations for any potential function $\phi : \mathcal{S} \rightarrow \mathbb{R}$

$$\mathcal{R}'(s, a, s') = \mathcal{R}'(s, a, s') + \gamma\phi(s') - \phi(s). \quad (3.2)$$

This holds in the same way for reward functions over state action pairs $\mathcal{R}(s, a)$ or only states $\mathcal{R}(s)$. For any other transformation on the reward function, there exist transition dynamics T for which the optimal policy of the transformed reward

⁷Consider that marginal differences between rewards of states can lead to a complete shift in the optimal policy. While this small difference might

⁸The idea for this metric was developed jointly by the author of this thesis and other members of the IRL benchmark team: Adria Garriga-Alonso, Anton Osika, Max Daniel, and Sayan Sarkar. <https://github.com/JohannesHeidecke/irl-benchmark>

function will differ from the original one. As long as no information is given on the transition dynamics, the transformations in equations 3.1 and 3.2 are both sufficient and necessary conditions to guarantee policy invariance after reward transformations.

When comparing a learned reward function $\hat{\mathcal{R}}$ to a ground truth reward function \mathcal{R} , we want to ignore any differences arising from policy-preserving transformations. Thus, we learn a linear transformation with parameters b, c and a potential function ϕ to minimize the L2 loss between the ground truth and the transformed learned reward

$$a, b, \phi = \arg \min_{b, c, \phi} \left\| \mathcal{R}(s, a, s') - (b \cdot \hat{\mathcal{R}}(s, a, s') + \gamma \phi(s') - \phi(s) + c) \right\|_2.$$

As before, b is constrained to be positive. As ϕ can in theory be any real function, we use an artificial neural network to approximate the best possible function. The constant c can easily be incorporated into ϕ and can for this reason be dropped⁹. The shaped loss then is

$$L_{\text{shaped}}(\mathcal{R}_1, \mathcal{R}_2) = \min_{b, \phi} \left\| \mathcal{R}(s, a, s') - (b \cdot \hat{\mathcal{R}}(s, a, s') + \gamma \phi(s') - \phi(s)) \right\|_2. \quad (3.3)$$

We model ϕ as a fully connected neural network with two hidden layers of 128 units each and ReLU activation functions [62]. A batch norm layer is used to speed up training [60] and dropout with a probability of 10% is used to avoid overfitting [63].

One difficulty is that we need to compute the loss on the entire reward function domain: all possible states or even state action pairs. This is impractical for larger environments, so we use stochastic gradient descent with batches sampled from the environment to learn a, b , and ϕ . For practical reasons, we limit our batches to samples from an equal mixture of transitions (s, a, r, s') from expert demonstrations and random agent demonstrations, 10,000 trajectories each. The data is split into 60% training set, 20% validation set, and 20% test set. In theory one could assign higher weights to rewards of states that are similar to expert demonstrations, but this is left for future research.

While this new shaped reward function loss brings the promise of directly comparing reward functions without having to run any RL algorithms, it also comes with serious shortcomings.

One weakness of the shaped reward loss metric is that, in practice, reinforcement learning algorithms have limited computational power and memory. Typical assumptions that guarantee convergence to optimal policies¹⁰ are thus usually not given. Some reward functions from an equivalence class might be significantly easier to learn than others and therefore better. The metric is completely agnostic to the “learnability” of a reward function it judges.

Another valid criticism is that there are more ways of transforming the reward function that keep optimal policy invariant, that go beyond linear transformations and potential shaping. However, they require knowledge of the transition dynamics¹¹.

⁹This can be done by using $\phi'(s) = \phi(s) + \frac{c}{\gamma-1}$ instead of ϕ .

¹⁰e.g. visiting each state infinitely many times

¹¹As a trivial example, consider that we can assign arbitrary rewards to states that are impossible to be reached according to T .

The shaped reward loss covers all transformations that are possible without knowing the transition dynamics [37], but in each specific case it can be agnostic to a large number of other possible transformations.

A third shortcoming of this metric is that even a small shaped reward loss for some reward estimate $\hat{\mathcal{R}}$ can still correspond to large differences in true return and $\hat{\mathcal{R}}$ might lead to a very weak optimal policy. Consider the simple toy example of an MDP with a single choice between two states, s_1 and s_2 with true rewards $\mathcal{R}(s_1) \gg \mathcal{R}(s_2)$. A wrong reward estimate $\hat{\mathcal{R}}$ assigns higher reward to s_2 . However, using shaping, this difference can be infinitesimally small, yielding a shaped reward loss close to zero even though the true return achieved is terrible.

Overall, the shaped reward loss metric is an interesting concept that tries to look at the problem of IRL from a different perspective, but there are still too many theoretical and practical objections (section 4.5) that should be addressed in future work.

4 | Results

In this chapter, the results of the conducted experiments, based on the methods described in chapter 3, are presented. The performance of the IRL algorithms on the original environments is presented in section 4.1. This is followed by an analysis of the robustness of learned reward functions, when used on transfer tasks with different transition dynamics, in section 4.2. Section 4.3 presents the performance improvements with pre-training. Subsequently, the effects of providing a smaller number of expert demonstrations is covered in section 4.4. In the end of the chapter, section 4.5 describes results for the shaped reward loss metric.

4.1 Original Task Performance

As described in the previous chapter, there is a variety of different modifications and choices of hyperparameters to evaluate. In this section, we look at the performance of agents trained with estimated reward functions $\hat{\mathcal{R}}$, where the performance is based on the ground truth reward function. We look at two slightly different aspects:

- After each IRL training epoch of 1000 environment steps, the sampling policy π_q is evaluated on a test environment for ten episodes. In this step, the policy does not use any exploration, i.e. it deterministically picks $\mu(s)$ and ignores $\sigma(s)$ (section 2.1.7) and does not apply ϵ -greedy exploration (section 2.1.6).
- After learning a new reward function, we use it with a fresh reinforcement learner and evaluate the true return of its policy after training for 50 (Pendulum) or 250 (Lunar Lander) epochs. This RL agent learns with a new exploration schedule and with a fixed reward function that does not get updated during training, making it easier to achieve good performance.

The performance of the sampling policy π_q is summarized in section 4.1.1 for the Pendulum environment and section 4.1.3 for the Lunar Lander environment. Results for taking learned reward functions and keeping them fixed for the training of new reinforcement learning agents can be found in section 4.1.2 for Pendulum and section 4.1.4 for Lunar Lander.

The choices of the hyperparameters λ for the reward magnitude penalty (section 3.1.4) and whether to use a mixture batch or not (section 3.1.5) was done based only on results from the Pendulum environment. Performing a hyperparameter search specifically for Lunar Lander was prohibitively costly given the computational environment.

4.1.1 Sampling Policy for Pendulum

In figure 4.1 we can see results for GCL on Pendulum without using mixture batches and with different choices of λ , the parameter defining the importance of the reward magnitude penalty (section 3.1.4). The experiments with a relatively high penalty of $\lambda = 100$ converged to the best results, almost matching true reward performance in the end of training. The dip in return coincides with the end of the ϵ -greedy exploration phase, after which less random transitions are sampled from π_q . Performance is shortly reduced but this effect is overcome after about 10 training epochs.

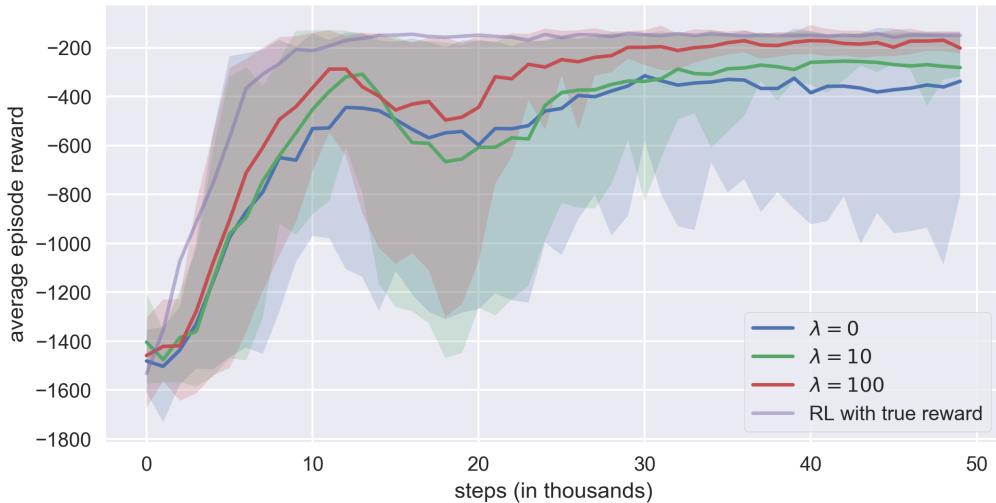


Figure 4.1: Reward Magnitude Penalty with GCL on Pendulum environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

We repeat the experiments from figure 4.1, but this time using mixture batches (section 3.1.5) coming from both expert demonstrations and sampled demonstrations for training. As figure 4.2 shows, results are clearly inferior in this setting. Contrary to how GCL was implemented in [14], we find that mixture batches can have an adverse effect, with training performance hardly exceeding a random policy.

While the reward magnitude penalty had a positive effect on the GCL algorithm, it shows different results on AIRL. As depicted in figure 4.3, when using a value of 100 for λ , which was the best result obtained for GCL, the learned reward function does not yield a strong sampling policy. With lower values for λ , results do converge to levels near the true reward RL agent. We can see that with a low value for λ , the return rises faster in early iterations, but converges to a lower performance compared to not using any reward magnitude penalty. At the same time, again contrary to the GCL results, using a mixture batch as described for the original implementation [15] has a strong positive effect on results. In fact, after a few thousand environment interactions the reward functions learned with mixture batches and no reward magnitude penalty train the policy to convergence faster than the true reward function.

As described in section 4.2, we want to learn reward functions that generalize well to changes in the transition dynamics when put to use on a new environment. One of the central claims of AIRL is that a disentangled reward function can be obtained

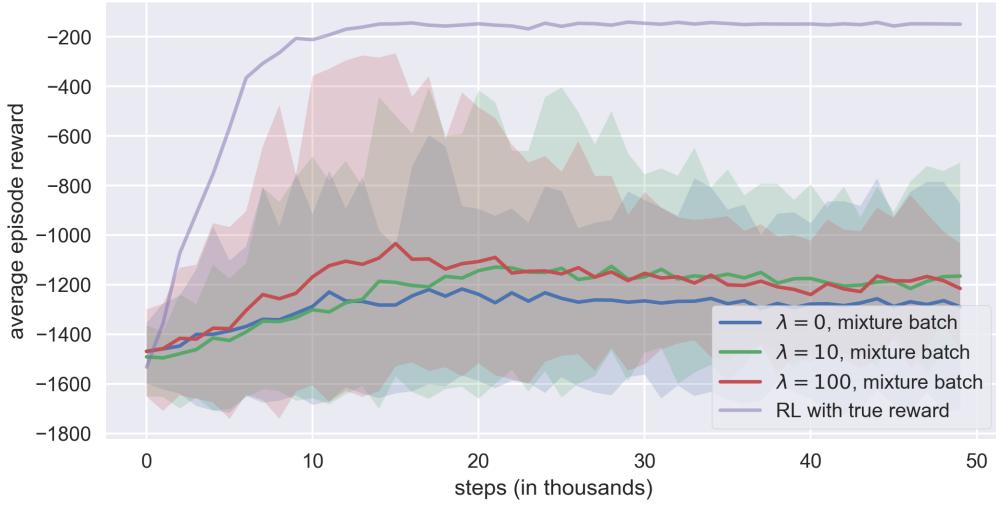


Figure 4.2: Reward Magnitude Penalty and mixture batch with GCL on Pendulum environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

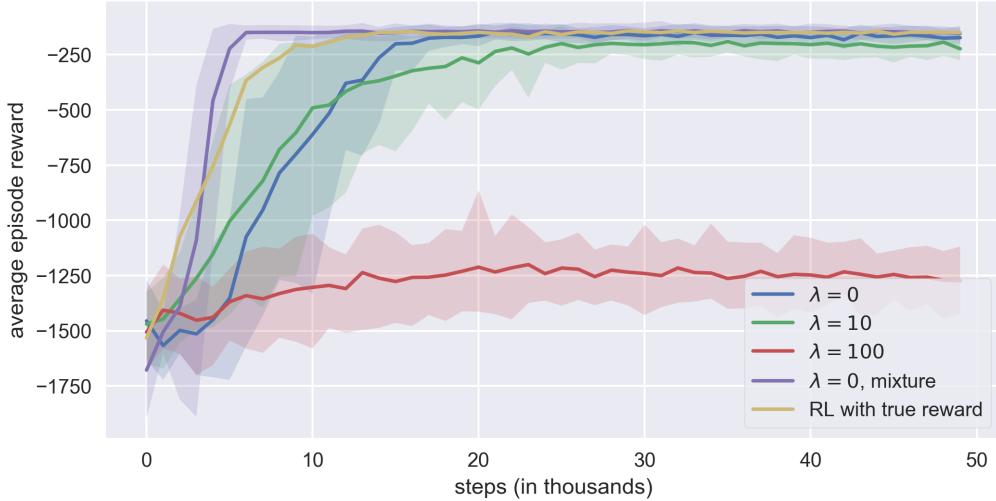


Figure 4.3: Reward Magnitude Penalty with AIRL on Pendulum environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

by restricting the reward function domain to states instead of state action pairs¹. For easier comparison, we also train a GCL algorithm that only depends on states. The difference in performance between the two GCL version (both using $\lambda = 100$ and no mixture batches) is compared in figure 4.4.

Based on the hyperparameter search for AIRL (figure 4.3), we settle for the two settings that worked best for the experiments described in the following parts of the thesis: $\lambda = 0$ and either using mixture batches or only sampled batches. While results converge much faster with mixture batches, both converge to a final which

¹The other changes in AIRL are due to using the principle of maximum *causal* entropy (section 2.3.7) - the main insight is that state based reward functions are needed to separate the reward function from transition dynamics.

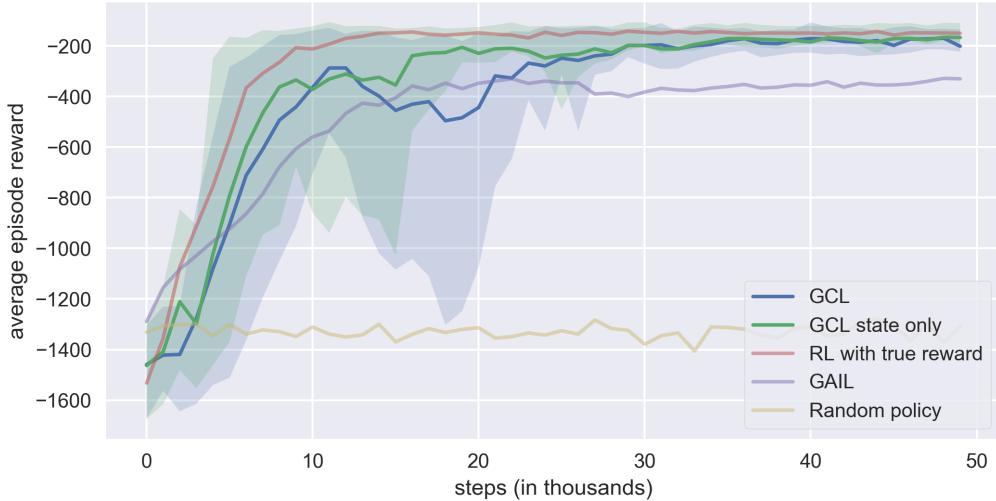


Figure 4.4: GCL results on Pendulum environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

at times even exceeds RL agents that were trained with the true reward function. The performance of both methods is contrasted with the baselines in figure 4.5.

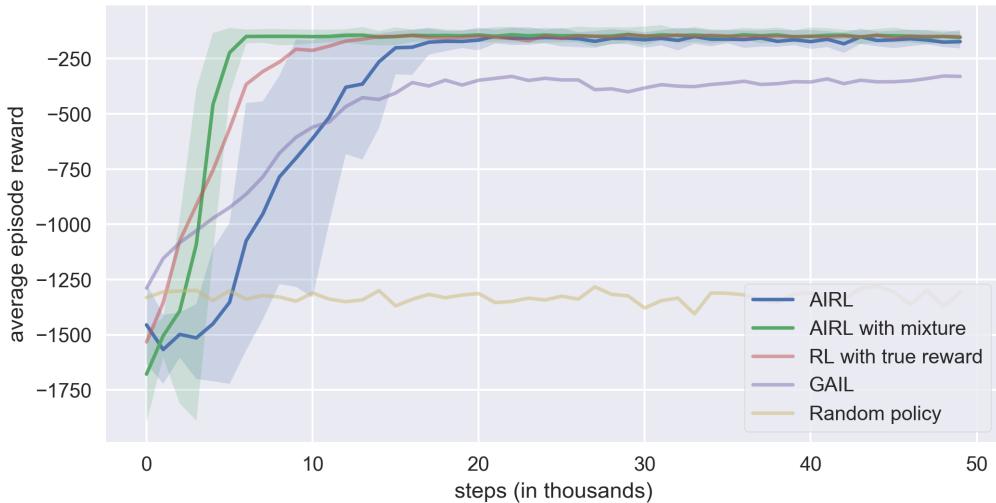


Figure 4.5: AIRL results on Pendulum environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

We can compare the performance of all four chosen settings directly in figure 4.6. While all four settings converge to expert-like performance, AIRL does so significantly faster and with significantly lower variance in the results between different random seeds - especially when using mixture batches.

4.1.2 Fixed Reward Function for Pendulum

When taking the learned reward function after 50 epochs and using it to train a new RL agent from scratch, functions learned with both GCL and AIRL lead to policies with average true returns very similar to what is achieved with the ground truth

reward function. This is illustrated in the violin-plots of figure 4.7. Average true returns are also summarized in table 4.1.

As table 4.1 shows, our results exceed those reported in [15]. One can suspect that a big part of the performance improvement is due to using a different reinforcement learning algorithm. Their expert baseline was trained with TRPO [70] and achieved -179.6 average return, while our SAC expert baseline achieved -150.5 on average. Interestingly, our experiments show that RL agents trained with the learned reward functions achieve better results on average than those trained with the original reward function. This could mean that the learned reward functions are shaped in a more learnable way.

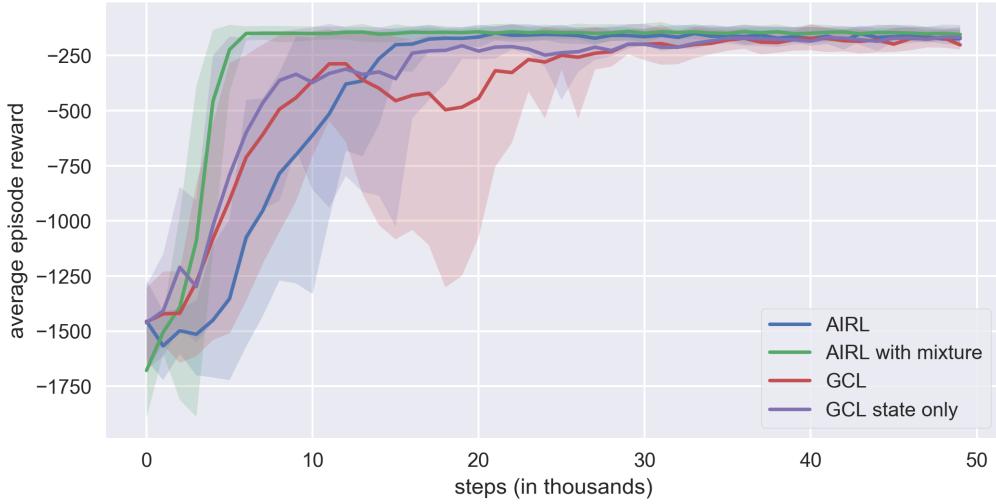


Figure 4.6: GCL and AIRL comparison on Pendulum environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

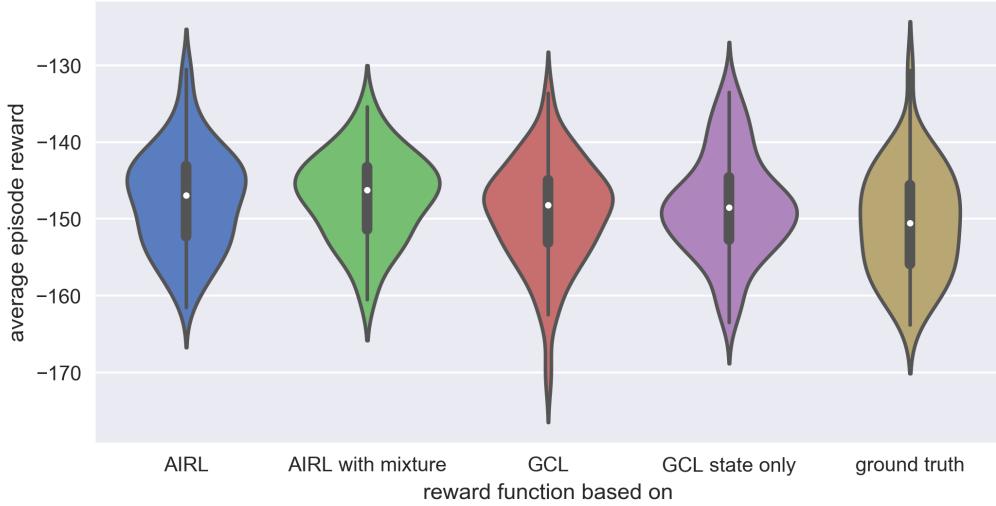


Figure 4.7: True performance on Pendulum after training on reward estimate (ground truth for comparison). Average trajectory return over 100 trajectories, after training for 50 epochs. Results from 100 different random seeds.

	GCL (ours)	AIRL (ours)	expert SAC	GCL [15]	AIRL [15]	expert [15], TRPO
Average return	$\hat{\mathcal{R}}(s, a)$: -149.24	no mixture: -147.43				-150.5 -261.5 -204.7 -179.6
	$\hat{\mathcal{R}}(s)$: -148.41	mixture: -147.09				

Table 4.1: Improvement over reported state of the art for Pendulum environment. Results for GCL (ours) and AIRL (ours) were obtained by running SAC on 100 reward functions learned by GCL and AIRL respectively for 50 epochs and then taking the average return over 100 trajectories.

This suspected shaping can be seen visually when looking at the reward function plots in figure 4.8. The shaping is especially apparent in the example reward function learned with AIRL and mixture batches in subfigure 4.8c. Considering that bright regions correspond to high reward, the function clearly encourages the pendulum to accelerate to a left-swing, slowing down the closer it gets to an upright position. Without the mixture batch, the learned reward function more closely resembles the ground truth (subfigure 4.8b) but with strong negative rewards for a larger area in the bottom half of the circle. The reward functions learned with GCL appear more “chaotic” but clearly also involve high reward regions for low speed at an upright position. While the highest reward with state-only GCL is achieved with a high speed in the top left area of the circle, this can only repeatedly be achieved by completely swinging around the circle, which passes through a lot of low reward regions. It seems like, even though the reward function looks less smooth and symmetrical compared to the others, the optimal policy it produces is still also optimal for the true reward function.

4.1.3 Sampling Policy for Lunar Lander

This subsection discusses results for the Lunar Lander environment (section 3.2.2). As stated before, the relatively long run-time of IRL for this more complex environment prohibited an extensive hyperparameter search. Instead, we use the settings that worked best for Pendulum (section 4.1.1), in the hope that they transfer well.

We first shift our focus on results obtained with GCL. In figure 4.9 we can compare the performance of our default GCL implementation with the state-only variant. It becomes evident that both version have very high variance in results throughout the training, an issue that was also reported in [15]. The average episode reward stays below the performance of GAIL and far below the performance of RL agents that are trained on the true reward function. The state-only version of GCL performs poorer on average and comes with higher variance. About 10% of the state action version results reach average episode rewards above 150 for the second half of training.

Figure 4.10 shows results for AIRL on the Lunar Lander environment. During training, the sampling policy π_q which is trained on the current reward estimates

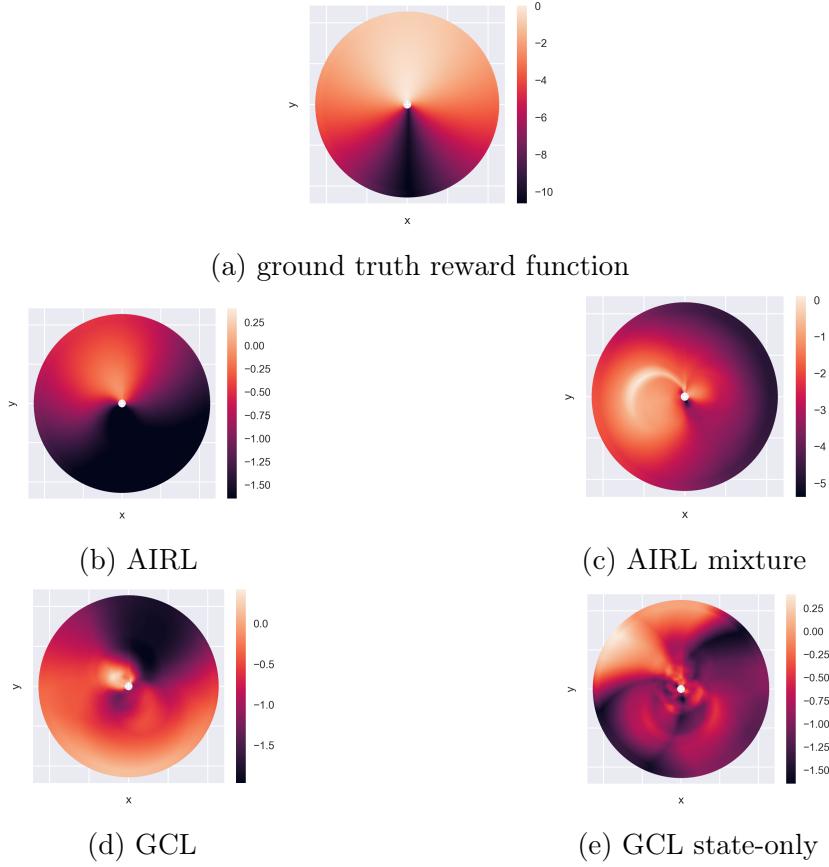


Figure 4.8: Visualization of learned reward functions for Pendulum environment. Each point (x, y) corresponds to an angle and velocity. Points next to the center mean zero velocity, points in the outmost circle correspond to maximal velocity, with linear interpolation in between. Drawing a line through the center and a point gives the corresponding angle of the pendulum. Points with bright color have high reward, points with dark color low reward.

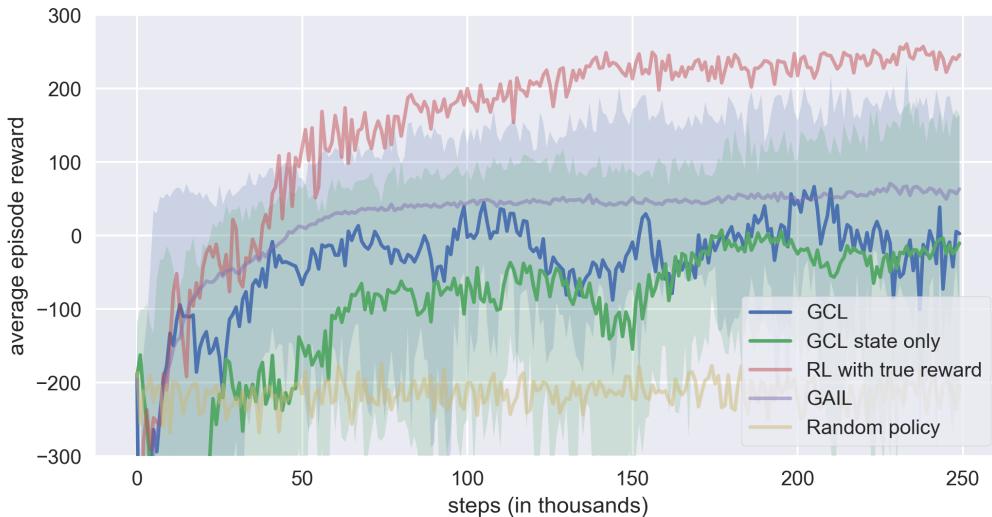


Figure 4.9: GCL results on Lunar Lander environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

on average converges to returns just above zero, far below the expert performance of more than 200 and below results from GAIL of about 50. The training converges quickly and does not change much after the initial 20% of training steps.

Only about 0.5% of transitions in the experience replay memory used for AIRL contain the high rewards assigned to either successfully finishing a landing or crashing. When randomly constructing batches in the IRL step to estimate the likelihood gradient, many batches do not contain any examples of completed landings. We hypothesize that this is a main cause for the convergence at suboptimal performance. In future work, one might want to try sampling entire trajectories for batches, or using prioritized experience replay [27].

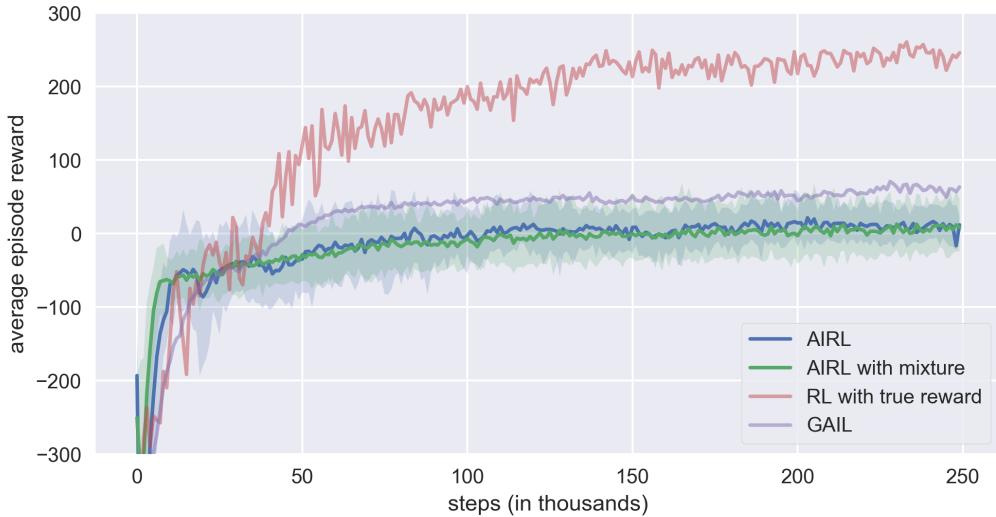


Figure 4.10: AIRL results on Lunar Lander environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

Figure 4.11 directly compares the training performance of AIRL and GCL on the Lunar Lander environment. It is evident that guided cost learning suffers from the problem of high variance which was one of the motivations behind developing AIRL [15]. Guided cost learning does at times exceed the performance of AIRL, but not consistently so.

4.1.4 Fixed Reward Function for Lunar Lander

Even though both IRL algorithms fail to reach expert performance during training, we can see in figure 4.12 that results are much better when using the learned reward functions to train new RL algorithms from scratch while keeping the reward functions fixed. As observed for the Pendulum environment before, results with this approach exceed the average performance of agents that were trained on the ground truth reward function. This can be seen both in figure 4.12 and table 4.2. Notably, a significant part of the 50 ground truth agents we trained got stuck at a return of about 90 and did not improve further. It looks like with the IRL reward functions this is much less of an issue.



Figure 4.11: GCL and AIRL comparison on Lunar Lander environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

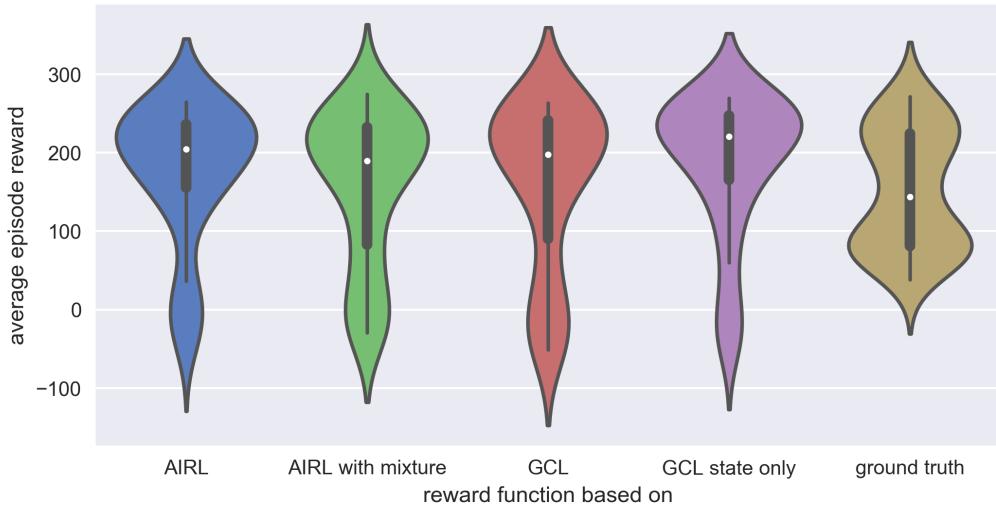


Figure 4.12: True performance on Lunar Lander after training on reward estimate (ground truth for comparison). Average trajectory return over 100 trajectories, after training for 250 epochs. Results from 50 different random seeds.

	GCL	GCL state-only	AIRL	AIRL with mixture	expert SAC
Average return	174.97	157.25	158.84	185.97	151.63

Table 4.2: Average return when using fixed GCL and AIRL reward functions for training. Results for GCL and AIRL were obtained by running SAC on 100 reward functions learned by GCL and AIRL respectively for 250 epochs and then taking the average return over 100 trajectories.

Visualizing the reward function estimates for Lunar Lander is more difficult, as the state variables alone are 8-dimensional. Instead, we visualize the average reward obtained at certain positions, aggregating over the other state variables and sampling from transitions provided by one of the expert agents. This can be seen in figure 4.13.

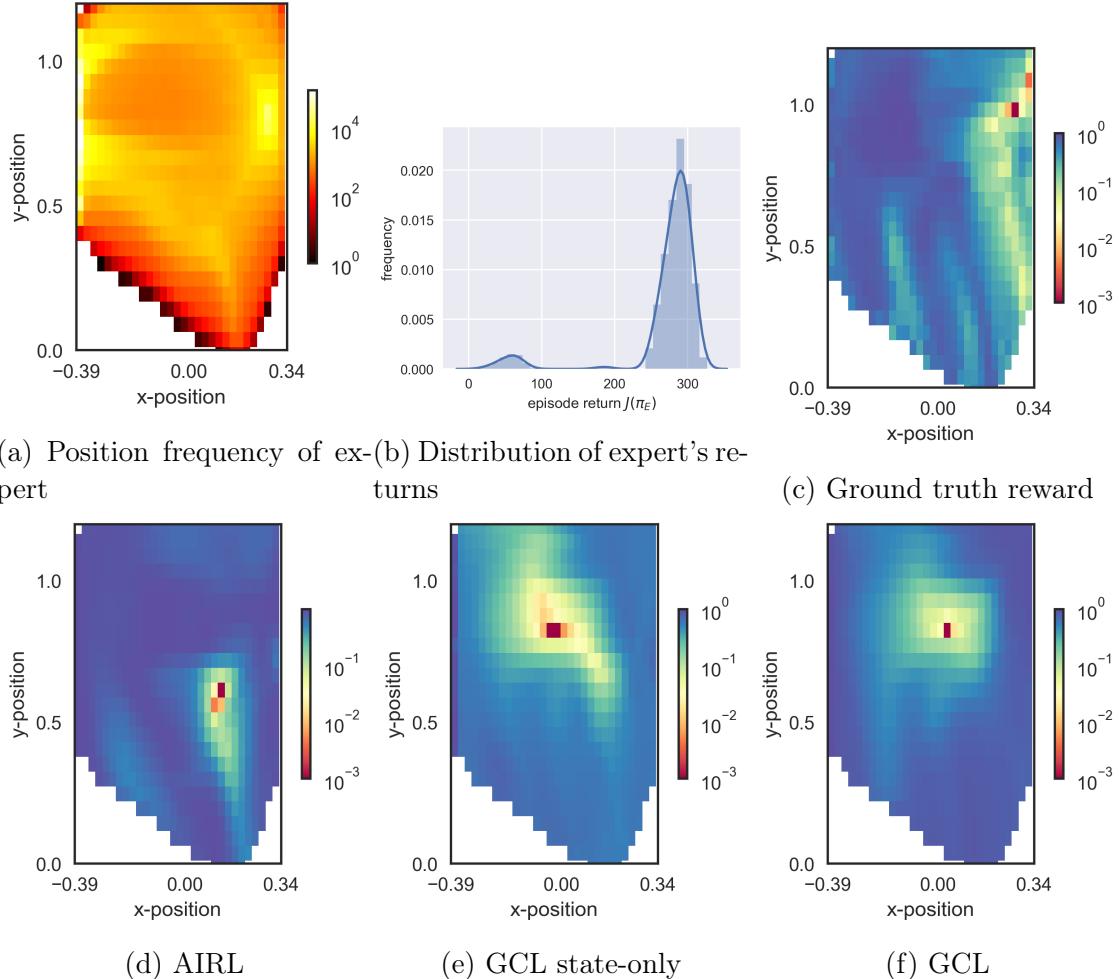


Figure 4.13: Visualization of learned reward functions for Lunar Lander environment. Plots (c)-(f) show the average reward of an expert agent at certain positions in the environment (aggregating over all speeds, angles, and accelerations encountered at this position). In (a) we see a heatmap of where the expert was how often (log-scale) and in (b) we see a distribution of returns achieved by the expert. Rewards are scaled to lie in [0, 1]. Visually, there is not a lot of similarity between the reward functions, but they all include high reward (blue) “paths” to the landing position.

4.2 Robustness of Learned Reward Function

The main objective of this thesis is to evaluate the robustness of the learned reward functions, i.e. how well they generalize to test environments that differ slightly from the training environment. This can be of high importance in real world tasks, e.g. when training a reward function within a simulator and later applying it in a

physical system. Often it cannot be guaranteed that the environment into which the agent gets deployed is and remains exactly the same as the training environment. Optimally, we want learned reward functions to be robust to at least some degree of distributional shift and to generalize well over a range of similar environments.

As described in section 3.5, modified versions of both the Pendulum and the Lunar Lander environment were created to test if reward functions learned on the original environments (based on demonstrations from the original environment) can successfully be applied there as well. In section 4.2.1 we present results for the Pendulum environment, followed by results for Lunar Lander in section 4.2.2.

4.2.1 Pendulum

For the Pendulum environment, the maximum speed possible was changed from the original value of 8 to either be 6 or 10, see section 3.5.1. Figure 4.15 shows that rewards learned with both variants of guided cost learning do not generalize well to the slow Pendulum environment. Their average performance lies far below GAIL performance and exhibits very high variance.

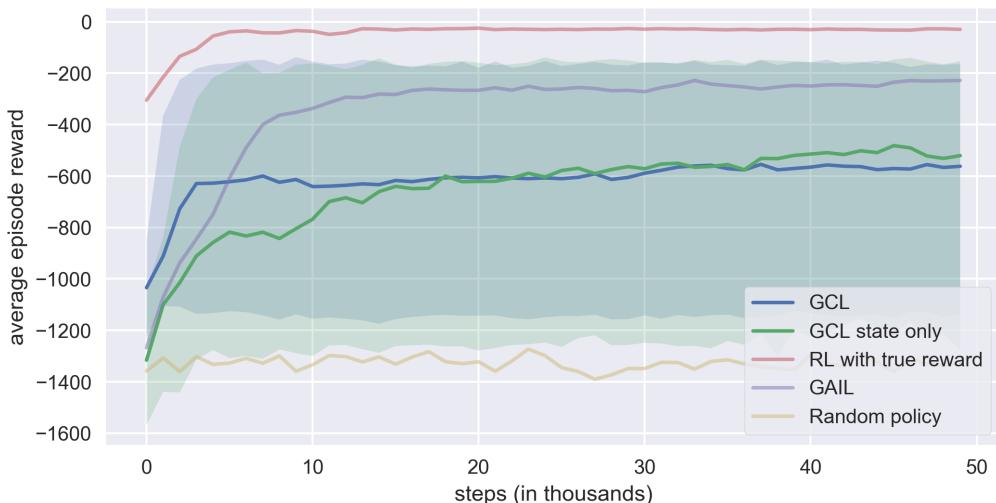


Figure 4.14: GCL results on Pendulum (max-speed 6) environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

Reward functions learned with AIRL, on the other hand, are clearly more robust for this scenario, as figure 4.15 demonstrates. Especially when using mixture batches, the learned policies transfer better than using GAIL and converge very rapidly. However, agents trained on the true reward function are still significantly better, the learned reward functions, while being better than mere behavioral cloning from GAIL, do not generalize entirely to the new scenario. Without the use of mixture batches, there is high variance in results for the first 30,000 training steps. Towards the end of training, average performance is slightly better than GAIL.

For the faster version of the Pendulum environment with a maximum speed of 10, results with GCL are depicted in figure 4.16. The normal version of guided cost learning with state action pairs as domain of the trained reward functions generalizes very poorly to this environment and converges to performance just slightly above



Figure 4.15: AIRL results on Pendulum (max-speed 6) environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

a random policy. The state-only version is about as good as GAIL but with much higher variance. Reducing the domain to only states has made the functions more robust. However, this was not enough to make them achieve performance close to agents trained on the true reward function.

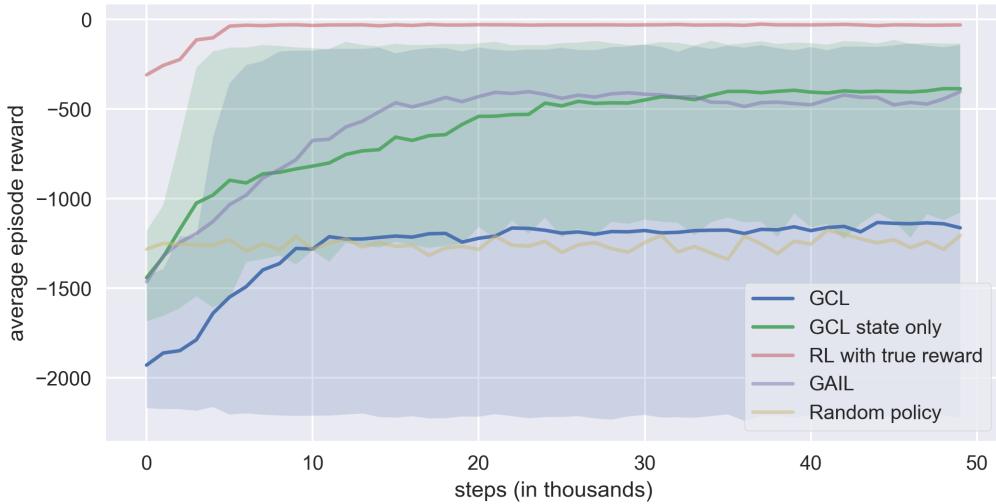


Figure 4.16: GCL results on Pendulum (max-speed 10) environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

As seen in the slow version of the Pendulum environment, AIRL also achieves better robustness performance on the fast variant. The results in figure 4.17 show that both variants of adversarial inverse reinforcement learning achieve higher results than the behavioral cloning baseline GAIL. Using mixture batches leads to high variance results in earlier training steps, but both algorithms converge to a final performance of slightly above -200 .

In summary, AIRL results were significantly more robust to the changes made in Pendulum environments, but the learned reward functions could not match results

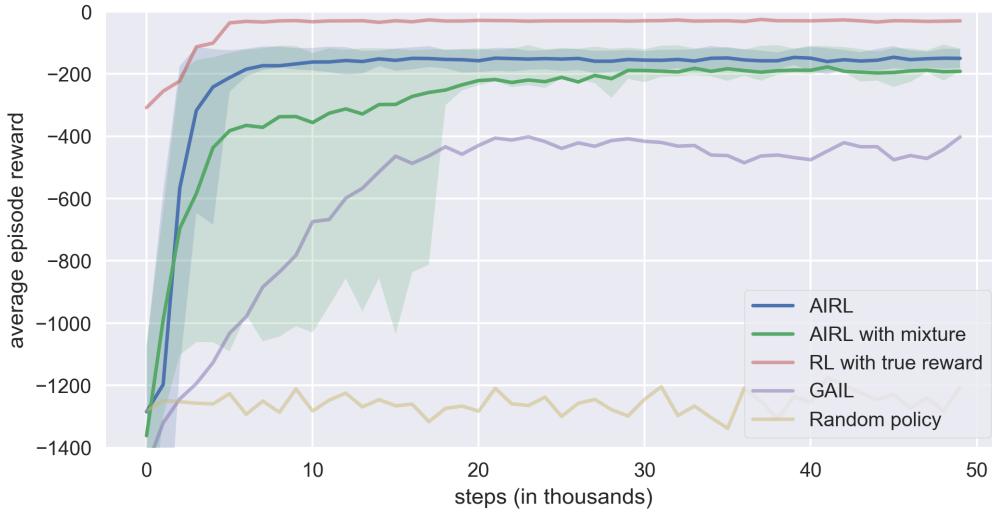


Figure 4.17: AIRL results on Pendulum (max-speed 10) environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

that were achieved with the true reward function. Complete robustness and transferability was not accomplished. However, AIRL results were more robust than behavioral cloning with GAIL.

4.2.2 Lunar Lander

In the modified Lunar Lander environment, several aspects of the game were changed at once, such as the power of both main engine and side engines (section 3.5.2). The results achieved with GCL reward functions on this modified environment are presented in figure 4.18. Clearly, the reward functions based on GCL are not robust to the changes in the modified environment and policies trained on them degrade to poor results, in the state-only case even worse than a random policy.

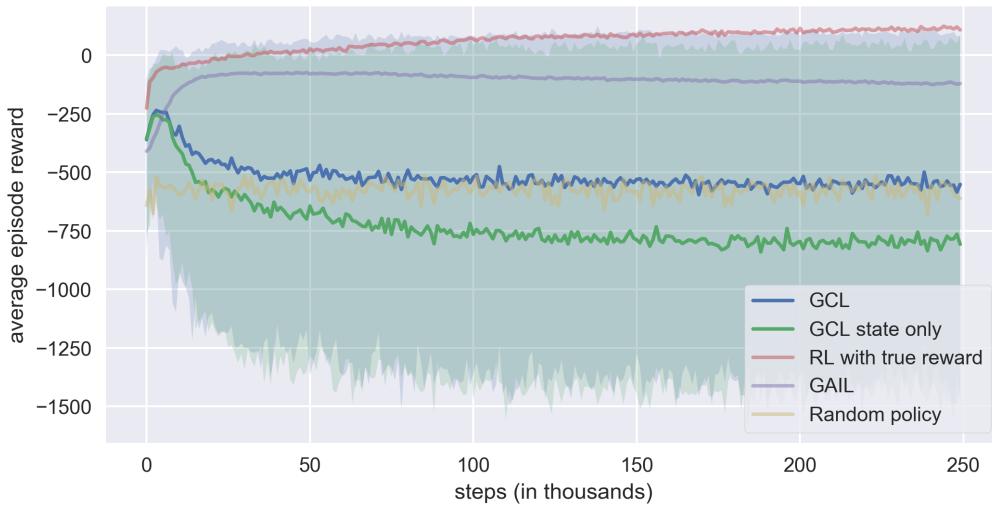


Figure 4.18: GCL results on Lunar Lander (modified) environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

As already observed for the Pendulum environment, reward functions learned with AIRL were more robust to the changes in the modified Lunar Lander environment. Figure 4.19 shows that both variants clearly exceed the robustness of GAIL while falling short of matching the performance of experts trained with the true reward function. The sampling policies π_q trained with AIRL reward functions appear to get stuck in local optima, while policies trained with the true reward function keep increasing their return.

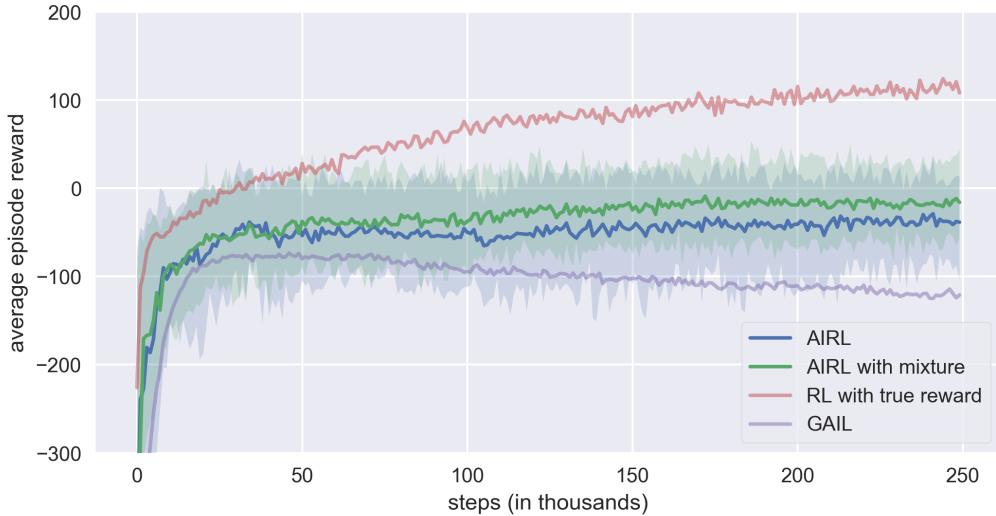


Figure 4.19: AIRL results on Lunar Lander (modified) environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

4.3 Pre-Trained Reward Function

As described in section 3.1.3, one of the modifications proposed in this thesis is to pre-train the reward function’s first hidden layer based on expert demonstrations and data generated from a random agent, such that the first layer already contains useful low-level features at the start of training. Given that training in the Pendulum environment already converges very rapidly without much space for improvement, we limit our experiments for this modification to the Lunar Lander environment.

Figure 4.20 shows the effect of a pre-trained reward network for guided cost learning. There is a small effect of the average training performance being more stable. After 100,000 steps the two variants are almost indistinguishable. Figure 4.21 presents results for AIRL. The effect of pre-training is strong in the first ten iterations of training, but diminishes afterwards. In both cases, there is no lasting effect of pre-training on the performance of the sampling policy π_q .

As section 4.1 showed, even if the sampling policy did not achieve expert-like returns, training a new RL algorithm with the learned reward function estimate can yield significantly better performance. Since the effect of pre-training during training was mostly seen in relatively early iterations, we stop training after 5,000 and 10,000 steps and use the reward estimate at this point to train new policies. The average return achieved by these policies is shown in table 4.3.

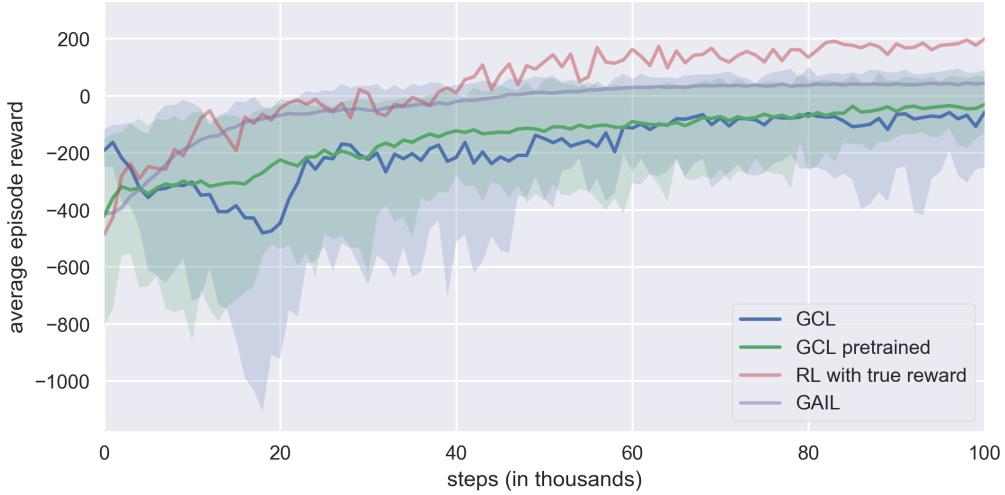


Figure 4.20: Pre-trained GCL results on Lunar Lander environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

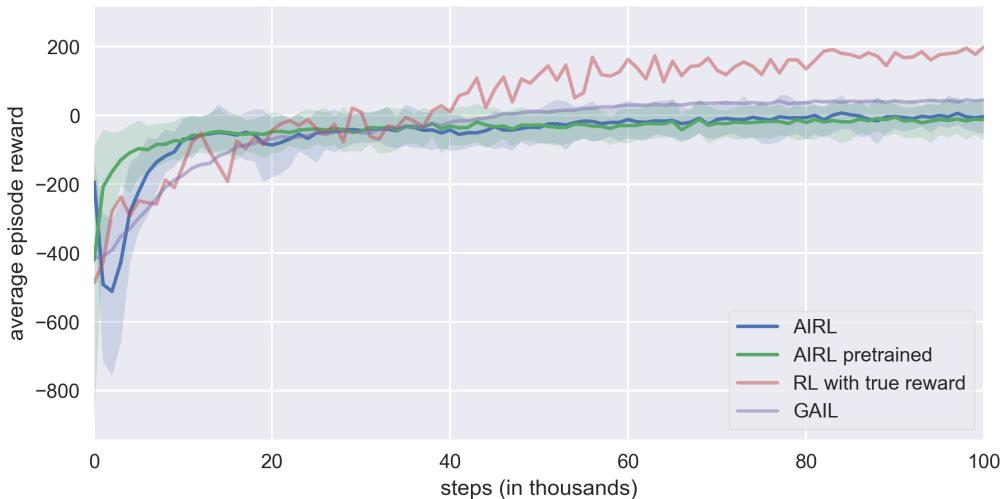


Figure 4.21: Pre-trained AIRL results on Lunar Lander environment. Average results over 50 different random seeds. Shaded area from 10th to 90th percentile.

For guided cost learning, performance after 5000 steps is similar for the default and pre-trained version of the algorithm. However, when comparing performance after 10,000 steps, the pre-trained reward functions already lead to a high average return of above 200, while reward functions without pre-training achieve more than 50 return less on average. For the reward functions trained with AIRL, we can already see a strong positive effect of pre-training after 5,000 steps. In general, AIRL seems to perform worse in early iterations of the training.

training $\hat{\mathcal{R}}$ for	GCL		AIRL	
	5,000 steps	10,000 steps	5,000 steps	10,000 steps
no pre-training	140.1 (120.8)	147.9 (38.5)	87.2 (121.3)	99.9 (124.6)
pre-training	162.6 (106.6)	203.3 (98.2)	138.1 (116.7)	142.9 (121.6)

Table 4.3: Effect of pre-training the reward function on early training stages. Results after training new RL algorithm for 250,000 steps on a reward function that was trained for 5,000 or 10,000 steps. Standard deviations in brackets.

4.4 Expert Data Quantity

The IRL experiments described in the other sections of this chapter were all run with a relatively high number of 1,000 expert demonstrations. In this section, we evaluate the effect of giving fewer demonstrations to the different IRL algorithms.

number of demonstrations	GCL	GCL state-only	AIRL	AIRL with mixture
1	-1293.02	-1288.48	-200.19	-208.95
10	-1286.99	-1253.56	-150.92	-150.91
100	-351.90	-180.62	-149.48	-150.11
1000	-149.24	-148.41	-147.43	-147.09

Table 4.4: Effect of expert data quantity for Pendulum environment. Average return of sampling policy π_q at end of training. Results are averaged over 10 different random seeds.

As table 4.4 reveals, guided cost learning is brittle when not provided with a large number of demonstrations. Negative effects are already noticeable with 100 demonstrations. When further reducing this number to 10 or below, the results are about as good as a random agent. It seems like the state-only variant of guided cost learning is less sensitive to reducing the number of demonstrations. AIRL, on the other hand, only shows a small decrease in average return. Even with just a single demonstration, the average returned achieved with AIRL drops by only about 50 points. In addition to learning more robust rewards, this is another strong argument in favor of using AIRL.

For Lunar Lander, both algorithms are effected negatively when reducing the quantity of expert data available, as shown in table 4.5. Again, AIRL performs better than GCL when provided with a smaller number of demonstrations, except for the extreme case of only training on a single demonstration.

number of demonstrations	GCL	AIRL
1	-121.33	-150.10
10	74.15	109.13
100	156.92	156.58
1000	158.84	174.97

Table 4.5: Effect of expert data quantity for Lunar Lander environment. Average return of sampling policy π_q at end of training. Results are averaged over 10 different random seeds.

4.5 Shaped Reward Loss

For the evaluation of the shaped reward loss (section 3.6), we learned 50 reward functions with GCL and AIRL, as well as generating 50 random reward functions of the same network architecture but with random² parameters θ . We then train reinforcement learning agents on all 150 reward functions and report their average return after training.

Optimally, we would expect a strong correlation between achieving near-optimal return and having a low shaped reward loss when compared with the ground truth reward function. At the same time, it should be possible to distinguish random reward functions from well-performing learned reward functions in most cases just by looking at the shaped reward loss.

Figure 4.22 compares the shaped reward loss for different reward functions and the corresponding performance of policies trained on these functions. While a correlation between high reward and low reward loss is clearly given, there is a quite high number of “outliers” with low reward and low shaped reward loss, or high reward and high shaped reward loss. Random reward functions are not linearly separable from learned reward functions. The correlation between high reward and low shaped reward loss can be used to make an initial assessment about which reward functions perform well, but it is not strong enough to make computing the average return of policies trained on a reward function estimate obsolete.

In conclusion, the shaped reward loss is not reliable enough to confidently predict the quality of a reward function. The theoretic weaknesses addressed in section 3.6 seem to have practical effects and will have to be addressed in future research before considering the use of this metric.

²Initialized with Kaiming He initialization [59].

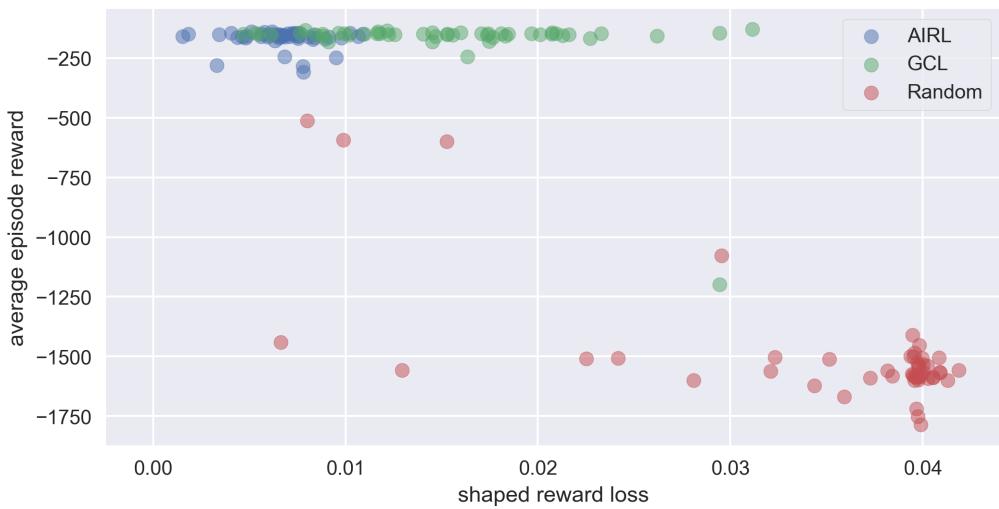


Figure 4.22: Shaped reward loss for Pendulum environment. Comparison of reward functions learned with AIRL and GCL compared to random reward functions (same architecture but random parameters θ).

5 | Discussion

In this chapter, the results and findings of this thesis are reviewed and put into context of previous academic work in the field. This is followed by a discussion of potential future extensions to what has been achieved.

5.1 Conclusions and Contributions

5.1.1 Inverse Reinforcement Learning

We successfully implemented guided cost learning [14] and adversarial inverse reinforcement learning [15] in combination with soft actor critic reinforcement learning agents. The algorithms were applied on the Pendulum and Lunar Lander environments. For Pendulum, we achieved results that surpass previously reported state of the art results for IRL [15]. To the best of our knowledge, Lunar Lander had previously not been solved with IRL.

Interestingly, when taking the learned reward function and using it to train new RL agents from scratch, the learned policies achieve higher return on average than policies that were trained on the original reward function, as well as converging to good results faster. We hypothesize that this is due to the IRL reward functions being shaped in a more learnable way. More experiments in other environments should be done to evaluate the generality of this effect. If confirmed, this would open up a promising avenue for reinforcement learning research: even when a reward function for some task is given, it might be worth-wile to learn a reward function estimate based on demonstrations of some initially trained expert agent, and to use this estimate to train new policies. Potentially, this process could even be iterated.

We found that AIRL needs a lower number of expert demonstrations to converge to strong results than GCL (section 4.4). Guided cost learning easily overfits to single trajectories, if not provided with enough variety of demonstrations.

5.1.2 Proposed Modifications on IRL Algorithms

We combine the soft actor critic algorithm [12] with inverse reinforcement learning, which to the best of our knowledge has not been done before. Using SAC in combination with maximum entropy IRL [11], [54] is a suitable choice as it is sample efficient and fairly robust, while implementing the maximum entropy principle itself and thus being consistent with the model of expert demonstration likelihood.

However, the choice of Gaussian policies as actors makes it impossible to learn multimodal behavior, such as swinging the pendulum clockwise and counter-clockwise with equal probabilities. As the Gaussian policy only has a single mean for each action dimension, it can not learn more than a single mode of behavior for each given state.

Pre-training a reward function’s first hidden layer (section 4.3) led to faster training in early iterations and faster convergence. Reward functions learned without pre-training were able to catch up in later stages of training. The effect of pre-training was stronger for AIRL than for GCL.

The adaptations made to GCL, i.e. using truncated importance sampling and a stable implementation of the log-sum-exp operator, led to more stable results and eliminated numerical errors (section 3.1.6). Using mixture batches (section 3.1.5) showed mixed results¹ empirically. Constructing batches for AIRL directly from the experience replay memory (section 3.1.7) worked well on the Pendulum environment, but was problematic in Lunar Lander where only a small fraction of transitions contains information about the very important last stages of landing.

5.1.3 Robustness Results

The main objective of this thesis is to evaluate the robustness of the learned reward functions when applied to environments with slightly different transition dynamics. Our results weaken the claim made by [15] that state-only reward functions learned with AIRL are “robust to changes in dynamics, enabling us to learn policies even under significant variation in the environment seen during training”. While our results confirm the findings from [15] of AIRL being more robust than GAIL, which is in turn more robust than GCL, the achieved results still often fall far behind the performance achieved with the true reward function.

Lacking robustness to distributional shift is a case of overfitting. However, there are different kinds of overfitting that can make a learned reward function lack robustness to distributional shift. The first type of overfitting is to transition dynamics. If, for given dynamics T , a certain action in a certain state often results in reaching a high reward state, using state action pairs as the reward domain can easily overfit the reward function to favor this particular action. However, in a different MDP, the same action could have a very different distribution over possible next states. As [15] correctly observes, using state-only reward function can successfully remove this type of overfitting.

Besides overfitting to single transition probabilities, there are other possible types of overfitting. Consider a case with more than one mode of optimal behavior, where the expert only demonstrates one of them. The learned reward function will then overfit to the states encountered during expert demonstrations and will assign far too low reward to other optimal trajectories. This was observed in the Pendulum environment, where both directions of swinging the pendulum up (clockwise or counter-clockwise) are optimal, but if only ever seeing clockwise demonstrations

¹No pun intended

from the expert, the reward function strongly overfits to favoring positions on the left side of the environment.

Using state-only reward functions does not make the learned reward function immune to transition dynamic changes. While overfitting to single step actions is not possible anymore, the transition dynamics will often have an effect on the distribution of states in the expert demonstrations. For example, two different paths through an MDP with the same initial state and goal state. Assuming that only the goal state has a positive reward and all other states have a reward of zero, both paths have different probabilities of either reaching the goal state or terminating before. The path with a higher probability of success will be chosen more often during expert demonstrations, and thus states on this path will be assumed to be desirable by the learned reward function. We can now change the transition dynamics in a way that the second path is more probable to succeed. The learned reward function will still encourage the RL agent to take the first path. This kind of overfitting cannot be overcome simply by using state-only AIRL.

We hypothesize that for truly robust reward functions it is necessary to train both on a high variety of expert demonstrations that truly represents the assumed expert demonstration distribution (i.e. demonstrations do not necessarily only come from a single mode of the distribution), and to train on a variety of training environments that sufficiently represent the family of possible environments the agents will be operating in.

More robustness could also be achieved by maintaining a measure of certainty in the reward function. This certainty should be higher for states and transitions that were repeatedly encountered during training, but lower for previously unseen situations. This concept has been introduced under the term *inverse reward design* [72].

5.1.4 Training with Fixed Reward Functions

During training of the reward estimate for Lunar Lander, the performance of the sampling policy π_q , which is trained to optimize the current reward estimate, did not converge to a return that matches the expert policy. However, if taking the learned reward function and keeping it fixed for the training of a new RL agent, the learned policies were often even better than agents of the same type trained on the ground truth reward function.

Apparently, training an agent on a reward function that is updating and changing at the same time makes it significantly harder for the RL agent to converge. There are at least two potential reasons for this. The first problem is that the sampling policy can get stuck in a local optimum which arises from an early (bad) reward estimate. As training progresses, there is less and less exploration and it becomes less likely that the policy will be able to “escape” this local optimum. Another reason could be that due to the coupling between reward function and sampling policy, oscillations might arise that prevent convergence. This problem of oscillations has been observed for GAN, emerging from the dynamics between discriminator and generator and often making convergence difficult [73]. Given that both GCL and AIRL are equivalent to a certain kind of GAN, it can be suspected that this problem needs to be dealt with here as well.

The major performance boost of GCL and AIRL results from training the sampling policy only once and updating the reward function at the same time (as opposed to training many policies and updating the reward function in between). This speed advantage comes at the cost of higher difficulty of converging to truly good behavior during training.

5.2 Future Work

There are multiple possibilities for extensions of the contributions made in this thesis and future work:

For the tested environments, it was sufficient to train a new policy on a learned reward function after it converged during training. However, more challenging tasks might require several iterations of training new RL agents to avoid getting stuck in local optima. A middle ground has to be found between stopping the learning process early and making sure that the sampling policy does converge to near-optimal performance. It could also help to schedule the updates of the reward function such that in later iterations the reward function changes less and less, facilitating the convergence of the sampling policy to good behavior. Both GCL and AIRL come with a large performance advantage compared to previous methods that repeatedly solve new MDPs whenever the reward function was updated. More research should be done on the benefits and drawbacks of both approaches. It should be investigated when it is preferable to re-start the RL algorithm instead of continuing with the current sampling policy.

As discussed in section 5.1.2, the Gaussian policy used in SAC can introduce strong limitations that make it impossible to model complex multi-modal behavior. For future research it would thus be desirable to make changes such that multi-modal behavior can be expressed, e.g. by using a mixture of Gaussians.

The behavioral cloning method GAIL learns a policy based on the expert demonstrations without maintaining an explicit reward function estimate. The learned policies were in many cases more transferable to new environments than policies trained on GCL reward functions. A possible path of future work would be to investigate if GAIL policies can be made more robust by also limiting the discriminator’s input to only states, such as in AIRL.

Overfitting is a common problem encountered in machine learning. Several methods have been developed in related fields to improve generalization of machine learning models and to avoid overfitting. Some of these methods, such as weight decay or dropout, could potentially directly be applied in IRL to obtain more robust reward functions.

One issue encountered when using the experience replay memory to construct training batches for IRL was that batches often did not contain rare but important transitions. To reduce this effect, prioritized experience replay could be applied to increase the probability of sampling transitions with high impact on the likelihood estimate.

Bibliography

- [1] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.
- [2] Maja J. Mataric. Reward functions for accelerated learning. In *Proceedings of the Eleventh International Conference on International Conference on Machine Learning*, ICML'94, pages 181–189, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.
- [3] Abdeslam Boularias, Jens Kober, and Jan Peters. Relative entropy inverse reinforcement learning. In Geoffrey Gordon, David Dunson, and Miroslav Dudák, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 182–189, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.
- [4] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.
- [5] Stuart Russell. Learning agents for uncertain environments (extended abstract). In *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, COLT' 98, pages 101–103, New York, NY, USA, 1998. ACM.
- [6] Andrew Ng and Stuart Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 663–670, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.
- [7] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robot. Auton. Syst.*, 57(5):469–483, May 2009.
- [8] Ivan Bratko, Tanja Urbančič, and Claude Sammut. Behavioural cloning: Phenomena, results and problems. *IFAC Proceedings Volumes*, 28(21):143–149, sep 1995.
- [9] Chelsea Finn, Paul F. Christiano, Pieter Abbeel, and Sergey Levine. A connection between generative adversarial networks, inverse reinforcement learning, and energy-based models. *CoRR*, abs/1611.03852, 2016.
- [10] E. T. Jaynes. Information theory and statistical mechanics. *Phys. Rev.*, 106:620–630, May 1957.

- [11] Brian D. Ziebart, Andrew Maas, J. Andrew Bagnell, and Anind K. Dey. Maximum entropy inverse reinforcement learning. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, pages 1433–1438. AAAI Press, 2008.
- [12] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.
- [13] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.
- [14] Chelsea Finn, Sergey Levine, and Pieter Abbeel. Guided cost learning: Deep inverse optimal control via policy optimization. In *International Conference on Machine Learning*, pages 49–58, 2016.
- [15] Justin Fu, Katie Luo, and Sergey Levine. Learning robust rewards with adversarial inverse reinforcement learning. *arXiv preprint arXiv:1710.11248*, 2017.
- [16] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.
- [17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.
- [18] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.
- [19] Sergey Levine, Peter Pastor, Alex Krizhevsky, Julian Ibarz, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, 37(4-5):421–436, 2018.
- [20] Georgios Theocharous, Philip S Thomas, and Mohammad Ghavamzadeh. Personalized ad recommendation systems for life-time value optimization with guarantees. In *IJCAI*, pages 1806–1812, 2015.
- [21] Engin Ipek, Onur Mutlu, José F Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 39–50. IEEE Computer Society, 2008.
- [22] Andrew G Barto, PS Thomas, and Richard S Sutton. Some recent applications of reinforcement learning. In *Proceedings of the Eighteenth Yale Workshop on Adaptive and Learning Systems*, 2017.

- [23] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.
- [24] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, England, 1994.
- [25] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 16, pages 2094–2100, 2016.
- [26] John N. Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. Technical report, IEEE Transactions on Automatic Control, 1997.
- [27] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.
- [28] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.
- [29] Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement learning with deep energy-based policies. *arXiv preprint arXiv:1702.08165*, 2017.
- [30] Scott Fujimoto, Herke van Hoof, and Dave Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.
- [31] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.
- [32] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058, 2017.
- [33] Rudolf Emil Kalman. When is a linear control system optimal? *Journal of Basic Engineering*, 86(1):51–60, 1964.
- [34] Hanan Shteingart and Yonatan Loewenstein. Reinforcement learning and human behavior. *Current Opinion in Neurobiology*, 25:93 – 98, 2014. Theoretical and computational neuroscience.
- [35] Dylan Hadfield-Menell, Stuart J Russell, Pieter Abbeel, and Anca Dragan. Cooperative inverse reinforcement learning. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 3909–3917. Curran Associates, Inc., 2016.
- [36] Daniel Dewey. Reinforcement learning and the reward engineering principle, 2014.

- [37] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287, 1999.
- [38] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, 2018.
- [39] Jonathan Sorg, Richard L Lewis, and Satinder P Singh. Reward design via online gradient ascent. In *Advances in Neural Information Processing Systems*, pages 2190–2198, 2010.
- [40] Jonathan Sorg, Satinder P Singh, and Richard L Lewis. Internal rewards mitigate agent boundedness. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 1007–1014, 2010.
- [41] Dean A. Pomerleau. Advances in neural information processing systems 1. chapter ALVINN: An Autonomous Land Vehicle in a Neural Network, pages 305–313. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989.
- [42] Rui Camacho and Donald Michie. Behavioral cloning A correction. *AI Magazine*, 16(2):92, 1995.
- [43] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1. ACM, 2004.
- [44] Nathan D Ratliff, J Andrew Bagnell, and Martin A Zinkevich. Maximum margin planning. In *Proceedings of the 23rd international conference on Machine learning*, pages 729–736. ACM, 2006.
- [45] Deepak Ramachandran and Eyal Amir. Bayesian inverse reinforcement learning. *Urbana*, 51(61801):1–4, 2007.
- [46] Brian D. Ziebart, J. Andrew Bagnell, and Anind K. Dey. Modeling interaction via the principle of maximum causal entropy. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 1255–1262, 2010.
- [47] Bernard Michini and Jonathan P How. Improving the efficiency of bayesian inverse reinforcement learning. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 3651–3656. IEEE, 2012.
- [48] Markus Wulfmeier, Peter Ondruska, and Ingmar Posner. Maximum entropy deep inverse reinforcement learning. *arXiv preprint arXiv:1507.04888*, 2015.
- [49] Stuart Armstrong and Sören Mindermann. Impossibility of deducing preferences and rationality from human policy. *CoRR*, abs/1712.05812, 2017.
- [50] Anders Krogh and John A Hertz. A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pages 950–957, 1992.

- [51] Hideki Asoh, Masanori Shiro1 Shotaro Akaho, Toshihiro Kamishima, Koiti Hasida, Eiji Aramaki, and Takahide Kohro. An application of inverse reinforcement learning to medical records of diabetes treatment. In *ECMLPKDD2013 Workshop on Reinforcement Learning with Generalized Feedback*, 2013.
- [52] Jaedeug Choi and Kee eung Kim. Map inference for bayesian inverse reinforcement learning. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 1989–1997. Curran Associates, Inc., 2011.
- [53] Jiangchuan Zheng, Siyuan Liu, and Lionel M Ni. Robust bayesian inverse reinforcement learning with sparse behavior noise. In *AAAI*, pages 2198–2205, 2014.
- [54] Brian D Ziebart, J Andrew Bagnell, and Anind K Dey. The principle of maximum causal entropy for estimating interacting processes. *IEEE Transactions on Information Theory*, 59(4):1966–1980, 2013.
- [55] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 2011.
- [56] Sergey Levine and Vladlen Koltun. Guided policy search. In *International Conference on Machine Learning*, pages 1–9, 2013.
- [57] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *Advances in Neural Information Processing Systems*, pages 4565–4573, 2016.
- [58] David Pfau and Oriol Vinyals. Connecting generative adversarial networks and actor-critic methods. *arXiv preprint arXiv:1610.01945*, 2016.
- [59] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.
- [60] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.
- [61] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [62] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.
- [63] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.
- [64] Hado P van Hasselt, Arthur Guez, Matteo Hessel, Volodymyr Mnih, and David Silver. Learning values across many orders of magnitude. In *Advances in Neural Information Processing Systems*, pages 4287–4295, 2016.

- [65] Edward L Ionides. Truncated importance sampling. *Journal of Computational and Graphical Statistics*, 17(2):295–311, 2008.
- [66] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.
- [67] Siddharth Reddy, Sergey Levine, and Anca Dragan. Shared autonomy via deep reinforcement learning. *arXiv preprint arXiv:1802.01744*, 2018.
- [68] James MacGlashan and Michael L Littman. Between imitation and intention learning. In *IJCAI*, pages 3692–3698, 2015.
- [69] Sergey Levine, Zoran Popovic, and Vladlen Koltun. Feature construction for inverse reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1342–1350, 2010.
- [70] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.
- [71] Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685, 2009.
- [72] Dylan Hadfield-Menell, Smitha Milli, Pieter Abbeel, Stuart J Russell, and Anca Dragan. Inverse reward design. In *Advances in Neural Information Processing Systems*, pages 6765–6774, 2017.
- [73] Jerry Li, Aleksander Madry, John Peebles, and Ludwig Schmidt. Towards understanding the dynamics of generative adversarial networks. *arXiv preprint arXiv:1706.09884*, 2017.