Adversarial Strategy Learning

Daniel Bairamian

Department of Electrical & Computer Engineering

McGill University

Montréal, Québec, Canada

December 15, 2021

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

Master of Science.

Abstract

Imitation learning is the process of learning a policy from observing expert demonstrations. Traditionally, this consists of exposing an agent to sets of state-actions trajectories. The expected result would then be that the agent matches the expert's policy through observation, and learning to behave similarly under comparable conditions and environments. Rather than imitating the expert, what if we wanted to encourage our agent to solve the task at hand, while specifically avoiding the expert's solution? We propose a novel approach to define and leverage the strategy of an expert, and through adversarial examples, encourage the agent to find a new solution to a given task.

To do so, we will not only be looking at the expert's policy, but unlike traditional imitation learning, we will also leverage the information of the Q-function of our expert. This information will determine the likelihood of an observation to be used as an adversarial example. Our experimental results show that our approach is able to extract multiple unique and distinguishable solutions from a given environment.

Résumé

L'apprentissage par imitation est le processus d'apprentissage d'une politique à partir de l'observation de démonstrations d'experts. Traditionnellement, cela consiste à exposer un agent à des ensembles d'états-actions de trajectoires. Le résultat attendu serait alors que l'agent corresponde sa politique a celle de l'expert par observation et qu'il apprenne à se comporter de manière similaire dans des conditions et des environnements comparables. Plutôt que d'imiter l'expert, que ce passerait-il si nous voulions encourager notre agent à résoudre la tâche à accomplir, tout en évitant spécifiquement la solution de l'expert ? Nous proposons une nouvelle approche pour définir et extraire la stratégie d'un expert, et à travers des exemples adversaires, encourager l'agent à trouver une nouvelle solution à une tâche donnée.

Pour ce, nous n'examinerons pas seulement la politique de l'expert, mais contrairement à l'apprentissage par imitation traditionnel, nous exploiterons également les informations de la fonction Q de notre expert. Cette information déterminera la probabilité qu'une observation soit utilisée comme exemple adversaires. Nos résultats expérimentaux montrent que notre approche est capable d'extraire plusieurs solutions uniques et distinguables pour un environnement donné.

Acknowledgements

First and foremost, I am incredibly grateful to my supervisor Derek Nowrouzezahrai, who has been an incredibly supportive mentor.

I am also grateful to Paul Barde for taking the time to share his ideas with me, always giving invaluable feedback and advice, which contributed to the iterative improvement of this project.

Thank you to Tord Jon for sharing his insight on the development of some of the tools created for this project.

I would also like to thank my friends Antonios Valkanas and Ali Shobeiri for their insightful discussions and feedback throughout this project.

Finally I would like to thank my parents Raffi and Nathalie.

Contents

List of Figures

[1] `https://github.com/danielbairamian/Spotter/blob/main/Results/MountainCar.md`

[2] https://github.com/danielbairamian/Spotter/blob/main/Results/FourRooms.md
[3] https://github.com/danielbairamian/Spotter/blob/main/Results/GridWorld.md
[4] https://github.com/danielbairamian/Spotter/blob/main/Results/Pendulum.md
[5] https://github.com/danielbairamian/Spotter/blob/main/Results/MountainCar2.md
[6] https://github.com/danielbairamian/Spotter/blob/main/Results/FourRooms2.md

List of Tables

Chapter 1

Introduction

In order to teach an agent how to interact with its environment, reinforcement learning (RL) is the procedure of training an agent by interacting with the world by performing actions at a given state, and observing the response of the environment. RL agents perform actions based on a probability distribution conditioned on their state, called a policy. After performing an action, the environment will respond to the agent with a reward value and a state update, from which the agent's policy can now sample a new action. The problem then becomes a sequential decision making process where the goal is for the agent to maximize the expected long term return through improving its policy. There are multiple different archetypes of learning algorithms to achieve the goal of RL. Policy-based methods allow the agent to directly learn a stochastic policy function from which the agent could sample actions, such as Trust Region Policy Optimization [3] (TRPO) and REINFORCE [4]. Value-based methods allow the agent to learn the value of a state through value functions, or learn the value of state-action pairs through Q-functions, and then subsequently act by choosing the best action based on these values, such as Deep Q-Network (DQN) [5]. Finally actor-critic

methods combine both previously described archetypes, by learning both a policy and a value function or Q-function, such as Advantage Actor-Critic [6] (A2C) and Soft Actor-Critic [7] (SAC).

Instead of having the agent learn through interacting with the environment, another related approach is imitation learning (IL) [8], where the agent is shown demonstrations of an expert interacting with the same environment. The problem then ends up resembling a traditional supervised machine learning (ML) problem, also referred to as behavioural cloning, as seen in NVIDIA's dataset aggregation algorithm [9] (DAgger). Another more modern approach to imitation learning called generative adversarial imitation learning [10] (GAIL) uses a framework similar to generative adversarial networks [11] (GANs), in order to train an agent to match policies with the expert using a discriminator on the policy outputs.
While these approaches can make the learning much faster, the expert's knowledge would then act as a performance ceiling, meaning the expected behaviour of our agent would be to, at best, match the expert's.

1.1 Contributions

We are interested in solving a given task through multiple unique and distinct strategies. We want to use expert demonstrations as adversarial examples, in order to encourage our agent to find a new solution to the given task. However, blindly avoiding all observed behaviour

is not ideal, as some states and actions could be essential for solving a given task. It is therefore crucial to determine if an observed behaviour is unique to a strategy, or common to all solutions for a given task. We propose a novel framework that identifies the emergent strategy of an expert by analyzing the Q-values along the trajectory of observations. Our expert agent is trained using SAC and therefore uses both actor and critic networks. While imitation learning usually only considers the actor network to learn a new policy, we will also make use of the critic network in order to determine the likelihood of the observation to be considered strategic, and therefore if it should be used as an adversarial example. We will then use this information as an intrinsic reward mechanism for our new agent, which should encourage it find a new unique solution to a task.

1.2 Thesis Overview

Chapter 2 will give an overview of reinforcement learning as well as explain the exploration versus exploitation problem. Chapter 3 will review some of the novel literature in exploration methods, as well as some existing work on learning diverse behaviour. Chapter 4 will present our novel framework's methodology by explaining each component and their role. Chapter 5 will present all of our experimental results. Finally, Chapter 6 will conclude with some discussion and future work ideas.

Chapter 2

Background

The goal is to formally introduce the RL framework, as well as any other relevant background. We will first introduce the concept of Markov Decision Processes in section 2.1. We will then introduce the concept of policy and value functions with their mathematical formulations in section 2.2. Using these definitions we will then present the various ways to accomplish decision making in sections 2.3, 2.4 and 2.5. Furthermore, we will discuss how Imitation Learning is used to transfer knowledge from an expert to an agent in section 2.6. Finally, we will present the exploration vs. exploitation problem in RL in section 2.7, which are the various ways we can encourage our agent to not always act optimally in order to explore its environment.

2.1 Markov Decision Process

The reinforcement learning framework is modeled as what is known as a Markov Decision Process (MDP), which is a mathematical framework that is used for decision making problems. An MDP is defined as:

- a state space $\mathcal{S}$, which is the set of states s the agent can be in

- an action space $\mathcal{A}$, which is the set of actions a the agent can take

- a reward function r where $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, which is the immediate reward r the agent observes when performing an action $a \in \mathcal{A}$, while being at state $s \in \mathcal{S}$

- a transition probability P where $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, which is the probability of transitioning from a given state $s \in \mathcal{S}$ to another state $s' \in \mathcal{S}$ when taking an action $a \in \mathcal{A}$, formally defined as $P(s'|s, a)$

We can therefore define an MDP as the tuple $\mathcal{M} = \{ \mathcal{S}, \mathcal{A}, P, r\}$. Our decision making interaction intervals are referred to as timesteps, where at each timestep t, the agent observes a state $s_t \in \mathcal{S}$, acts on the environment with an action $a_t \in \mathcal{A}$, transitions to state $s_{t+1} \in \mathcal{S}$ with probability $P(s_{t+1}|s_t, a_t)$, while also observing a reward $r_{t+1} = r(s_t, a_t)$ as seen in figure 2.1. One of the key property of an MDP is the memoryless property, known as the markov property. This means, the transition probability should only be conditioned on the current state and action, disregarding any previous information from the past

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

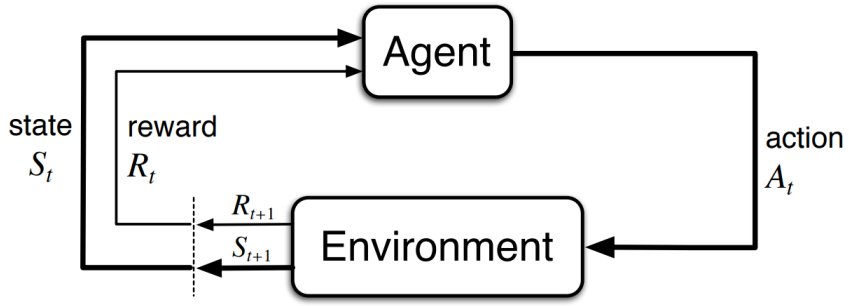


Figure 2.1: The agent-environment interaction in a Markov Decision Process [1]

An example of this would be chess or tic tac toe. For a given board layout, in order to compute what the next optimal move is, we only need to consider the current state. The history of the board does not give us any additional information on what the next best move is. In figure 2.2, we can see an example of the markovian property on a tic tac toe board.

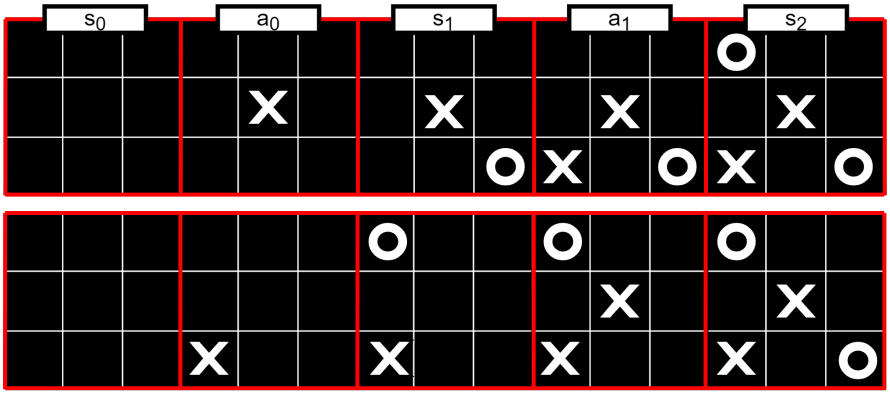


Figure 2.2: Tic Tac Toe: Markovian property example

At state s_2, both boards look exactly identical, however the chain of events that lead to this state are completely different. Still, the information needed to compute the next best move a_2 is exactly the same for both scenarios, namely only s_2. The history of events is therefore irrelevant to the computation of our next decision.

2.2 Policy and Value Functions

The goal of RL is to maximize the expected long term return. The agent will interact with the environment until the latter responds with a signal indicating the end of the task. We call the sequence of states from the initial state to the terminal state an episode, and the length of an episode is called a horizon. The return of a state in an episode of infinite horizon is defined as G_t, which is the sum of discounted reward from the state s_t until the end of the episode

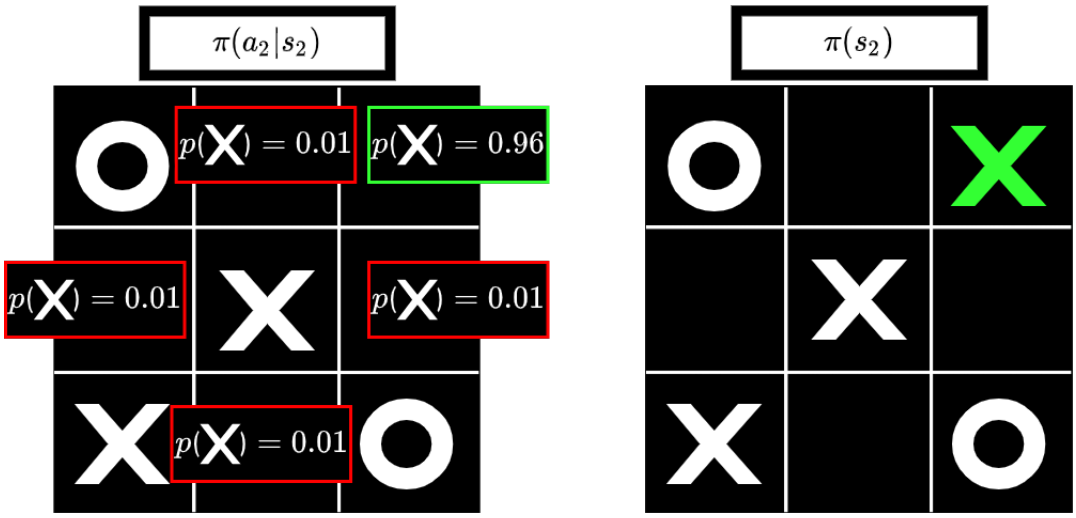
$$\begin{aligned} G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \dots \\ G_t &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \ . \end{aligned} \tag{2.2}$$

Here $\gamma \in [0, 1]$ is called the discount factor, which is used to balance our agent's long term planning. Setting γ closer to zero will make it very myopic with almost no interest in the delayed long term reward, while setting γ closer to one will assign high importance to later reward, suitable for long term planning. In practice, this value is generally really close to one, $\gamma \approx 0.99$. The goal is then to maximize the expected return $\mathbb{E}[G_t]$. From eq. (2.2),

$$\begin{aligned} G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \dots \\ G_t &= r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + \dots) \\ G_t &= r_{t+1} + \gamma G_{t+1} \ . \end{aligned} \tag{2.3}$$

2.2.1 Policy Functions

We define $\pi(a|s)$ as a stochastic policy function, which is a conditional probability distribution over actions $a \in \mathcal{A}$ given a state $s \in \mathcal{S}$. Similarly, we denote $\pi(s)$ as a deterministic policy function. Rather than returning a distribution over actions, this policy will return a single action. An example of both types of policy functions can be seen in figure 2.3



(a) Tic Tac Toe: Stochastic policy **(b)** Tic Tac Toe: Deterministic policy

Figure 2.3: Tic Tac Toe policy examples

2.2.2 Value Functions

We denote $V^{\pi}(s) : \mathcal{S} \rightarrow \mathbb{R}$ as the value function, which takes as input a state s and outputs the expected return from the input state when taking actions with respect to the policy

π

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t | s_t = s] \ . \tag{2.4}$$

Computing the value of a state requires to then compute an infinite sum. However, we can rewrite the equation of the value function using eq. (2.1), (2.2), (2.3) and (2.4), to transform the problem into a tractable recursive definition, commonly known as Bellman equations [12]

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{\pi}[G_t | s_t = s] \\ &= \mathbb{E}_{\pi}[r_{t+1} + \gamma G_{t+1} | s_t = s] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} P(s'|s,a)[r(s,a) + \gamma \mathbb{E}_{\pi}[G_{t+1} | s_{t+1} = s']] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} P(s'|s,a)[r(s,a) + \gamma V^{\pi}(s')] \ . \end{aligned} \tag{2.5}$$

Simply put, in order to estimate the value of a state s, we need to consider all possible actions weighted by their respective probabilities $\sum_{a \in \mathcal{A}} \pi(a|s)$. For each action, we then need to consider all possible state s' we can end up in, weighted by their respective transition probability $\sum_{s' \in \mathcal{S}} P(s'|s,a)$. Finally we need to combine the immediate reward we get for taking the current action we're considering a, as well as the discounted value of the next state s' : $[r(s,a) + \gamma V^{\pi}(s')]$.

Similar to attributing a value to a single state s, we can also assign a value to a state-action pair (s, a). We denote $Q^\pi(s, a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ as the Q function, which takes as input a state s and action a and outputs the expected return from the input state when taking the input action

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t | s_t = s, a_t = a] \ . \tag{2.6}$$

Similarly to the value function, we can transform the problem of evaluating the Q function into a recursive formulation, with the following Bellman equation

$$\begin{aligned} Q^\pi(s, a) &= \mathbb{E}_\pi[G_t | s_t = s, a_t = a] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma G_{t+1} | s_t = s, a_t = a] \\ &= \sum_{s' \in \mathcal{S}} P(s'|s, a)[r(s, a) + \gamma \mathbb{E}_\pi[G_{t+1} | s_{t+1} = s'] \\ &= \sum_{s' \in \mathcal{S}} P(s'|s, a)[r(s, a) + \gamma V^\pi(s')] \ . \end{aligned} \tag{2.7}$$

We can also see from eq. (2.5) and (2.7) that the relationship between a value function and a Q function is the following

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \cdot Q^\pi(s, a) = \mathbb{E}_\pi[Q^\pi(s, a)] \ . \tag{2.8}$$

Finally, we denote $A^\pi(s, a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ as the advantage function. This gives us an estimate

of the value of an action compared to the other actions, on average when sampled from the policy π

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \ . \tag{2.9}$$

2.3 Value Function Based Methods

As previously stated, the goal of RL is to maximize the expected long term return. This directly translates to then learning the optimal policy, denoted by π^*. One approach is to consider a deterministic policy $\pi^*(s)$, where the action taken with respect to the highest return from our value function, known as the optimal value function $V^*(s)$, or from our Q function, known as the optimal Q function $Q^*(s,a)$, also referred to as greedy policy

$$V^*(s) = \max_{\pi} V^{\pi}(s) = \max_{a} \sum_{s' \in \mathcal{S}} P(s'|s,a)[r(s,a) + \gamma V^*(s')] \tag{2.10}$$

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) = \sum_{s' \in \mathcal{S}} P(s'|s,a)[r(s,a) + \gamma \max_{a'} Q^*(s',a')] \ . \tag{2.11}$$

2.3.1 Dynamic Programming

The first approach to find the optimal policy π^* is a set of algorithms from the dynamic programming (DP) family, that directly make use of our Bellman equations, namely Policy Iteration (Algorithm 1) and Value Iteration (Algorithm 2) [1] . The convergence of V^π

Algorithm 1: Policy Iteration

Step 1: Initialize V^π and π arbitrarily;
Step 2: Policy Evaluation;
while $V^\pi(s)$ *has not converged* **do**
    for *all* $s \in \mathcal{S}$ **do**
        $v \leftarrow V^\pi(s)$; `/* Save previous value of` $V^\pi(s)$ `*/`
        $V^\pi(s) \leftarrow \sum_{s' \in \mathcal{S}} P(s'|s, \pi(s))[r(s, \pi(s)) + \gamma V^\pi(s')]$;
Step 3: Policy Improvement;
for *all* $s \in \mathcal{S}$ **do**
    $a \leftarrow \pi(s)$; `/* Save previous value of` $\pi(s)$ `*/`
    $\pi(s) \leftarrow \arg\max_a \sum_{s' \in \mathcal{S}} P(s'|s, a)[r(s, a) + \gamma V^\pi(s')]$;
    if $a \neq \pi(s)$ **then**
        goto Step 2 ; `/* Policy is not stable, go back to Step 2 */`
return V^π and π

Algorithm 2: Value Iteration

Initialize V^π arbitrarily;
while $V^\pi(s)$ *has not converged* **do**
    for *all* $s \in \mathcal{S}$ **do**
        $v \leftarrow V^\pi(s)$; `/* Save previous value of` $V^\pi(s)$ `*/`
        $V^\pi(s) \leftarrow \max_a \sum_{s' \in \mathcal{S}} P(s'|s, a)[r(s, a) + \gamma V^\pi(s')]$;
$\pi(s) = \arg\max_a \sum_{s' \in \mathcal{S}} P(s'|s, a)[r(s, a) + \gamma V^\pi(s')] \quad \forall s \in \mathcal{S}$;
return V^π and π

is determined by the maximum observed changes between $V^\pi(s)$ and v. If all observed

values are less than some small threshold, then we convisder the value function to have converged.

2.3.2 Temporal Difference Learning

One of the issues with dynamic programming is that we require a probability model of the world in order to compute the expectation of the entire distribution of next states and rewards, which we may not always have, and could be intractable for complex environments. Another approach is instead to learn directly from interacting with the environment, without requiring a model of the world, known as *model free* methods. Temporal Difference learning (TD) [13] is a class of learning algorithms which are model free, that learn by iteratively updating a current estimate of the value function, known as bootstrapping. We initialize our value function arbitrarily, e.g. $V^\pi(s) = 0 \; \forall \; s \in \mathcal{S}$, and approximate the infinite horizon return G_t, defined in eq. (2.2) with an n-step return $G_{t:t+n}$ defined as

$$\begin{aligned} G_{t:t+n} &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n V^\pi(s_{t+n}) \\ &= \sum_{k=0}^{n-1} [\gamma^k r_{t+k+1}] + \gamma^n V^\pi(s_{t+n}) \; . \end{aligned} \tag{2.12}$$

We therefore have an estimate of how good our approximated return $G_{t:t+n}$ is, by computing the TD error $\delta_{t+n} = G_{t:t+n} - V^\pi(s_t)$, then use this error to update our value function

estimation. This is known as n-step TD

$$V^{\pi}(s_t) \leftarrow V^{\pi}(s_t) + \alpha[G_{t:t+n} - V^{\pi}(s_t)] \ . \tag{2.13}$$

Where $\alpha \in [0, 1]$ is a learning rate. Note that setting $n = \infty$ no longer approximates the n-step return, but rather gets the exact infinite horizon, known as ∞-step TD or Monte Carlo (MC) methods. However waiting for an episode to terminate is not ideal, and often not practical for complex environments. The simplest case is the one-step TD method, known as TD(0), with the following update rule for $V^{\pi}(s_t)$

$$\begin{aligned} V^{\pi}(s_t) &\leftarrow V^{\pi}(s_t) + \alpha[G_{t:t+1} - V^{\pi}(s_t)] \\ &\leftarrow V^{\pi}(s_t) + \alpha[r_{t+1} + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)] \ . \end{aligned} \tag{2.14}$$

We can also use the same idea to evaluate a Q function using a one step return

$$Q^{\pi}(s_t, a_t) \leftarrow Q^{\pi}(s_t, a_t) + \alpha[r_{t+1} + \gamma Q^{\pi}(s_{t+1}, a_{t+1}) - Q^{\pi}(s_t, a_t)] \ . \tag{2.15}$$

This idea is the basis of the famous one-step TD control algorithm, known as Q-learning [14] (Algorithm 3).

Algorithm 3: Q-Learning

Initialize Q^π arbitrarily, $\alpha \in [0, 1]$, small $\epsilon > 0$;
for *each episode* **do**
    for *each step of the episode* t **do**
        Choose a_t from $Q^\pi(s, a)$ $(\epsilon - greedy)$; `/* more on this in section 2.7 */`
        Take action a_t, observe $r_{t+1} = r(s_t, a_t)$ and s_{t+1};
        $Q^\pi(s_t, a_t) \leftarrow Q^\pi(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q^\pi(s_{t+1}, a) - Q^\pi(s_t, a_t)]$;
        $s_t \leftarrow s_{t+1}$;
        if s_t *is terminal* **then**
            go to next episode ;
return Q^π

2.4 Function Approximators

Thus far, in order to estimate the value of a state or state-action pair, we would have had to see that specific input beforehand multiple times in order to get an accurate estimation. This is not only intractable for high dimensional environments, but also impossible for environments with continuous state spaces, as opposed to discrete state space. Similarly to state spaces, our action space can also be either discrete or continuous. Therefore, we use function approximators to return an estimate of our value, Q and policy functions. We denote V^π_ϕ and Q^π_ϕ as a value function and Q function parametrized by the set of learnable parameters ϕ respectively. Similarly, we denote π_θ as a policy function parametrized by the set of learnable parameters θ. The idea of function approximators is to use neighbouring values already seen to estimate an input value that we may have never encountered. The accuracy of this estimation will be directly linked to the power of the approximator's

ability to generalize. Recent advances in deep learning [15] have made neural networks the prime candidate for function approximaton, due to their ability to generalize but most importantly due to the fact that they are differentiable, and thus enabling the use of back-propagation [16] for learning.

2.5 Policy Optimization

Rather than deterministically taking actions according to the value or Q function as seen in section 2.3, a new class of methods known as policy gradients [17], aim at directly optimizing the policy function. We will also see how these methods can be greatly improved by reusing the concept of bootstrapping seen in section 2.3.2, known as Actor-Critic (AC) [18] methods.

2.5.1 Policy Gradients

In policy gradients , we consider a parametrized stochastic policy π_θ. The goal is the same as before, namely maximizing the expected return. Here we denote our objective function $J(\pi_\theta)$ as the expected return of some trajectory when taking actions with respect to to the policy π_θ

$$J(\pi_\theta) = V^{\pi_\theta}(s_0) = \mathbb{E}_{\pi_\theta}[G_t | s_t = s_0] \ . \tag{2.16}$$

Where $[G_t|s_t = s_0]$ is the total expected return of the entire episode. The aim is the to update the parameters θ of our policy

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta) \ . \tag{2.17}$$

Given an episode $\tau = (s_0, a_0, s_1, a_1, \ldots)$, in order to compute our policy gradient updates, we need to first compute the log probability of a trajectory $\log P(\tau|\theta)$, as well as its gradient $\nabla_\theta \log P(\tau|\theta)$ with respect to our parameters θ. We denote $p(s_0)$ as the probability of our trajectory starting in the observed state s_0. Note that it does not depend on our policy, but rather on the environment itself. An environment does not necessarily have a fixed starting point, and therefore we need to consider this in our derivation

$$\begin{aligned}
P(\tau|\theta) &= p(s_0) \prod_{t=0}^{\infty} P(s_{t+1}|s_t, a_t) \pi_\theta(a_t|s_t) \\
\log P(\tau|\theta) &= \log p(s_0) + \sum_{t=0}^{\infty} \log P(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t) \\
\nabla_\theta \log P(\tau|\theta) &= \cancel{\nabla_\theta \log p(s_0)} + \sum_{t=0}^{\infty} \cancel{\nabla_\theta \log P(s_{t+1}|s_t, a_t)} + \nabla_\theta \log \pi_\theta(a_t|s_t) \\
&= \sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) \ .
\end{aligned} \tag{2.18}$$

Finally, remember the log-trick:

$$\begin{aligned} \nabla_\theta \log P(\tau|\theta) &= \frac{\nabla_\theta P(\tau|\theta)}{P(\tau|\theta)} \\ \nabla_\theta P(\tau|\theta) &= P(\tau|\theta)\nabla_\theta \log P(\tau|\theta) \ . \end{aligned} \tag{2.19}$$

Putting eq. (2.16), (2.18) and (2.19) together, we get the following derivation for policy gradients

$$\begin{aligned} \nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathbb{E}_{\pi_\theta}[G_t|s_t = s_0] \\ &= \nabla_\theta \int_\tau P(\tau|\theta)[G_t|s_t = s_0] \\ &= \int_\tau \nabla_\theta P(\tau|\theta)[G_t|s_t = s_0] \\ &= \int_\tau P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)[G_t|s_t = s_0] \\ &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log P(\tau|\theta)[G_t|s_t = s_0]] \\ &= \mathbb{E}_{\pi_\theta}[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t)[G_t|s_t = s_0]] \ . \end{aligned} \tag{2.20}$$

An example of a policy gradient algorithm is REINFORCE [4] (Algorithm 4).

Algorithm 4: REINFORCE

Initialize π_θ arbitrarily, $\alpha \in [0, 1]$;
for *each episode* **do**
  Generate episode with respect to π_θ;
  for *each step of the episode* t **do**
    $G_t \leftarrow \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$;
    $\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta \log \pi_\theta(a_t|s_t)$; `/* Update the parameters of` π_θ `*/`
return π_θ

2.5.2 Actor-Critic

We have seen before how estimating the infinite horizon return G_t can be approximated with temporal difference learning. The same can applied in Actor-Critic methods. Rather than using G_t, we can use an n-step return estimation $G_{t:t+n}$ instead. The simplest case is the one-step actor-critic algorihtm (Algorithm 5). Instead of directly learning a policy function, we learn both a value function V_ϕ^π, and a policy function π_θ. The policy function is referred to as the actor, as it is responsible for taking actions. The value function is referred to as the critic, as it is responsible for estimating the current state, and then transferring this information to the actor for the latter to use it to update its policy. We therefore have a way to optimize our policy through gradients without having to run episodes to completion.

Algorithm 5: One-step Actor-Critic

Initialize π_θ, V_ϕ^π arbitrarily and two learning rates $\alpha^\theta, \alpha^\phi \in [0, 1]$;
for *each episode* **do**
  for *each step of the episode* t **do**
    $a_t \sim \pi_\theta$;
    Take action a_t, observe $r_{t+1} = r(s_t, a_t)$ and s_{t+1};
    $\delta \leftarrow r_{t+1} + \gamma V_\phi^\pi(s_{t+1}) - V_\phi^\pi(s_t)$;
    $\phi \leftarrow \phi + \alpha^\phi \delta \nabla_\phi V_\phi^\pi(s_t)$; `/* Update the parameters of` V_ϕ^π `*/`
    $\theta \leftarrow \theta + \alpha^\theta \gamma^t \delta \nabla_\theta \log \pi_\theta(a_t|s_t)$; `/* Update the parameters of` π_θ `*/`
    $s_t \leftarrow s_{t+1}$;
    if s_t *is terminal* **then**
      go to next episode ;
return π_θ,V_ϕ^π

2.6 Imitation Learning

So far, we have seen that in order to learn an optimal policy π^*, we either need to know the model of our environment (DP), or directly interact with it (TD). Imitation learning is an alternative framework of learning a policy function by observing another agent's actions, referred to as the expert. The problem then becomes a classical supervised learning problem, where the agent is exposed to some dataset of state and actions $\mathcal{D} = \{s_1, a_1, s_2, a_2, \ldots, s_N, a_N\}$ extracted from the expert, who we assume operates under the optimal policy. The expected end result would then be that both the agent's and the expert's policy will be identical, and since the expert's policy is optimal, our agent's policy will also be optimal. A popular example of this is the DAgger [9] algorithm from NVIDIA (Algorithm 6), where they use a human-in-the-loop approach to iteratively expand their dataset $\mathcal{D}$.

Algorithm 6: DAgger: Dataset Aggregation

Initialize π_θ arbitrarily;
collect dataset $\mathcal{D} = \{s_1, a_1, s_2, a_2, \ldots, s_N, a_N\}$ from an expert ;
while *training* **do**
    Fit π_θ to the dataset $\mathcal{D}$;
    Using π_θ, collect new dataset of states only $\mathcal{D}_{\pi_\theta} = \{s_1, s_2, \ldots, s_M\}$;
    Ask a human to label each action a_t, $\mathcal{D}_{\pi_\theta} = \{s_1, a_1, s_2, a_2, \ldots, s_M, a_M\}$;
    Aggregate both datasets, $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_{\pi_\theta}$;
return π_θ

Another popular approach to imitation learning takes inspiration from recent advances in

generative methods in deep learning with a framework inspired by Generative Adversarial Networks [11] (GAN), called Generative Adversarial Imitation Learning [10] (GAIL, Algorithm 7). In the paper, the authors define a discriminator D parametrized by weights w, which takes as input a state-action pair (s, a), and returns a value between zero and one, $D_w : \mathcal{S} \times \mathcal{A} \rightarrow (0, 1)$. The goal of this discriminator is to determine if the input state-action pair came from the agent, or the expert. An output closer to zero means that the discriminator thinks that the input pair belongs to the expert's policy, while a value of one means that it belongs to the agent's policy. Since this is a continuous value, an output of 0.5 would indicate that the discriminator has no clue weather or not the observed input was from the agent or the expert, meaning our agent has successfully fooled the discriminator into making it believe it is acting like the expert, and therefore learned a policy which is optimal.

Algorithm 7: GAIL: Generative Adversarial Imitation Learning

Initialize π_θ, D_w arbitrarily;
Sample expert trajectories τ_E from the expert policy π_E;
while *training* **do**
    Sample agent trajectory $\tau_\theta \sim \pi_\theta$;
    Calculate discriminator loss function
     $L(D_w) = \mathbb{E}_{\tau_\theta}[\log(D_w(s, a))] + \mathbb{E}_{\tau_E}[\log(1 - (D_w(s, a))]$;
    Update w with gradient descent with respect to $\nabla_w L(D_w)$;
    for *each* (s_t, a_t) *pair in* τ_θ **do**
        $r_t = -\log(D_w(s_t, a_t))$
    Update θ with policy gradients ;
return π_θ

The reward observed by the agent is maximized as $D_w(s, a)$ tends towards zero, meaning the discriminator thinks the agent is acting like the expert. When the agent's policy matches the expert's policy at optimality, the discriminator's output will be 0.5 for all input pairs (s, a), as it will not be able to differentiate between the two functions.

2.7 Exploration vs. Exploitation

Thus far, all of our efforts were dedicated to finding the optimal policy π^*. However, having our agent always act greedily is not desirable, as we want our agent to explore its environment in order to potentially find a greater source of reward to what's currently known. This is referred to as the exploration versus exploitation dilemma. We briefly mentioned the simplest form of exploration when we presented Q-learning (Algorithm 3), which is ϵ-greedy exploration (Algorithm 8). This technique consists of taking a random action with probability ϵ, or taking the greedy action with probability $1 - \epsilon$.

Algorithm 8: ϵ-greedy exploration

```
p ~ U(0, 1);              /* Draw a random number p uniformly between (0, 1) */
if p < ε then
    a_t ← random ∈ A ;          /* take a random action with probability ε */
else
    a_t ← arg max_a Q^π(s_t, a) ;    /* take greedy action with probability 1 − ε */
```

Another approach to exploration is to let the agent sample actions according to a softmax

distribution over the Q values observed, regulated by a parameter τ called the temperature, known as Boltzmann exploration (Algorithm 9). The temperature will control the amount of exploration or exploitation we want our agent to do, by directly affecting the distribution the agent is sampling from. Setting $\tau = \infty$ will make the action sampling uniform, while setting $\tau = 1$ will make the action sampling directly proportional to the Q values.

Algorithm 9: Boltzmann exploration

for *each possible action* a_t *at state* s_t **do**

$$p(a_t) = \frac{\exp \frac{Q^{\pi}(s_t,a_t)}{\tau}}{\sum_{a \in \mathcal{A}} \exp \frac{Q^{\pi}(s_t,a)}{\tau}};$$

Sample a_t according to the probability calculated ;

2.7.1 Intrinsic vs. Extrinsic Reward

Another form of exploration can be achieved through intrinsic reward. Unlike extrinsic reward, which is the reward the agent observes from the environment $r_t = r(s_t, a_t)$, we can encourage exploration through an internal source of bonus reward. One simple example would be to give bonus reward based on the amount of time an agent visited a certain state, known as count-based exploration (Algorithm 10). For this method we need to keep a counter of the number of total states visited n, a counter that keeps track of the number of visit for each state $N(s)$, and finally some reward function $\mathcal{B}(N(s), n)$ that will give a reward based on these two counters. There are many ways to design this reward function, some popular choices are the following:

- $B(N(s), n) = \sqrt{\frac{2\log(n)}{N(s)}}$
- $B(N(s), n) = \sqrt{\frac{1}{N(s)}}$ [19]
- $B(N(s), n) = \frac{1}{N(s)}$ [20]

Algorithm 10: count-based exploration

$n = 0$; `/* total states visited counter */`
$N(s) = 0$ for all $s \in \mathcal{S}$; `/* per state visit counter */`
for *each step of the episode* t **do**
  n += 1;
  $N(s_t)$ += 1;
  $r_t = r(s_t, a_t)$;
  $r_{intrinsic} = \mathcal{B}(N(s_t), n)$;
  agent observed reward $r_t + r_{intrinsic}$

As a state s is visited more frequently, $N(s)$ will increase, and therefore the reward observed at a subsequent visit will decrease. This encourages our agent to explore early on, then exploit as time goes on.

Chapter 3

Literature Review

We will present some of the relevant literature required to understand our method. We will first introduce how exploration can be achieved for complex and continuous state spaces with Random Network Distillation (RND) [21], by using a set of two neural networks that generate a source of intrinsic bonus reward similar to count based exploration. We will then introduce entropy regularized RL, a framework that uses entropy as a mechanism to boost exploration through regularization of the policy function, which has seen a lot of popularity recently with the famous paper Soft Actor Critic (SAC) [7]. Finally, we will see how an agent can learn various skills without an extrinsic reward signal, with purely unsupervised exploration, from the paper "Diversity Is All You Need" (DIAYN) [2].

3.1 Random Network Distillation

As stated in section 2.7, we can use count based exploration as an intrinsic source of reward to encourage exploration. However, just like our motivation for function approximators seen in section 2.4, when dealing with continuous state spaces, counting the

occurrence of a sate becomes impractical. The paper "Exploration by Random Network Distillation" [21] proposes a framework using two neural networks that mimics the idea of count based exploration. This method consists of two architecturally identical networks, randomly initialized separately; which take as input a state s_t, and output a random vector of arbitrary size $\mathbb{R}^k$. The first one is fixed, called the target network $f : \mathbb{R}^{|S|} \to \mathbb{R}^k$, the second one is called the predictor network $\hat{f} : \mathbb{R}^{|S|} \to \mathbb{R}^k$. The predictor network $\hat{f}$ is trained to minimize the mean squared error (MSE) between its output and the output of the target network f : $MSE(\hat{f}, f) = ||\hat{f}(s;\theta) - f(s)||^2$, by gradient descent with respect to the parameters of the predictor network θ. The target network f is frozen on initialization, meaning its weights never get updated. A visualization of this framwork can be seen in figure 3.1.

Since both networks have been randomly initialized separately, their outputs will be seemingly random for any given input. However, the predictor network is being trained to match the target network, therefore with each subsequent observation of a datapoint, the MSE will be reduced. The value of the MSE will then give an estimate of how often a specific state s has been seen. States that have not been observed much will have a relatively high MSE, while states that have been observed more frequently will have a relatively low MSE. This idea is therefore directly analogous to the count based exploration we have seen before, where the MSE can be directly used as an intrinsic source of bonus

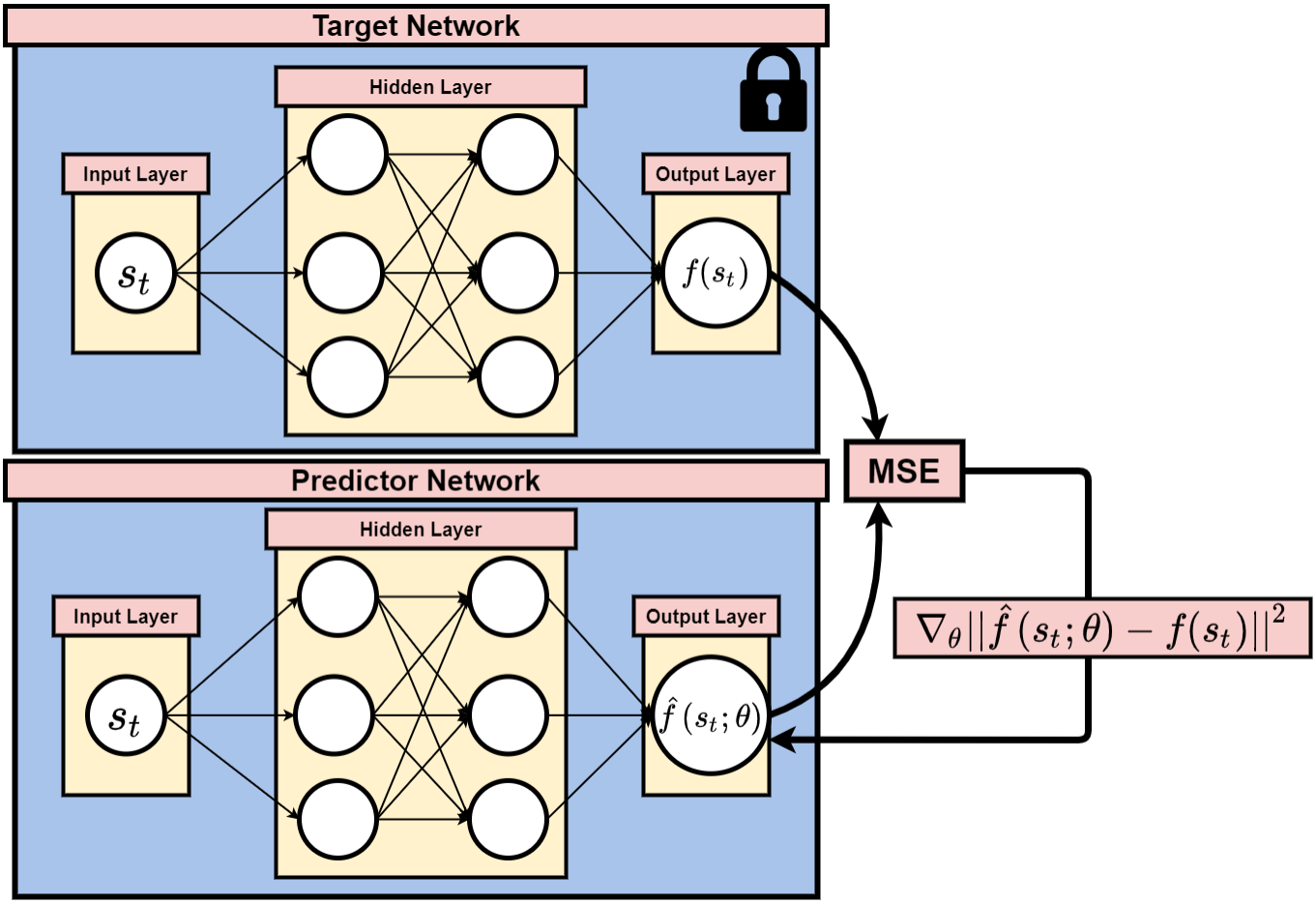


Figure 3.1: Random Network Distillation Architecture

reward.

Since some sort of unsupervised learning is occurring on the data, this framework is somewhat similar to that of an autoencoder [22] (AE). The MSE error between the two networks can be seen as the reconstruction error observed in AEs, however an important difference to note is that this method does not aim to learn some efficient encoding or representation of the data

as in AEs. The randomness of the target network prevents this method from generalizing, and rather forces the predictor network to memorize [23, 24], which, for this application, is a desired property. By not allowing the model to learn some underlying representation of the data, unseen states will be correctly attributed a high MSE value.

3.2 Entropy Regularized Reinforcement Learning

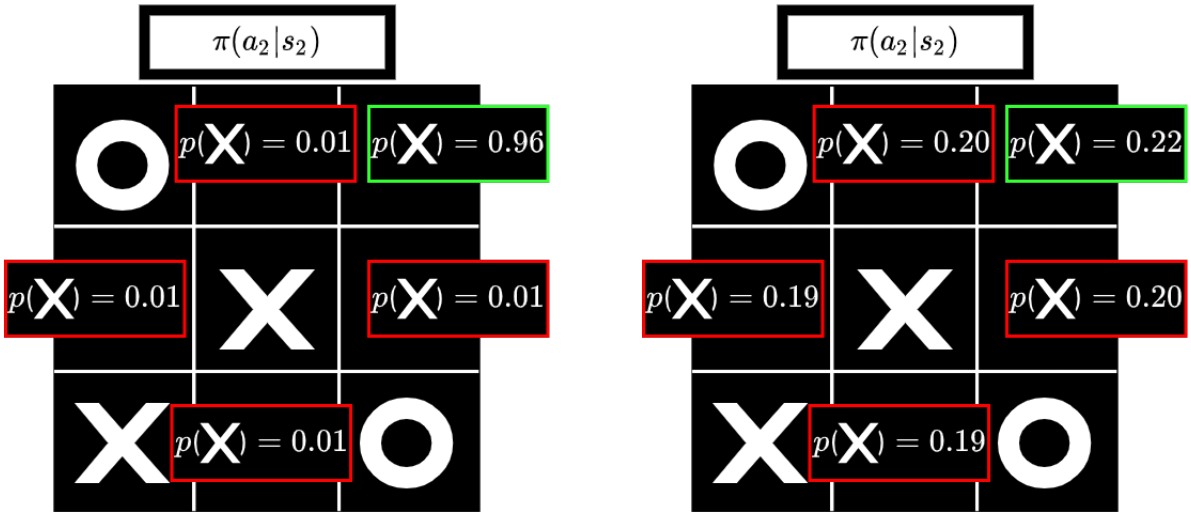
For a given random variable X, we denote the entropy as a $H(X)$, which quantifies the predictability or uncertainty of the random variable. We know from information theory [25] that a high entropy value denotes high uncertainty of the random variable, while a low entropy denotes low uncertainty

$$\begin{aligned} H(X) &= \mathbb{E}_{x \sim P(x)}[-\log(P(x))] \\ &= -\sum_{x \in X} P(x) \log(P(x)) \ . \end{aligned} \tag{3.1}$$

For a given stochastic policy $\pi(a|s)$, we can estimate the predictibilty of an action by the entropy of the policy $H(\pi(\cdot|s))$

$$\begin{aligned} H(\pi(\cdot|s)) &= \mathbb{E}_{\pi}[-\log(\pi(a|s))] \\ &= -\sum_{a \in \mathcal{A}} \pi(a|s) \log \pi(a|s)) \ . \end{aligned} \tag{3.2}$$

An example of entropy for policies can be seen in figure 3.2. The policy on the left has a low entropy, meaning its outcomes are very predictable. Indeed, the optimal action has a probability of being sampled of 0.96, and the observed entropy is 0.322. The policy on the right has a high entropy, meaning its outcomes are unpredictable. The optimal action only has a probability of being sampled of 0.22, while all other actions have relatively close probability, and the observed entropy is 2.32. In the context of RL, entropy is used as a



(a) Low entropy stochastic policy **(b)** High entropy stochastic policy

Figure 3.2: Low entropy vs. high entropy policy examples

regularizer. Similar to regularization in traditional supervised learning, entropy regularization prevents the policy function from overfitting to its current experience, which may often be a suboptimal solution to the given task. By preventing the stochastic policy to skew its distribution, exploration becomes baked inside the sampling process itself, as otherwise low probability actions are now given a higher sampling rate. This is the

fundamental idea behind Soft Actor Critic [7].

3.2.1 Soft Actor Critic

In regular policy gradient methods (section 2.5), we saw in eq. (2.16) the objective function we're optimizing is $J(\pi_\theta) = \mathbb{E}_{\pi_\theta}[G_t|s_t = s_0]$. The new entropy regularized objective function adds the regularization term $\alpha H(\pi_\theta(\cdot|s))$, where $\alpha \in \mathbb{R}^+$ is a coefficient to control the exploration versus exploitation balance. Setting $\alpha = 0$ would give back the original objective function

$$\begin{aligned}
J(\pi_\theta) &= \mathbb{E}_{\pi_\theta}[G_t|s_t = s_0] + \sum_{k=0}^{\infty} \alpha\gamma^k H(\pi_\theta(\cdot|s_{t+k})|s_t = s_0) \\
&= \mathbb{E}_{\pi_\theta}[G_t|s_t = s_0] + \sum_{k=0}^{\infty} \alpha\gamma^k \mathbb{E}_{\pi_\theta}[-\log(\pi_\theta(a_{t+k}|s_{t+k}))|s_t = s_0] \\
&= \mathbb{E}_{\pi_\theta}[G_t|s_t = s_0] - \mathbb{E}_{\pi_\theta}[\sum_{k=0}^{\infty} \alpha\gamma^k \log(\pi_\theta(a_{t+k}|s_{t+k}))|s_t = s_0] \\
&= \mathbb{E}_{\pi_\theta}[G_t - \sum_{k=0}^{\infty} \alpha\gamma^k \log(\pi_\theta(a_{t+k}|s_{t+k}))|s_t = s_0] \\
&= \mathbb{E}_{\pi_\theta}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} - \sum_{k=0}^{\infty} \alpha\gamma^k \log(\pi_\theta(a_{t+k}|s_{t+k}))|s_t = s_0] \\
&= \mathbb{E}_{\pi_\theta}[\sum_{k=0}^{\infty} \gamma^k (r_{t+k+1} - \alpha \log(\pi_\theta(a_{t+k}|s_{t+k})))|s_t = s_0] \ .
\end{aligned} \tag{3.3}$$

The entropy regularized objective function then simplifies down to giving an extra reward signal to the agent at every timestep, proportional to the entropy of the policy. From the new objective function, we can also write out the new value and Q functions, now called soft

value and soft Q functions

$$V^{\pi_\theta}_{soft}(s) = \mathbb{E}_{\pi_\theta}[\sum_{k=0}^{\infty} \gamma^k (r_{t+k+1} - \alpha \log(\pi_\theta(a_{t+k}|s_{t+k})))|s_t = s] \tag{3.4}$$

$$Q^{\pi_\theta}_{soft}(s,a) = \mathbb{E}_{\pi_\theta}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} - \sum_{k=1}^{\infty} \alpha\gamma^k \log(\pi_\theta(a_{t+k}|s_{t+k})))|s_t = s, a_t = a] \ . \tag{3.5}$$

Note that for the soft Q function, the entropy bonus is omitted for the first timestep. The relationship between the two new soft functions is then given by

$$V^{\pi_\theta}_{soft}(s) = \mathbb{E}_{a\sim\pi_\theta}[Q^{\pi_\theta}_{soft}(s,a) - \alpha \log \pi(a|s)] \tag{3.6}$$

$$\begin{aligned} Q^{\pi_\theta}_{soft}(s,a) &= \mathbb{E}_{\substack{s'\sim P \\ a'\sim\pi_\theta}}[r(s,a) + \gamma(Q^{\pi_\theta}_{soft}(s',a') - \alpha \log \pi(a'|s')] \\ &= \mathbb{E}_{s'\sim P}[r(s,a) + \gamma V^{\pi_\theta}_{soft}(s')] \ . \end{aligned} \tag{3.7}$$

3.3 Learning Different Skills

Up until this point, all of our efforts were focused on finding an optimal policy. Some recent efforts proposed a new framework using a maximum entropy policy to let an agent explore its environment and learn a set of skills without a reward function. In the paper "Diversity is All You Need: Learning Skills without a Reward Function" [2], the author define a latent variable $Z \sim p(z)$, on which they condition their policy $\pi_\theta(a|s,z)$. Here, the latent variable is

referred to as a skill. The goal of their new policy function is to have distinguishable outputs based on the skill it's conditioned on, by evaluating the next state s_{t+1} when sampling from $\pi_\theta(a_t|s_t, z)$. The authors define a discriminator $q_\phi(z|s)$, whose goal is to approximate $p(z|s)$. In essence, the goal of this discriminator is to estimate from the state it's conditioned on s, the skill z that produced this state. For a given state s_t, when sampling an action a_t from $\pi_\theta(a_t|s_t, z)$, the next state s_{t+1} should be such that it is maximally distinguishable from other skills, by the discriminator $q_\phi(z|s_{t+1})$. The reward the agent observes at timestep t is entirely intrinsic, as its goal is to purely maximize the discriminability of the skills. The reward observed is $r_t = \log q_\phi(z|s_{t+1}) - \log p(z)$, and the policy parameters are updated using Soft Actor Critic [7]. A depiction of the framework can be seen in figure 3.3.

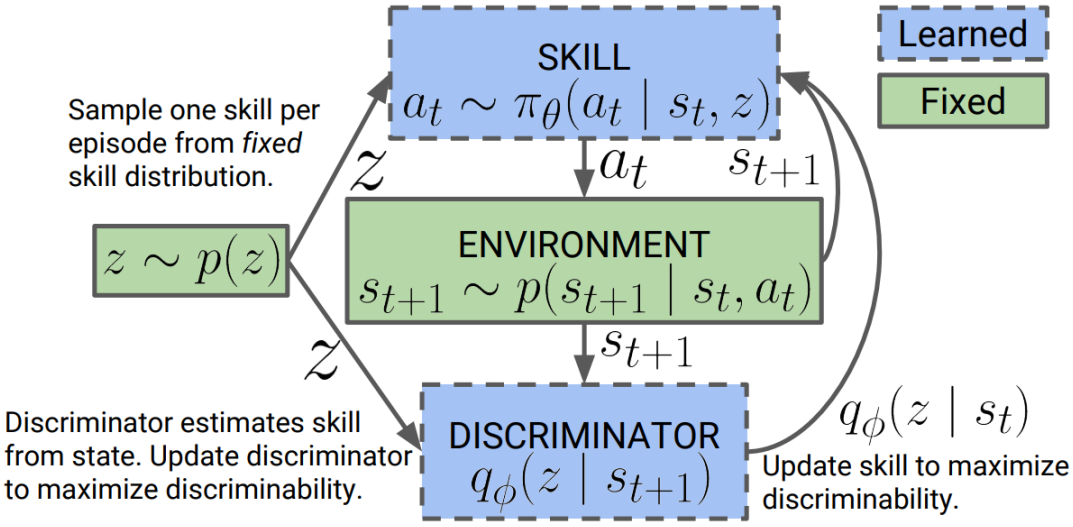


Figure 3.3: Diversity Is All You Need Architecture [2]

The formulation of the reward function comes from maximizing the entropy of $p(z)$, while

minimizing the entropy of $p(z|s)$, approximated by $q_\phi(z|s)$. By maximizing the entropy of the prior distribution $H(Z)$, this encourages the skill diversity the agent will generate. Conversely, by minimizing $H(Z|S)$, this encourages the agent's policy to maximize discriminability

$$\begin{aligned} H(Z) - H(Z|S) &= \mathbb{E}_{s\sim\pi(z)}[-\log p(z)] - \mathbb{E}_{\substack{z\sim p(z)\\ s\sim\pi(z)}}[-\log q_\phi(z|s)] \\ &= \mathbb{E}_{\substack{z\sim p(z)\\ s\sim\pi(z)}}[\log q_\phi(z|s)] - \mathbb{E}_{s\sim\pi(z)}[\log p(z)] \\ &= \mathbb{E}_{\substack{z\sim p(z)\\ s\sim\pi(z)}}[\log q_\phi(z|s) - \log p(z)] \ . \end{aligned} \tag{3.8}$$

The prior distribution of the latent variable z is fixed to be a uniform distribution, as it guarantees a maximum entropy. The resulting learned skills are all uniquely distinguishable, with all of the skills that do correspond to solving the task being different solutions. However, there is no guarantee that a learned skill will result in solving the task, as the reward is purely intrinsic. While some of the learned skills could be valid solutions, some other skills could also be seemingly random actions. The main idea here is to conduct an exhaustive search in policy space, by extracting as many unique policies, regardless of their effectiveness.

Chapter 4

Methodology

We will present our novel framework called *Adversarial Strategy Learning*. The goal of this framework is to learn multiple unique strategies to solve a given environment, by treating expert demonstrations as adversarial examples. The first challenge that comes to mind is defining a strategy. We do not want to treat every expert observation as adversarial, as some behaviour might be common to every solution for a task, and we therefore need to determine the likelihood of an observation being part of the strategy or not. To do so, our framework analyses the Q-values along the expert's trajectory, specifically, we will look at the directional gradient of Q-values. Our framework also makes use of a Random Network Distillation (RND) [21] unit, however instead of using it as a source of intrinsic reward as it is traditionally, we will make use of some of its innate properties to act as a gating function on the state space. Before considering if an observation is part of a the strategy or not, we first consider if it was ever part of the expert demonstrations, using the RND gating function. We then use this new information to compute a new source of intrinsic bonus reward, however rather than being an exploration reward, this new reward signal's purpose is to discourage

the agent from mimicking the identified strategic behaviour. We run our new framework on the OpenAI Gym [26] environments MountainCarContinuous and Pendulum, as well as a few custom GridWorld-like environment specifically curated to demonstrate our algorithm.

4.1 Defining a Strategy

The first goal of our framework is to determine which behaviour would be considered part of the unique strategy of an expert. We start with an agent fully trained with an actor-critic algorithm (SAC [7] in our case) which will serve as our expert, and save both the policy function π_θ and Q-function $Q_\phi^{\pi_\theta}$. For a given state s_t, we sample an action a_t from the policy $a_t \sim \pi_\theta(a_t|s_t)$, as well as evaluate the Q-value of our state-action pair $Q_\phi^{\pi_\theta}(s_t, a_t)$. We then proceed to take the gradient of our Q-function with respect to the input state s_t, $\nabla_{s_t} Q_\phi^{\pi_\theta}(s_t, a_t)$, then project this gradient along the vector towards the next state s_{t+1}. This will give us a new scalar value that we denote by grad_t, given by:

$$\text{grad}_{t+1} = \nabla_{s_t} Q_\phi^{\pi_\theta}(s_t, a_t) \cdot \overrightarrow{(s_{t+1} - s_t)} \,. \tag{4.1}$$

Note that since the evaluation of grad_t requires the information of a previous state s_{t-1}, we set $\text{grad}_{t=0} = 0$. For a given expert on the OpenAI Gym [26] MountainCarContinuous environment, a visualization of the evolution of the gradient value grad_t along the trajectory

can be seen in figure 4.1. Note that we're using $\nabla_{s_t} Q(s_t, a_t)$ as a shorthand for $\nabla_{s_t} Q_\phi^{\pi_\theta}(s_t, a_t) \cdot \overrightarrow{(s_{t+1} - s_t)}$. The expert's behaviour that generated related to figure 4.1 can be seen in 4.2.

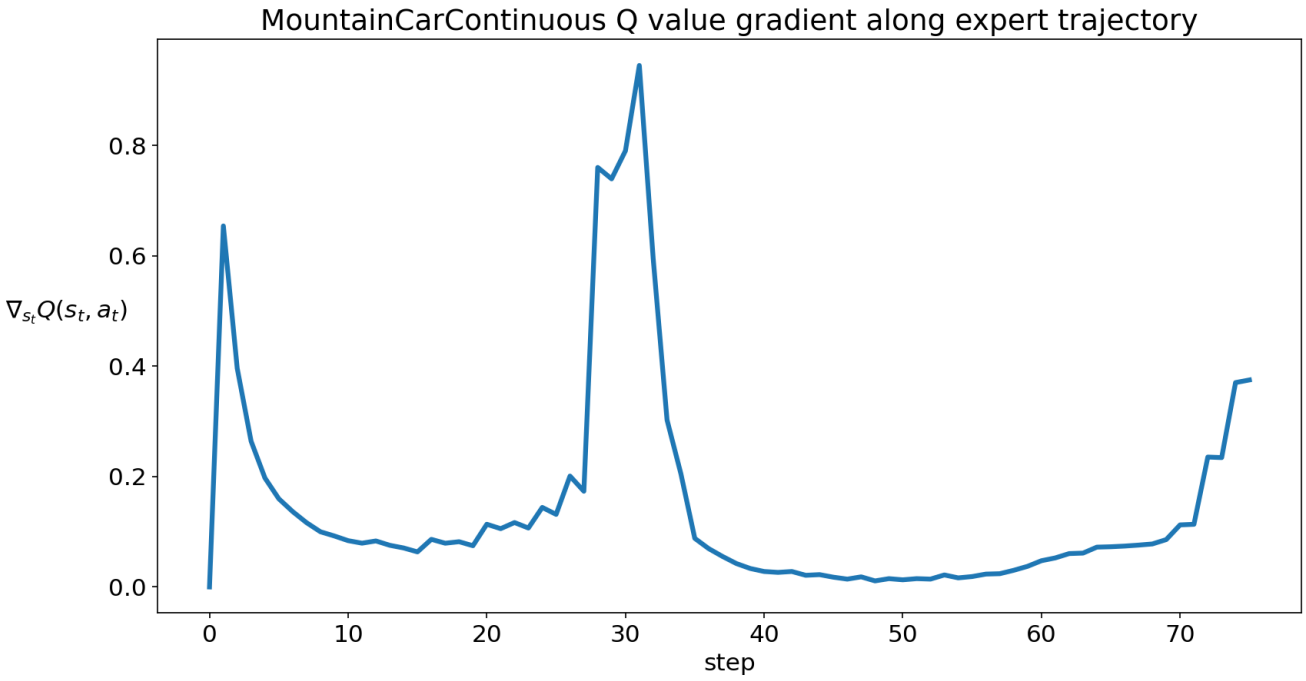


Figure 4.1: Directional gradient of Q-values along expert trajectory, MountainCar

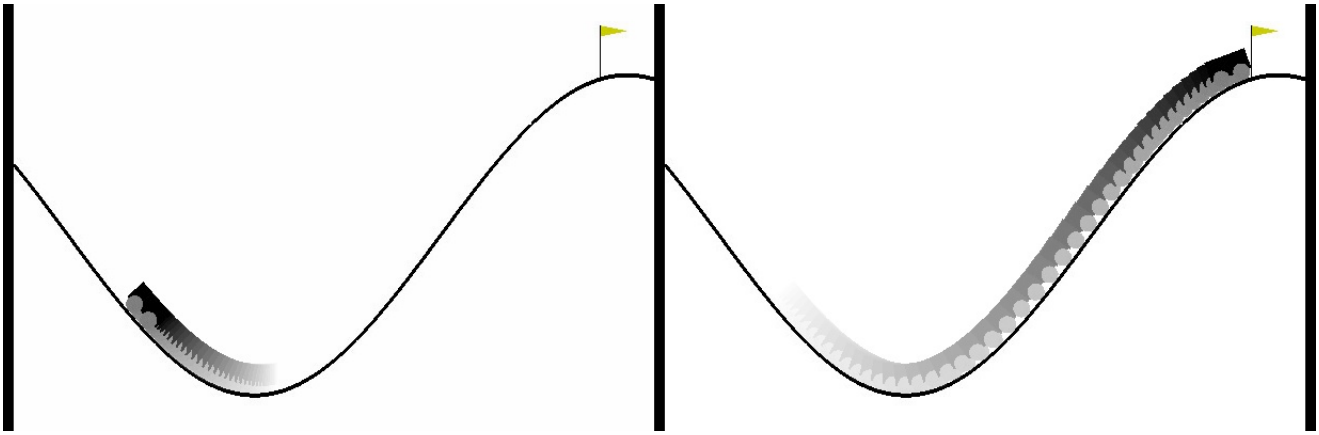


Figure 4.2: MountainCar expert behaviour, panorama view

The panorama view of the expert in figure 4.2 splits the trajectory into two parts. The view on the left correspond to frames zero to 34, while the view on the right correspond to frames 35 and onward. We use an alpha blending on the frames to show the progression of the expert over time. The peak of gradients we see roughly around steps 30 corresponds to the moment where the expert reaches its maximum height on the left hill, which is also the moment just enough elevation is reached to generate the momentum required to solve the task. This also corresponds to the high level idea of the strategy of this task, which is generating enough momentum. The desired behaviour of our framework would then be to discourage our new agent to mimic the behaviour observed in these high gradient states.

4.2 Data Generation and State Resampling

We start by gathering trajectories from our expert, storing the observations in an ordered data buffer, to keep track of the continuity of our observations in order to evaluate our directional gradient. At each timestep t, we record the current state s_t and the action taken a_t sampled from the policy $\pi_\theta(a_t|s_t)$, and store them in our buffer. We then evaluate our directional gradient grad_t that we also store in this buffer. This dataset now associates each state with the directional gradient observed when our expert transitioned to it. Since we're merging the observations of many trajectories and we already computed our directional gradient, we no longer need to keep track of the continuity of our observations. We then simply combine all our data points into one unified unordered buffer of pairs of state-gradient (s_i, grad_i) of some size N. Once our dataset is assembled, we proceed to a state re-sampling with repetition. We create a new empty dataset of similar size N, which we populate with states sampled proportional to their associated grad_i value. To do so, perform a softmax across all values of grad_i, and assign this value to each respective state as their probability of being sampled $P(s_i)$

$$P(s_i) = \frac{\exp(\text{grad}_i)}{\sum_{k=0}^{N} \exp(\text{grad}_k)} \ . \tag{4.2}$$

We proceed to populate our new dataset by repeatedly sampling N states with repetition, with respect to the new computed sampling probability $P(s_i)$. We associate to every state s_i some occurrence value occ_i, which is simply a counter that keeps track of how many times a specific state was resampled. Our new dataset then consists of the pairs (s_i, occ_i). We perform this resampling of our dataset over many iterations, then average the result of the occurrence values. Finally, we normalize our occurrence values occ_i to be between zero and one by a simply dividing by the range and subtracting the minimum

$$\text{occ}_i = \frac{\text{occ}_i - \min\{\text{occ}_0, \text{occ}_1, \ldots, \text{occ}_N\}}{\max\{\text{occ}_0, \text{occ}_1, \ldots, \text{occ}_N\} - \min\{\text{occ}_0, \text{occ}_1, \ldots, \text{occ}_N\}} \ . \tag{4.3}$$

Our final dataset then consists of the pairs (s_i, occ_i). The states that observed higher values of directional gradients grad_i will have an occurrence occ_i value closer to one, while the states that observed relatively lower values of grad_i will have an occ_i value closer to zero. We interpret this dataset as a mapping between a state, and its likelihood to be considered part of the expert's strategy.

4.3 Intrinsic Bonus Reward Mechanism

In order to discourage an agent from imitating the expert's strategy, we generate a source of intrinsic reward based on the dataset from in section 4.2. The two main components of this method are the state likelihood discriminator presented in section 4.3.1, and the random network distillation gating function presented in section 4.3.2. We finally also explain how we combine the components to generate the intrinsic reward in section 4.3.3.

4.3.1 State Likelihood Discriminator

We define a discriminator D parametrized by weights w, which takes as input a state s, and returns a value between zero and one, $D_w(s) : \mathcal{S} \rightarrow (0, 1)$. The goal of this discriminator is determine the likelihood of the input state to be considered part of the expert's strategy. This discriminator is trained through supervised learning to directly learn the dataset we created (s_i, occ_i), using the occurrence value occ_i as the output target.

4.3.2 Random Network Distillation Gating Function

We define a random network distillation with an architecture similar to what was presented in section 3.1, however rather than using it as a source of intrinsic reward for exploration, our RND will serve a different purpose. The discriminator introduced in section 4.3.1 is only trained on expert examples, and is therefore expected to behave poorly on non-expert

examples. Before we assess the importance of a state to the expert's strategy, we need to first determine if the state was ever encountered by the expert, to get an accurate result from our state likelihood discriminator. Our solution to this is to use some of the underlying assumptions of the random network distillation and use it as a gating function.

Since we know that a data point that was seen by a random network distillation would have lower mean squared error than one that was not, we train our RND over many iterations on the expert states. After enough iterations, we perform a final forward pass on all our data points, and store some of the statistics observed of the entire expert dataset. We save the mean MSE μ, as well as the standard deviation σ to later determine if a given state was ever part of the expert dataset.

We define a function $rnd(s)$ as the function that takes an input a state s, and returns the random network distillation's mean squared error, $rnd(s) : \mathcal{S} \rightarrow \mathbb{R}$. We define a second function $gating(s)$ as the gating function based on the RND, which takes as input a state s, and returns either zero or one, $gating(s) : \mathcal{S} \rightarrow \{0, 1\}$. The output of the gating function translates to the state being part of the expert dataset or not. This is done by computing $rnd(s)$ and and checking if $rnd(s) > \mu + 3\sigma$. Note that for numerical stability, all distances measured by our random network distillation will be on a logarithmic scale. The architecture of our random network distillation gating function can be seen in figure 4.3.

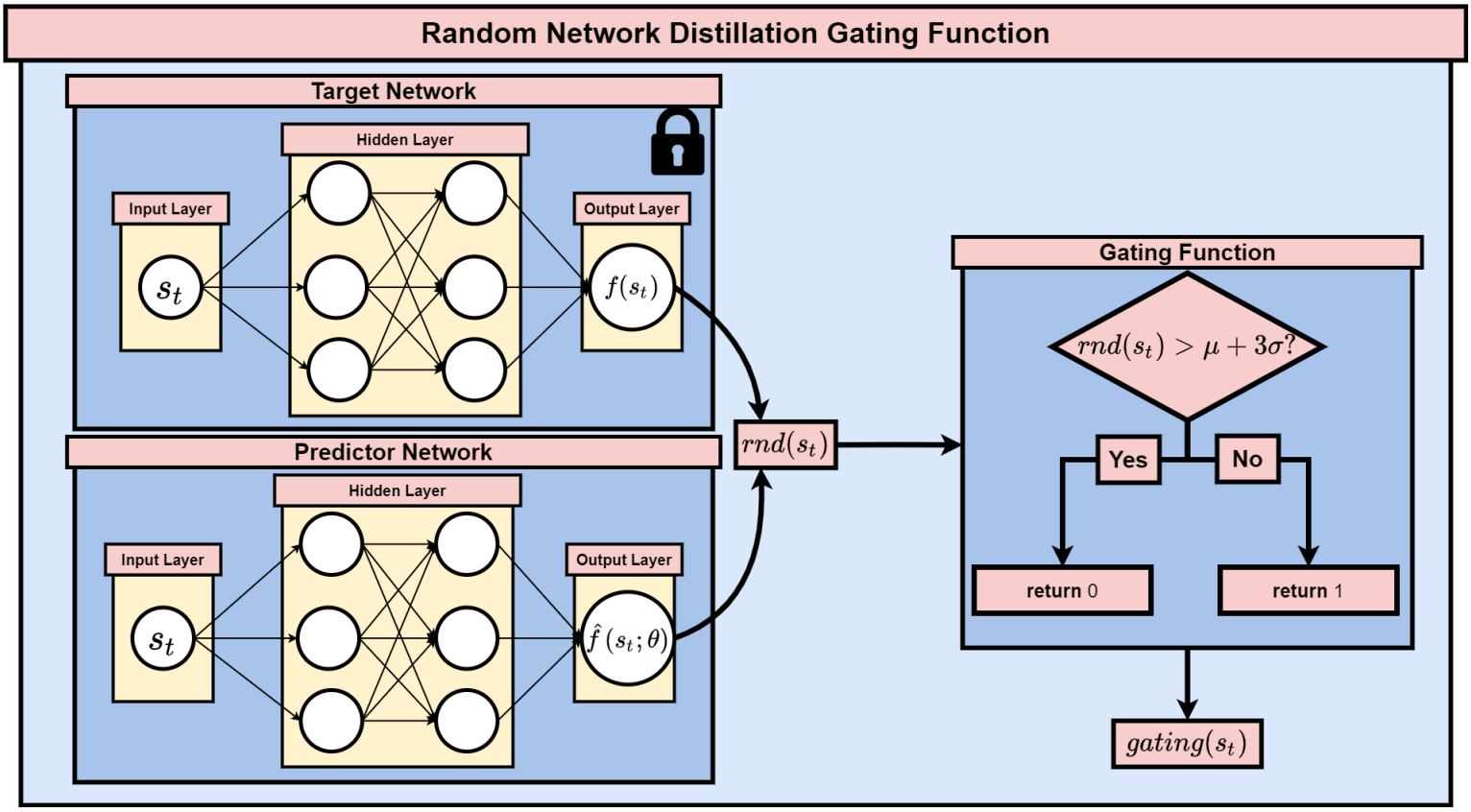


Figure 4.3: Random Network Distillation Gating Function Architecture

4.3.3 Combining the Learned Functions

We take the expert trajectories, which are the same one seen in panorama view in figure 4.2 and project the state space into a 2D plane. The x-axis represents the position of the agent, and the y-axis represents the velocity of the agent. The visualization of the expert's projected state space can be seen in figure 4.4. To illustrate how both the random network distillation

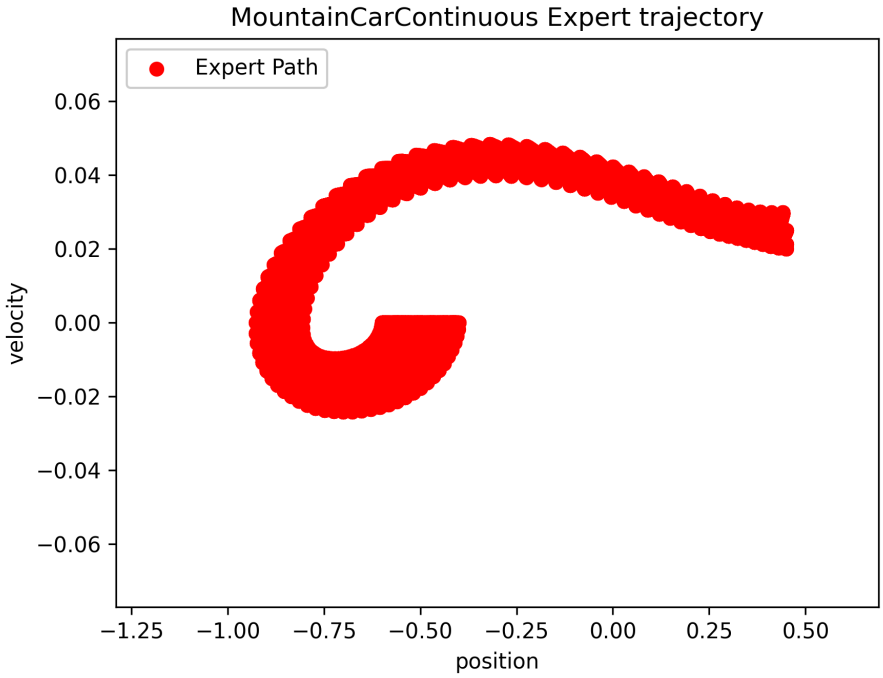
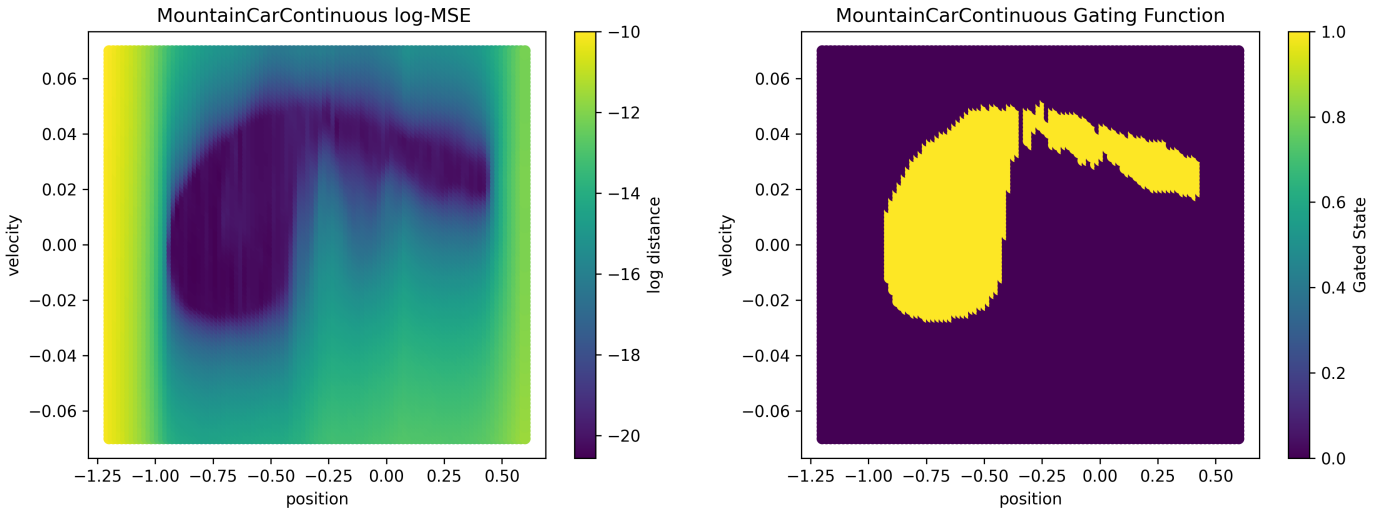


Figure 4.4: MountainCar expert behaviour, projected view

and the gating functions are functioning, we uniformly sample the entire state space of the environment, and run every data point through our $rnd(s)$ and $gating(s)$ functions. We visualize these outputs in figure 4.5.

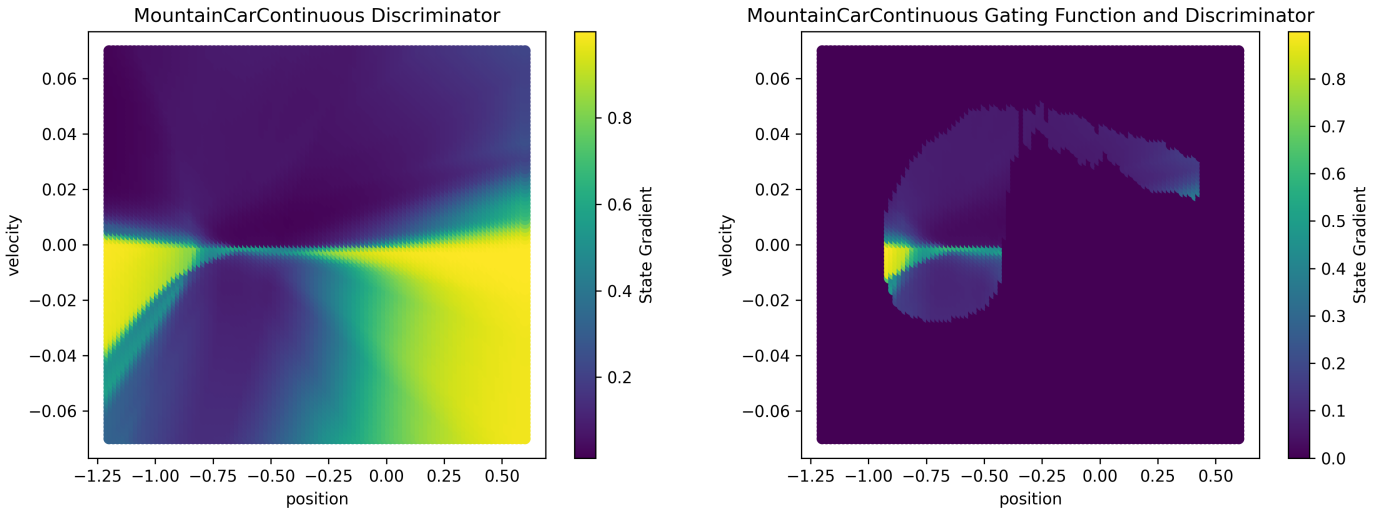


(a) MountainCar expert: $rnd(s)$

(b) MountainCar expert: $gating(s)$

Figure 4.5: MountainCar expert learned RND and gating functions

By combining our discriminator with our gating function, we then have a way to estimate the importance of a state to the expert's strategy, on the entire state space. To do so, we simply multiply the outputs of $D_w(s)$ and $gating(s)$. We visualize this in figure 4.6.



(a) MountainCar expert: $D_w(s)$ **(b)** MountainCar expert: $D_w(s) * gating(s)$

Figure 4.6: MountainCar expert intrinsic bonus map

Since this value is strictly between zero and one, we can therefore scale it to any environment based on a hyperparameter we denote by C_{ASL}, which would be a constant dependant on the task. Since we want to discourage our agent to imitate our expert, this constant would have to be strictly negative. The reward our agent observes R_{agent} is then a combinations of the environment reward R_{env} and the intrinsic reward $D_w(s) * gating(s) * C_{ASL}$

$$R_{\text{agent}} = R_{\text{env}} + D_w(s) * gating(s) * C_{ASL} \ . \tag{4.4}$$

Chapter 5

Experiments

We will present all of our experiments conducted using our novel framework along with their hyperparameters and results. For each experiment, we will provide a visual representation of the strategy for both the expert and the newly trained agent. Depending on the environment, we will either provide a timelapsed image with progressive alpha-blending, similar to what was shown in figure 4.2, or simply a projected trajectory onto a 2D graph as shown in figure 4.4. However, since these results are best seen in video format, we also provide a rendered video of the results with each experiment, in the form of an externally hosted link. We're only interested in extracting distinct unique solutions to solve a task, and do not care so much about the optimality of each solution, so long as it reasonably solves the task. This means, the unique strategy consists of a novel sequence of events, and not a inconsequential sequence of events followed by an already seen solution. The results are therefore not quantifiable by some metric such as the reward observed, but requires rather some form of human estimation of a strategy to solve a problem, i.e. the technique used to generate enough momentum or finding a path in a maze.

To ensure that the comparison between the expert and the agent is on the exact same condition, we created a tool[1] that takes as input an arbitrary amount of RL agents and an environment. As some environments have varying initial conditions, our tool first randomly initializes the environment, then creates a clone for each agent to use. The agents then run on their cloned environment until completion. We save the rendering of each agent individually, as well as stack them on top of each other for a simpler comparison. We also provide a functionality for our tool to change a target pixel value into another specified value. This allows us to colour code our agents and differentiate them from the experts, making it easier to evaluate our results. All experiments were implemented using PyTorch [27, 28].

[1] `https://github.com/danielbairamian/Spotter`

5.1 MountainCar

The first experiment we run is on the MountainCarContinuous environment from the OpenAI gym [26]. The setup of this experiment is a car located at the bottom of a hill, with the goal being climbing the hill to reach a target, represented by the yellow flag, seen in figure 5.1. What makes this problem interesting is that the car's motor is too weak to directly climb the hill, and the agent therefore needs to use the environment physics to generate momentum. The state space consists of two dimensions, the car position with range $[-1.2, 0.6]$, and the car velocity with range $[-0.07, 0.07]$. The action space consists of a single dimension which represents the push on the car, a value with the range $[-1.0, 1.0]$. The reward observed by the agent is 100 when reaching the target at the top of the hill, minus the squared sum of actions from the start to the goal. The expert's strategy can be seen in figure 5.2. The resulting trained agent's behaviour compared to the expert's behaviour can be seen in figure 5.3. The hyperparameters used in this experiment can be found in table 5.1.

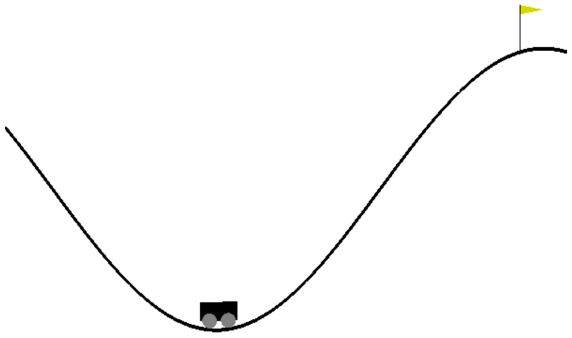
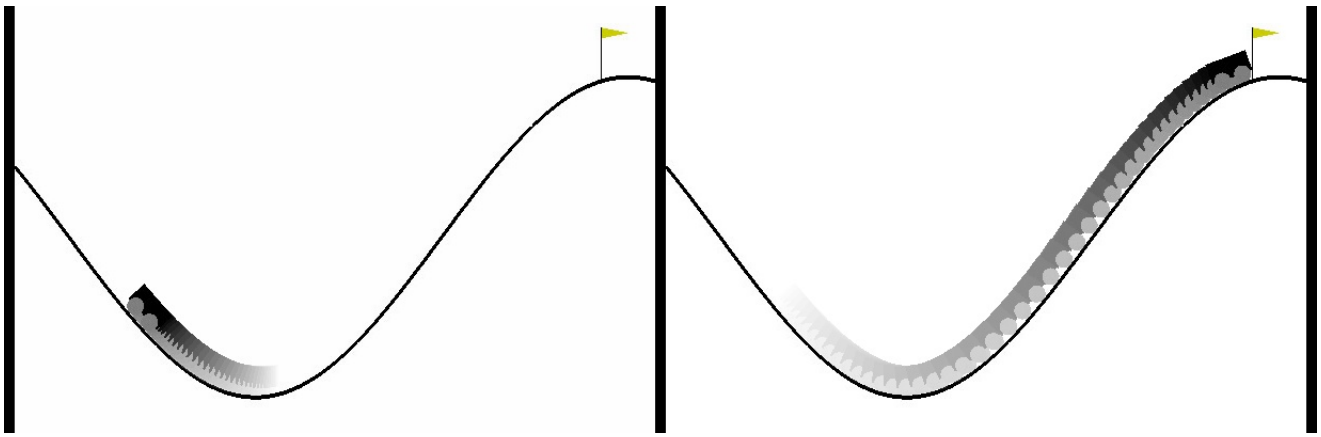
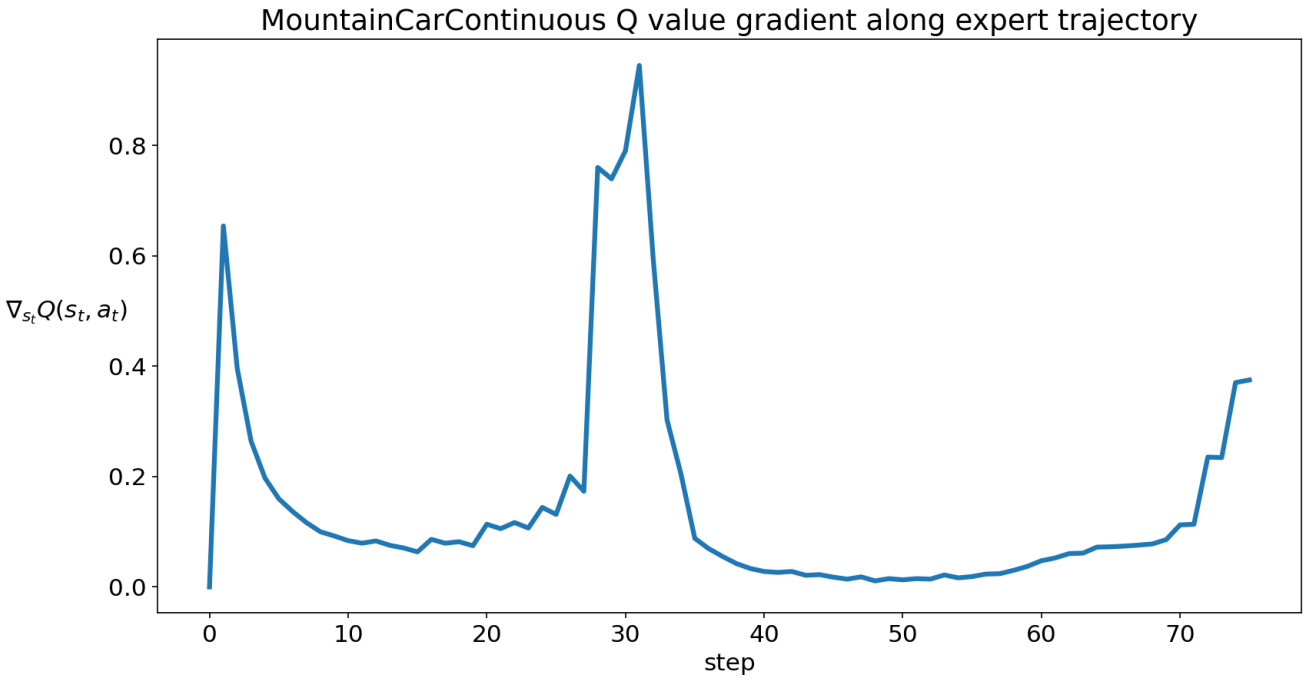


Figure 5.1: MountainCar Experiment Setup

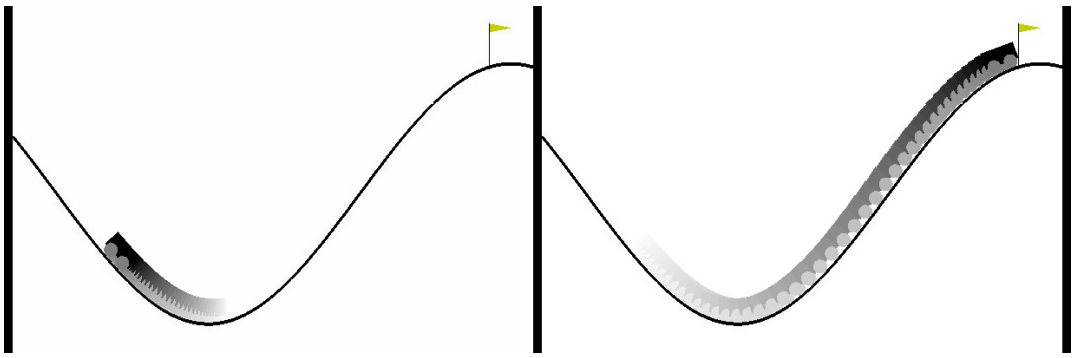


(a) Expert panorama view

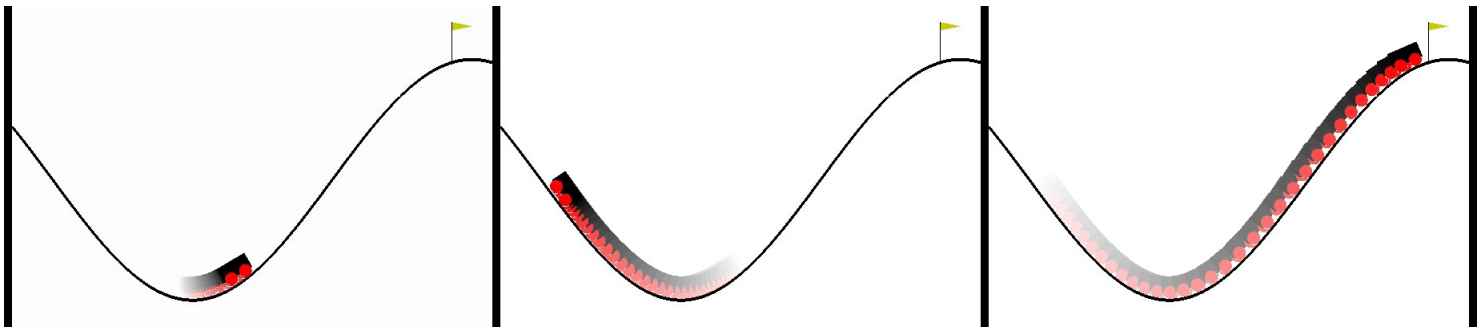


(b) Expert gradient view

Figure 5.2: Expert MountainCar strategy



(a) MountainCar expert behaviour



(b) MountainCar agent behaviour

Figure 5.3: MountainCar expert vs. agent behaviour, panorama view [a]

[a] `https://github.com/danielbairamian/Spotter/blob/main/Results/MountainCar.md`

Data Generation Hyperparameters	**Value**
Expert Trajectories Extracted	1000
Resampling Repetition	1000
Random Network Distillation Hyperparameters	**Value**
Network Dimensions	[2, 256, 256, 256, 1024]
Network Activations	[ReLU, ReLU, ReLU, Identity]
Training Epochs	1000
Learning Rate	1e-3
Batch Size	1024
Discriminator Hyperparameters	**Value**
Network Dimensions	[2, 256, 256, 256, 1]
Network Activations	[ReLU, ReLU, ReLU, Sigmoid]
Training Epochs	100
Learning Rate	1e-3
Batch Size	256
RL Agent Hyperparameters	**Value**
Policy Network Dimensions	[2, 256, 256, 1]
Policy Network Activation (Gaussian Net)	[ReLU, ReLU, Identity]
Q-Network Dimensions	[3, 256, 256, 1]
Q-Network Activations	[ReLU, ReLU, Identity]
Actor & Critic Learning Rate	1e-3
SAC exploration parameter α	0.2
C_{ASL}	-25

Table 5.1: MountainCarContinuous experiment hyperparameters

5.2 FourRooms

The second experiment we run is a custom environment we call FourRooms, inspired from the work by [29]. We design a simple 2D environment split into four rooms connected by small openings. The agent, symbolized by the red circle, starts in the bottom left room at the position $[2.0, 2.0]$ with some small Gaussian noise on both dimensions, seen in figure 5.4. The goal is situated at the upper right room, represented by the orange rectangle. The reason we chose this setup is because this environment has two clear solutions. If our method works correctly, it should correctly extract the only two paths available to get to the goal. The state space consists of two dimensions, the agent's x and y position, both with range $[0.0, 10.0]$. The action space consists of a single dimension value with range $[-\pi, \pi]$, which represents an angular value. At every time step, the agent moves in the direction of this angle at some fixed speed of 0.25. The reward observed by the agent is 100 when reaching the goal, and a negative reward for every step where the agent is not at the goal. To encourage the agent to go towards the goal room, rather than giving a negative distance to the goal as the reward, we give a reward of -1.0 if the agent is in the bottom left room, a reward of -0.1 if the agent is in the room which contains the goal, and a reward of -0.5 if the agent is in any of the other two rooms. This way, we only encourage the agent to traverse rooms, without giving it any information about the goal location. The expert's strategy can be seen in figure 5.5. The resulting trained agent's behaviour compared to the expert's behaviour

can be seen in figure 5.6. The hyperparameters used in this experiment can be found in table 5.2.

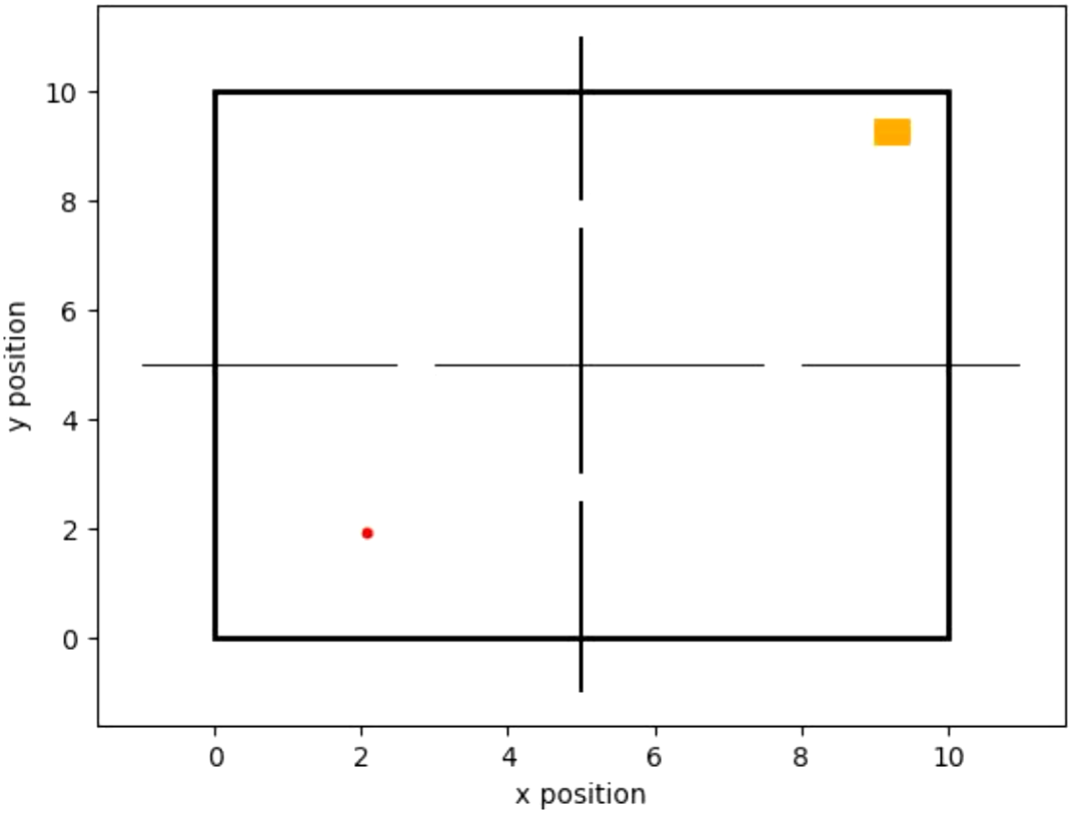
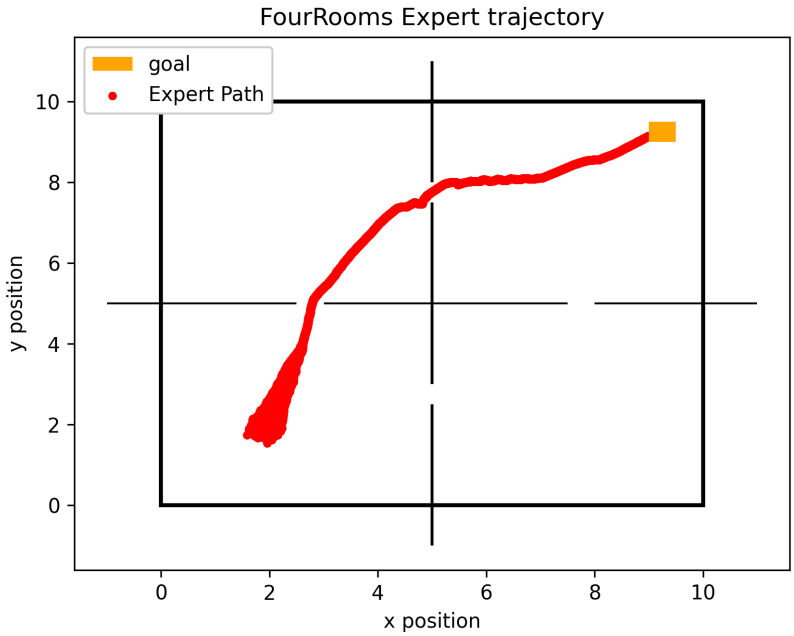
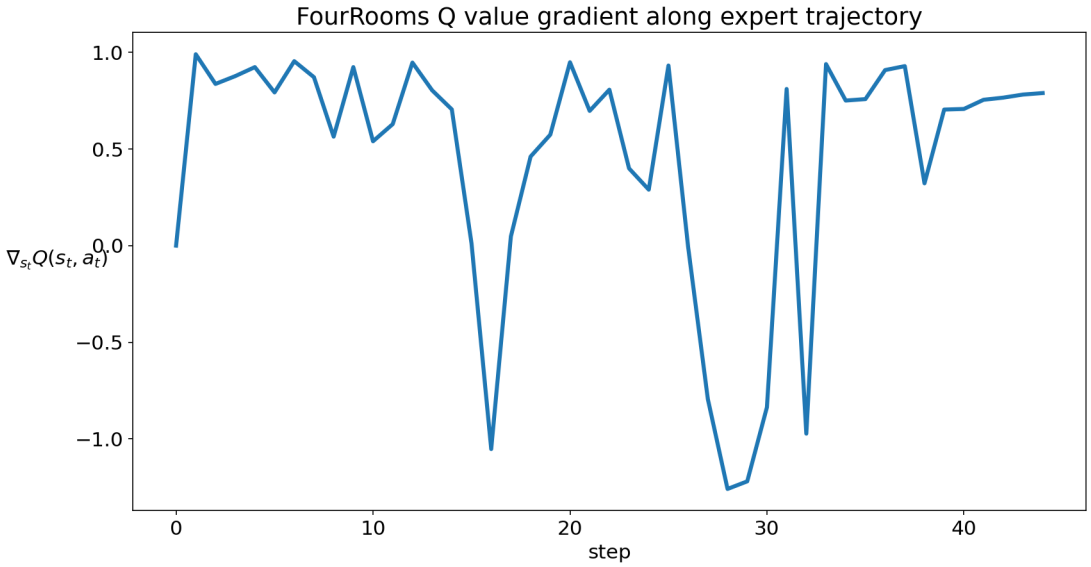


Figure 5.4: FourRooms Experiment Setup



(a) Expert panorama view



(b) Expert gradient view

Figure 5.5: Expert FourRooms strategy

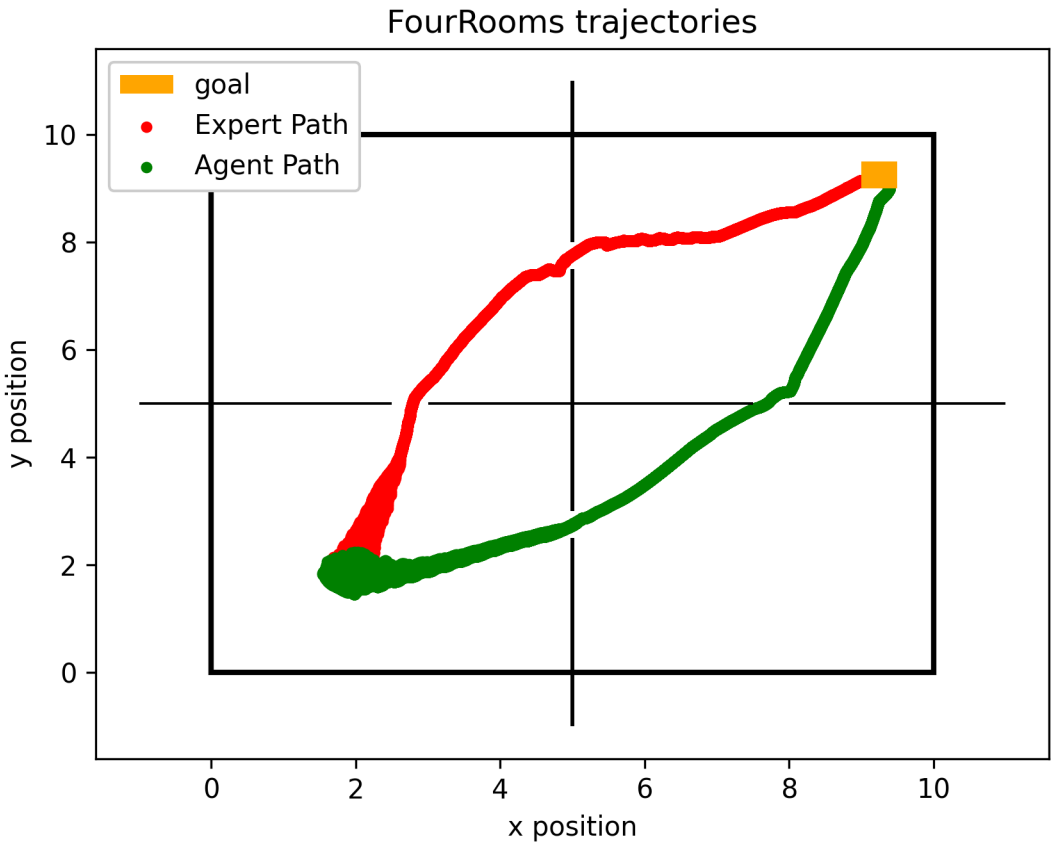


Figure 5.6: FourRooms expert vs. agent behaviour, projected view [a]

[a] https://github.com/danielbairamian/Spotter/blob/main/Results/FourRooms.md

Data Generation Hyperparameters	**Value**
Expert Trajectories Extracted	1000
Resampling Repetition	1000
Random Network Distillation Hyperparameters	**Value**
Network Dimensions	[2, 256, 256, 256, 1024]
Network Activations	[ReLU, ReLU, ReLU, Identity]
Training Epochs	1000
Learning Rate	1e-3
Batch Size	1024
Discriminator Hyperparameters	**Value**
Network Dimensions	[2, 256, 256, 256, 1]
Network Activations	[ReLU, ReLU, ReLU, Sigmoid]
Training Epochs	100
Learning Rate	1e-3
Batch Size	256
RL Agent Hyperparameters	**Value**
Policy Network Dimensions	[2, 256, 256, 1]
Policy Network Activation (Gaussian Net)	[ReLU, ReLU, Identity]
Q-Network Dimensions	[3, 256, 256, 1]
Q-Network Activations	[ReLU, ReLU, Identity]
Actor & Critic Learning Rate	1e-3
SAC exploration parameter α	0.2
C_{ASL}	-50

Table 5.2: FourRooms experiment hyperparameters

5.3 GridWorld

The third environment we simply call GridWorld, is similar to the previous one, only now without the walls, and modified starting and goal positions, seen in figure 5.7. The agent observes a reward of 100 when reaching the goal, and a negative reward of -1.0 for every step otherwise. The idea here is to test how our method would deal with environments that have smooth Q-value gradients. The expert's strategy can be seen in figure 5.8. The resulting trained agent's behaviour compared to the expert's behaviour can be seen in figure 5.9. The hyperparameters used in this experiment can be found in table 5.3.

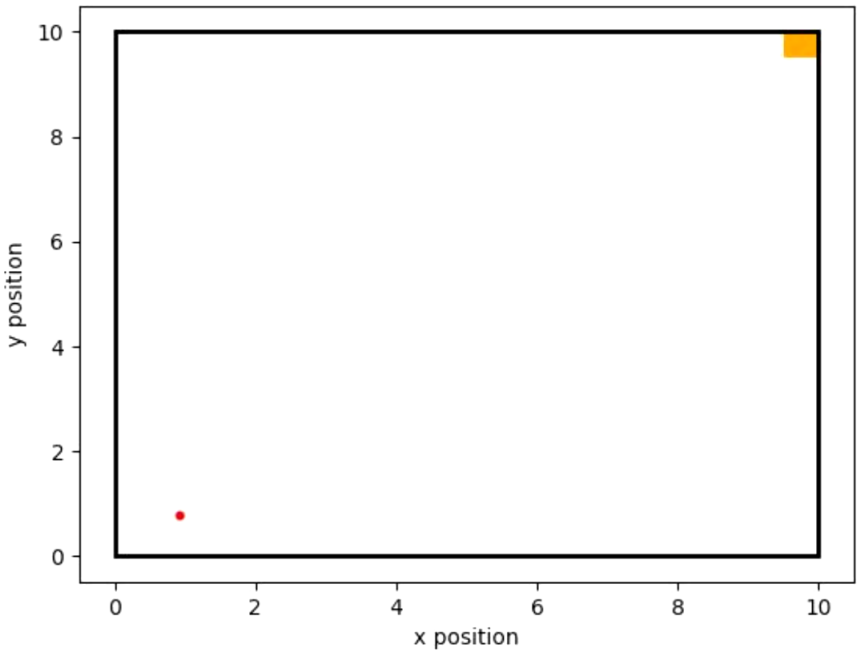
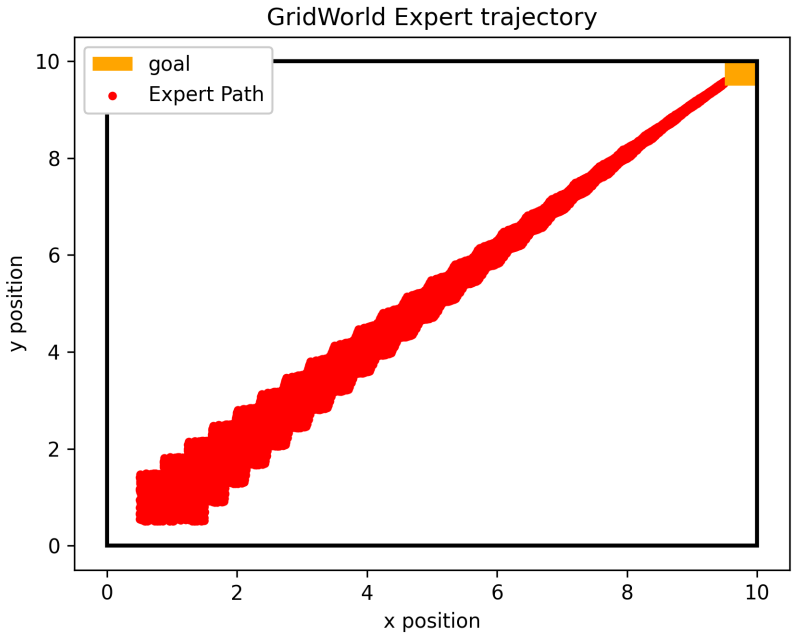
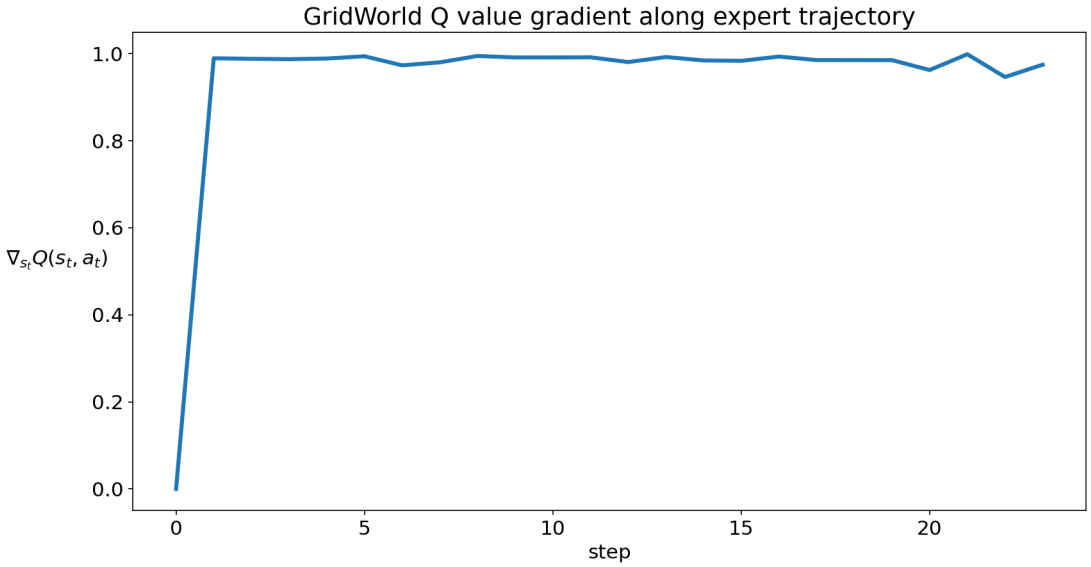


Figure 5.7: GridWorld Experiment Setup



(a) Expert panorama view



(b) Expert gradient view

Figure 5.8: Expert GridWorld strategy

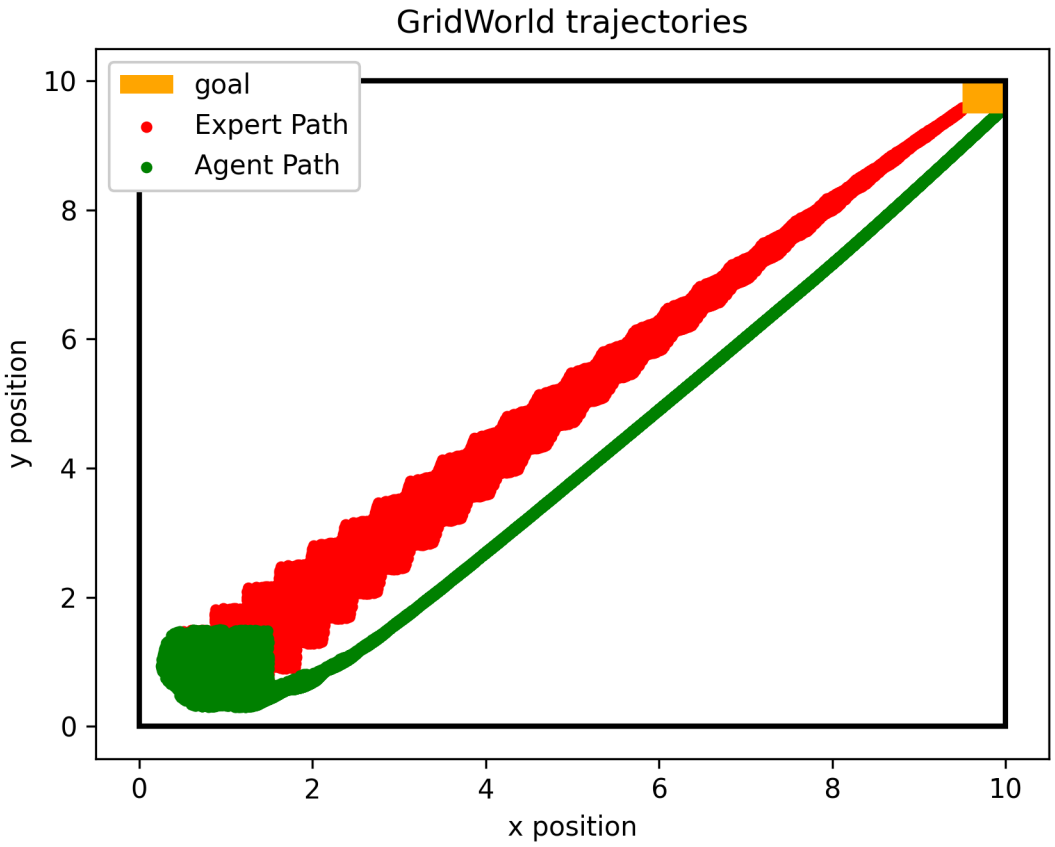


Figure 5.9: GridWorld expert vs. agent behaviour, projected view[a]

[a] https://github.com/danielbairamian/Spotter/blob/main/Results/GridWorld.md

Data Generation Hyperparameters	**Value**
Expert Trajectories Extracted	1000
Resampling Repetition	1000
Random Network Distillation Hyperparameters	**Value**
Network Dimensions	[2, 256, 256, 256, 1024]
Network Activations	[ReLU, ReLU, ReLU, Identity]
Training Epochs	1000
Learning Rate	1e-3
Batch Size	1024
Discriminator Hyperparameters	**Value**
Network Dimensions	[2, 256, 256, 256, 1]
Network Activations	[ReLU, ReLU, ReLU, Sigmoid]
Training Epochs	100
Learning Rate	1e-3
Batch Size	256
RL Agent Hyperparameters	**Value**
Policy Network Dimensions	[2, 256, 256, 1]
Policy Network Activation (Gaussian Net)	[ReLU, ReLU, Identity]
Q-Network Dimensions	[3, 256, 256, 1]
Q-Network Activations	[ReLU, ReLU, Identity]
Actor & Critic Learning Rate	1e-3
SAC exploration parameter α	0.2
C_{ASL}	-50

Table 5.3: GridWorld experiment hyperparameters

5.4 Pendulum

The fourth experiment we run is on the PENDULUM environment from the OpenAI gym. The setup of this experiment consists of balancing a pendulum upright by exerting some force on the joint. The force the joint can exert is limited, and therefore not able to get upright directly from any position. The agent needs to learn how to use the environment physics to help it gain the momentum required to hoist itself upward. The pendulum's initial position is completely randomised for each episode, with a random velocity, seen in figure 5.10. The black curved arrow indicates the joint effort applied at the current step. The state space consists of three dimensions, $\cos\theta$ and $\sin\theta$ of the current angle θ, with range $[-1.0, 1.0]$, as well as the current angular velocity $\dot{\theta}$ with range $[-8.0, 8.0]$. The action space consists of a single dimension which represents the joint effort, with value $[-2.0, 2.0]$. The reward observed by the agent is $r_t = -(\theta^2 + 0.1 * \dot{\theta}^2 + 0.001 * a_t^2)$.

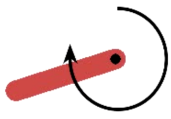
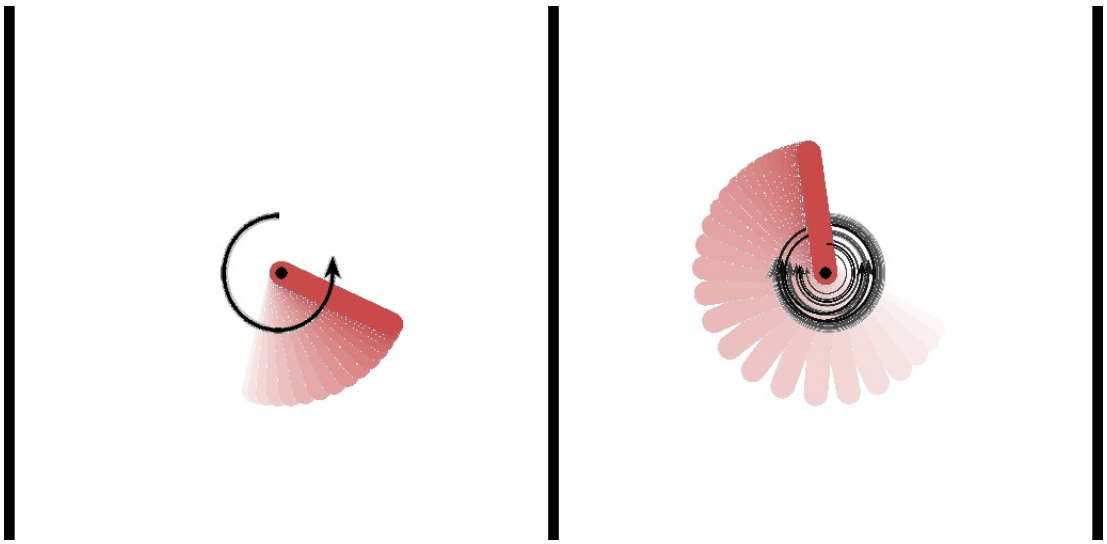
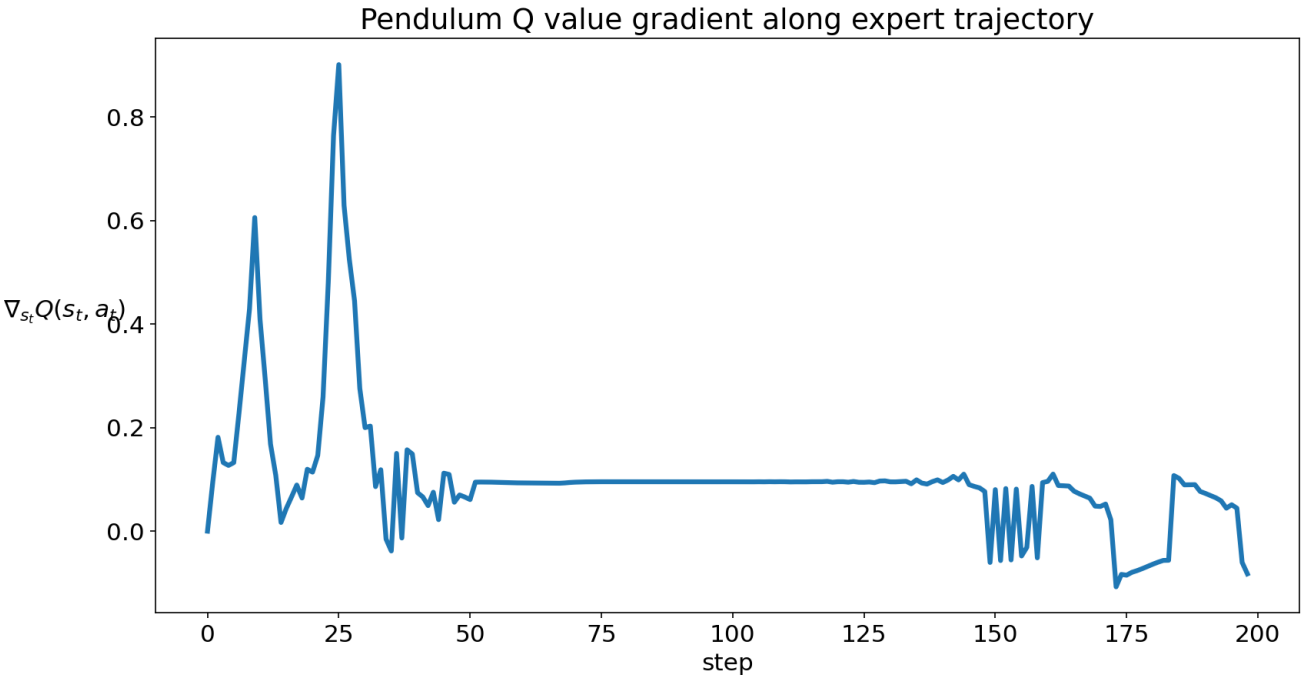


Figure 5.10: Pendulum Experiment Setup

The PENDULUM environment poses a unique challenge to our approach, as each episode is initialized randomly. Since we're banning strategic states along trajectories, the importance of a given state will drastically change given the starting position. If we naively apply our technique to this environment, our discriminator ends up severely banning the state space almost entirely, and our new agent gets a strong negative reward signal for whatever action it takes, ending up doing nothing. Our solution to this is to concatenate the initial state to the state space. This means, for environments where the starting position varies so much that a different solution is required to solve the task, we provide the initial state as additional information. Our state s_t becomes $[s_t, s_0]$, and our method is then applied as previously mentioned without any additional changes. This environment has multiple obvious strategies given the starting position. Sometimes, the agent requires so much momentum to reach the top that it needs to perform two swings to generate enough speed. We can see this in figure 5.11, where we can clearly identify two peaks in Q-value gradient. Other cases sometimes only require the agent to perform a single swing to reach the top. We can see this in figure 5.12, where we can clearly identify a single peak in Q-value gradient. Finally, sometimes the agent is initialised at an angle high enough where minimal effort is required to reach an upright position. We can see this in figure 5.13, where the gradient information is relatively flat.

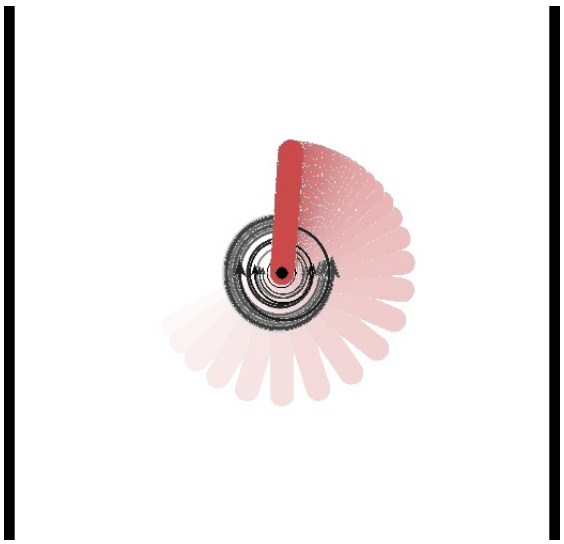


(a) Expert panorama view: Two swings

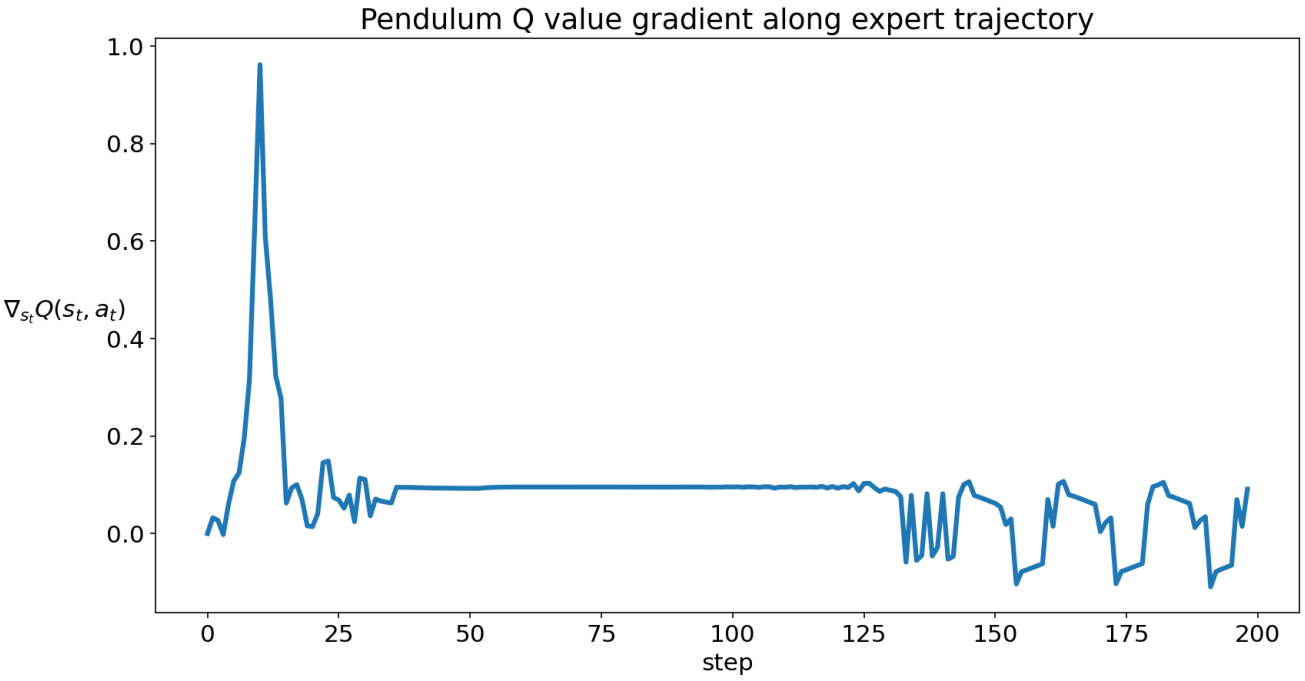


(b) Expert gradient view: Two swings

Figure 5.11: Expert Pendulum strategy: Two swings



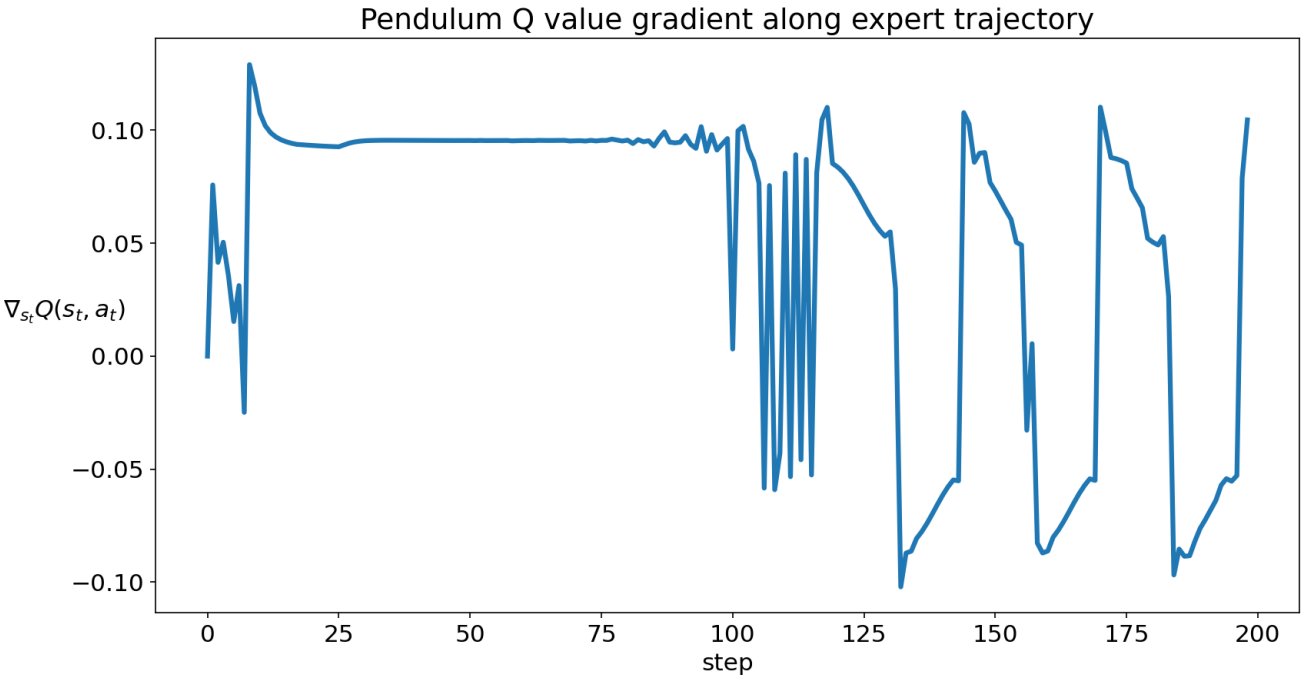
(a) Expert panorama view: One swing



(b) Expert gradient view: One swing

Figure 5.12: Expert Pendulum strategy: One swing

(a) Expert panorama view: No swings



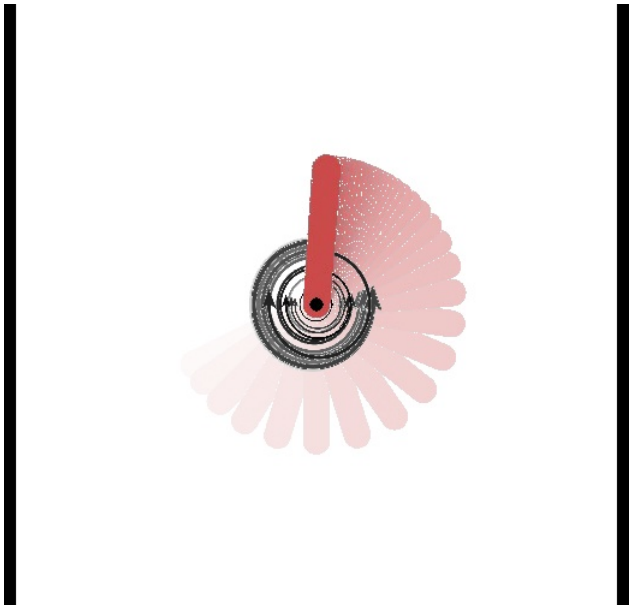
(b) Expert gradient view: No swings

Figure 5.13: Expert Pendulum strategy: No swings

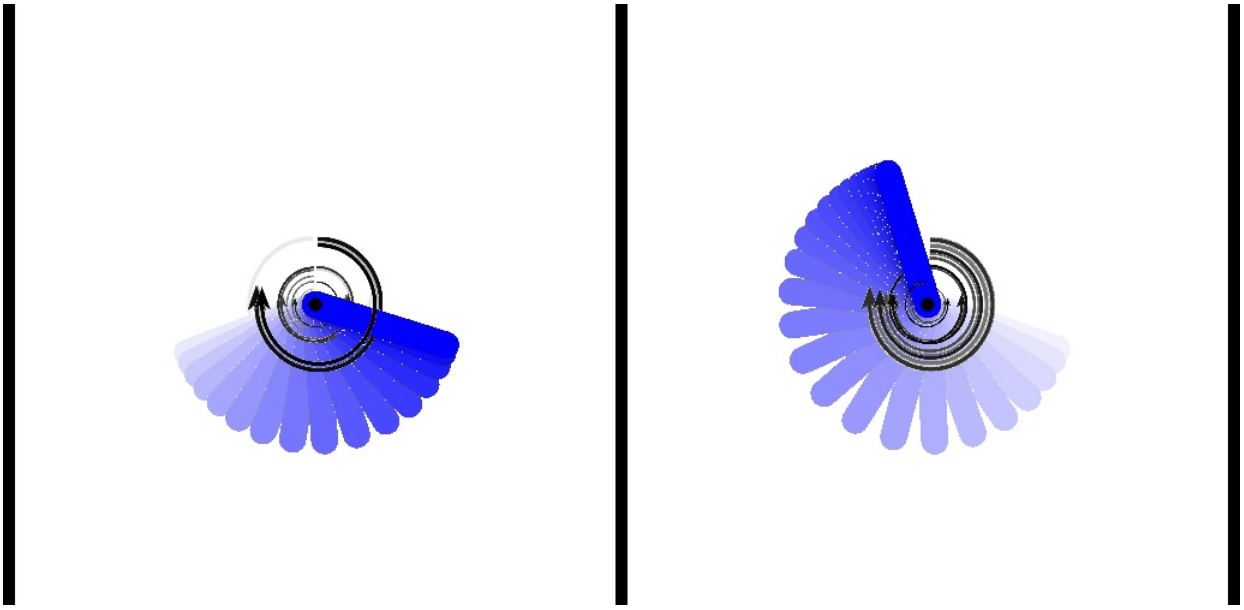
One of the resulting trained agent's behaviour compared to the expert's behaviour can be seen in figure 5.14. In some cases where the expert required to perform two swings to reach the top, we notice that our agent ends up just copying the expert's strategy. This could be a result of poor hyperparameter choices or too few data or training. However what we think the most likely explanation is that this might be due to the fact that there is no other way to solve the task when two swings are required. The hyperparameters used in this experiment can be found in table 5.4.

Data Generation Hyperparameters	**Value**
Expert Trajectories Extracted	10000
Resampling Repetition	1000
Random Network Distillation Hyperparameters	**Value**
Network Dimensions	[6, 256, 256, 256, 1024]
Network Activations	[ReLU, ReLU, ReLU, Identity]
Training Epochs	1000
Learning Rate	1e-3
Batch Size	1024
Discriminator Hyperparameters	**Value**
Network Dimensions	[6, 256, 256, 256, 1]
Network Activations	[ReLU, ReLU, ReLU, Sigmoid]
Training Epochs	100
Learning Rate	1e-3
Batch Size	256
RL Agent Hyperparameters	**Value**
Policy Network Dimensions	[6, 256, 256, 1]
Policy Network Activation (Gaussian Net)	[ReLU, ReLU, Identity]
Q-Network Dimensions	[7, 256, 256, 1]
Q-Network Activations	[ReLU, ReLU, Identity]
Actor & Critic Learning Rate	1e-3
SAC exploration parameter α	0.2
C_{ASL}	-30

Table 5.4: Pendulum experiment hyperparameters



(a) Pendulum expert behaviour



(b) Pendulum agent behaviour

Figure 5.14: Pendulum agent vs. expert behaviour, panorama views[a]

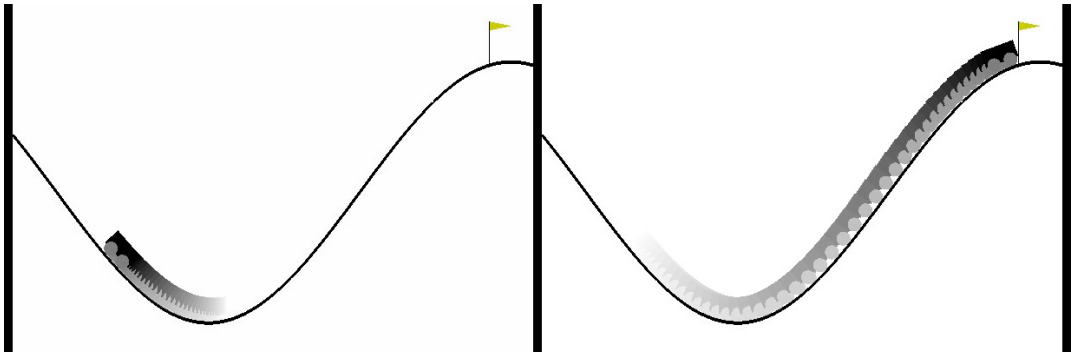
[a] `https://github.com/danielbairamian/Spotter/blob/main/Results/Pendulum.md`

5.5 Training a second agent

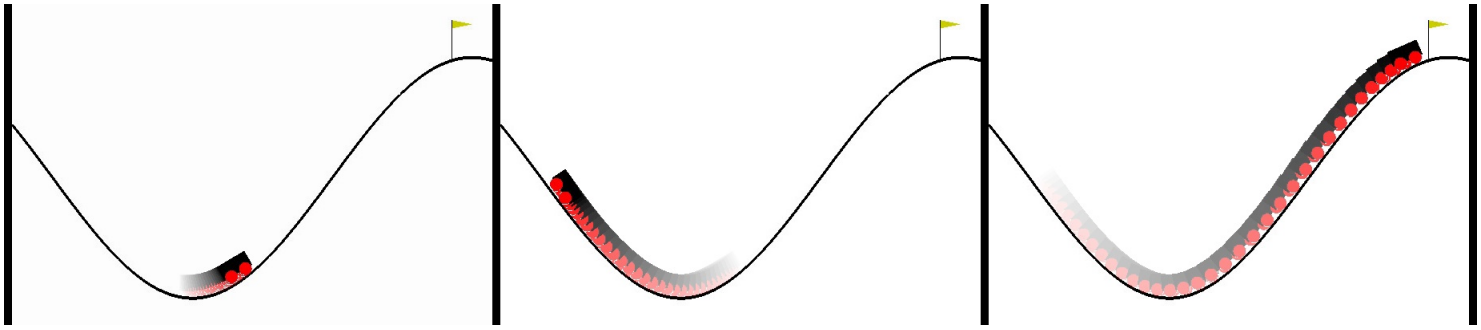
We perform additional experiments on some of the previously mentioned environments, by training a second agent. Once we fully train our first agent, we now treat it, and the original expert, as two experts. We extract trajectories from the new agent, and combine them with the existing trajectories from the original expert. Once the two sets are combined we apply our method without any additional change, by treating the combined dataset as a single expert.

5.5.1 MountainCar: Second Agent

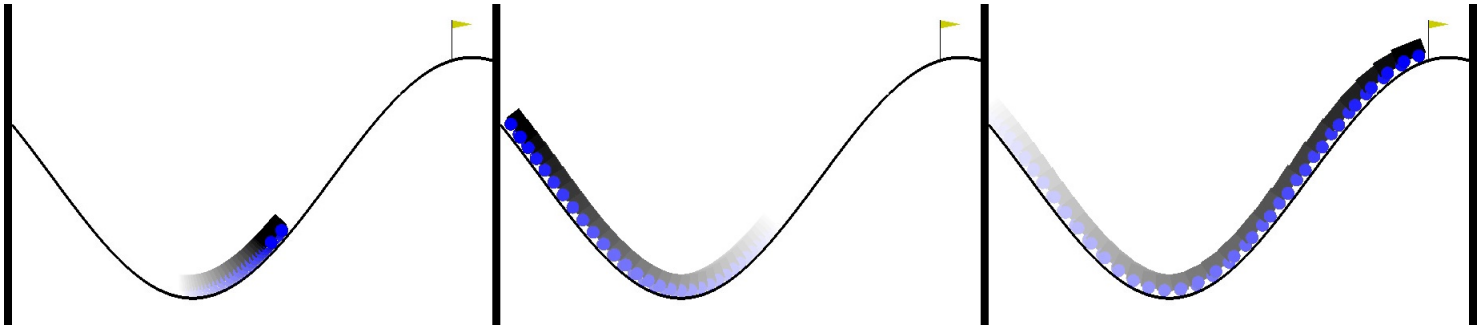
The first experiment we run is on the MountainCarContinuous environment, without any modification to the environment. The resulting trained agent's behaviour compared to the two experts' behaviour can be seen in figure 5.15. The hyperparameters used in this experiment can be found in table 5.5.



(a) MountainCar expert 1 behaviour



(b) MountainCar expert 2 behaviour



(c) MountainCar agent behaviour

Figure 5.15: MountainCar two experts vs. agent behaviour, panorama views[a]

[a] `https://github.com/danielbairamian/Spotter/blob/main/Results/MountainCar2.md`

Data Generation Hyperparameters	**Value**
Expert 1 Trajectories Extracted	1000
Expert 2 Trajectories Extracted	1000
Resampling Repetition	1000
Random Network Distillation Hyperparameters	**Value**
Network Dimensions	[2, 256, 256, 256, 1024]
Network Activations	[ReLU, ReLU, ReLU, Identity]
Training Epochs	1000
Learning Rate	1e-3
Batch Size	1024
Discriminator Hyperparameters	**Value**
Network Dimensions	[2, 256, 256, 256, 1]
Network Activations	[ReLU, ReLU, ReLU, Sigmoid]
Training Epochs	100
Learning Rate	1e-3
Batch Size	256
RL Agent Hyperparameters	**Value**
Policy Network Dimensions	[2, 256, 256, 1]
Policy Network Activation (Gaussian Net)	[ReLU, ReLU, Identity]
Q-Network Dimensions	[3, 256, 256, 1]
Q-Network Activations	[ReLU, ReLU, Identity]
Actor & Critic Learning Rate	1e-3
SAC exploration parameter α	0.2
C_{ASL}	-25

Table 5.5: MountainCar second agent experiment hyperparameters

5.5.2 FourRooms (Hacked)

The second experiment we run is on the FourRooms environment, where we introduce a collision bug on the intersection of the four walls in the center of the environment. The goal of this experiment is to demonstrate some of the use case of our method, which is finding exploits in environments. Since there are two clear strategies available, introducing a third agent would have one of two outcomes: The first outcome is that the agent ends up either copying one of the experts or simply not solving the task, since there are no other solutions available to it. The second outcome is that there is in fact a third solution available, through an exploit or some design oversight, in our case the collision bug. The collision bug in our case is emulated by letting the agent go from the bottom left room, directly to the top right room, if it is at a precise position almost exactly on the intersection, with an angle between zero and $\frac{\pi}{2}$, meaning in the direction of the intersection. We should note that this collision bug did not exist in the previous FourRooms experiment. While it would be possible for a regular agent without our method to find the bug, there would be no guarantee that the bug would have been found. The idea here is to show that we can somewhat guarantee the detection of an exploit, if that exploit leads to a solution. The resulting trained agent's behaviour compared to the two experts' behaviour can be seen in figure 5.16. The hyperparameters used in this experiment can be found in table 5.6.

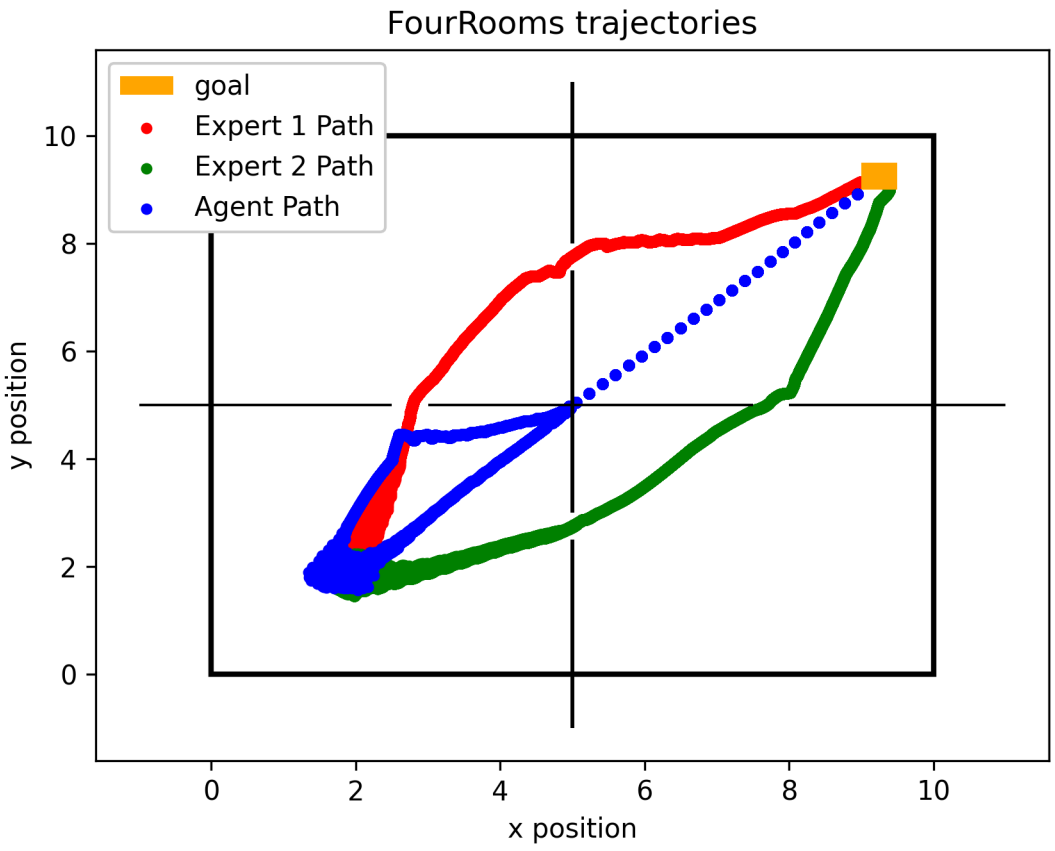


Figure 5.16: FourRooms two experts vs. agent behaviour, projected view[a]

[a] `https://github.com/danielbairamian/Spotter/blob/main/Results/FourRooms2.md`

Data Generation Hyperparameters	**Value**
Expert 1 Trajectories Extracted	1000
Expert 2 Trajectories Extracted	1000
Resampling Repetition	1000
Random Network Distillation Hyperparameters	**Value**
Network Dimensions	[2, 256, 256, 256, 1024]
Network Activations	[ReLU, ReLU, ReLU, Identity]
Training Epochs	1000
Learning Rate	1e-3
Batch Size	1024
Discriminator Hyperparameters	**Value**
Network Dimensions	[2, 256, 256, 256, 1]
Network Activations	[ReLU, ReLU, ReLU, Sigmoid]
Training Epochs	100
Learning Rate	1e-3
Batch Size	256
RL Agent Hyperparameters	**Value**
Policy Network Dimensions	[2, 256, 256, 1]
Policy Network Activation (Gaussian Net)	[ReLU, ReLU, Identity]
Q-Network Dimensions	[3, 256, 256, 1]
Q-Network Activations	[ReLU, ReLU, Identity]
Actor & Critic Learning Rate	1e-3
SAC exploration parameter α	0.2
C_{ASL}	-50

Table 5.6: FourRooms Hacked experiment hyperparameters

Chapter 6

Discussion and Conclusion

We presented our novel framework for identifying strategic behaviour in fully trained agents. Our method is primarily based on analysing the change in directional Q-value gradients observed along an episode. We then use these identified strategic behaviours as adversarial examples for a new agent, with the goal of learning a new unique strategy. Our experimental results show that our method is successful at correctly identifying strategic events by showing peaks of gradients at key moments, as well as correctly encouraging a new agent to learn new unique behaviour. Our experiments also show that our method is capable of using multiple agents as experts, making it possible to exhaustively search for all possible unique solutions for a given task.

6.1 Limitations and Future Work

One key limitation of our framework is that training multiple agents cannot be done in parallel, and requires sequential training. An interesting extension of this work would be to make the training of multiple agents parallelisable. As opposed to training one agent at a

time, we could imagine some training regiment similar to what we see in GANs, where both the expert and the agent learn together. For an arbitrary number of agents K, each agent would treat the remaining $K - 1$ agents as adversarial experts.

Another avenue of future work is to see how our method would deal with large dimension state spaces. We have only trained on relatively low dimensional environments. It would be interesting to extend our work to higher dimensional state spaces, or even environments that use raw pixel state spaces. This could pose an interesting challenge for our framework, as the uniqueness of a state space in higher dimension could be tougher to estimate. Our method would most likely then benefit with some sort of representation learning [30] of the state space.

Furthermore, defining a strategy when dealing with partially observable MDPs (POMDPs) would also be an interesting line of work. As the information received could be inconsistent across observation, the use of architectures like LSTMs [31] and Transformers [32] would most likely be required. The use of these recurrent units could also give a more sophisticated definition of a strategy allowing for more complex solutions to the given task, as the sequence of states used to defined a key moment would increase from one transition to an arbitrary amount, likely controlled by some hyperparameter.

Bibliography

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.

[2] B. Eysenbach, A. Gupta, J. Ibarz, and S. Levine, "Diversity is all you need: Learning diverse skills without a reward function," 2018.

[3] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," *CoRR*, vol. abs/1502.05477, 2015.

[4] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems* (S. Solla, T. Leen, and K. Müller, eds.), vol. 12, MIT Press, 2000.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[6] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016.

[7] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," 2018.

[8] M. Bain and C. Sammut, "A framework for behavioural cloning," in *Machine Intelligence 15, Intelligent Agents [St. Catherine's College, Oxford, July 1995]*, (GBR), p. 103–129, Oxford University, 1999.

[9] S. Ross, G. J. Gordon, and J. A. Bagnell, "No-regret reductions for imitation learning and structured prediction," *CoRR*, vol. abs/1011.0686, 2010.

[10] J. Ho and S. Ermon, "Generative adversarial imitation learning," *CoRR*, vol. abs/1606.03476, 2016.

[11] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," 2014.

[12] R. Bellman, "Dynamic programming," *Science*, vol. 153, no. 3731, pp. 34–37, 1966.

[13] R. Sutton, "Learning to predict by the method of temporal differences," *Machine Learning*, vol. 3, pp. 9–44, 08 1988.

[14] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, May 1992.

[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[16] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, Oct 1986.

[17] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems* (S. Solla, T. Leen, and K. Müller, eds.), vol. 12, MIT Press, 2000.

[18] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," in *Advances in Neural Information Processing Systems* (S. Solla, T. Leen, and K. Müller, eds.), vol. 12, MIT Press, 2000.

[19] A. L. Strehl and M. L. Littman, "An analysis of model-based interval estimation for

markov decision processes," *Journal of Computer and System Sciences*, vol. 74, no. 8, pp. 1309–1331, 2008. Learning Theory 2005.

[20] J. Z. Kolter and A. Y. Ng, "Near-bayesian exploration in polynomial time," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, (New York, NY, USA), p. 513–520, Association for Computing Machinery, 2009.

[21] Y. Burda, H. Edwards, A. Storkey, and O. Klimov, "Exploration by random network distillation," 2018.

[22] M. A. Kramer, "Nonlinear principal component analysis using autoassociative neural networks," *AIChE Journal*, vol. 37, no. 2, pp. 233–243, 1991.

[23] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," 2017.

[24] D. Krueger, N. Ballas, S. Jastrzebski, D. Arpit, S. Kanwal, T. Maharaj, E. Bengio, A. Fischer, and A. Courville, "Deep nets don't learn via memorization," 01 2017.

[25] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.

[26] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[27] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[28] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[29] M. Janner, Q. Li, and S. Levine, "Reinforcement learning as one big sequence modeling problem," *arXiv preprint arXiv:2106.02039*, 2021.

[30] Y. Bengio, A. C. Courville, and P. Vincent, "Unsupervised feature learning and deep learning: A review and new perspectives," *CoRR*, vol. abs/1206.5538, 2012.

[31] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*,

vol. 9, no. 8, pp. 1735–1780, 1997.

[32] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017.