Appendices

A DEFINITIONS

Definition 1.1: Domain vocabulary & labeling function

A *vocabulary* is a set of environment or domain specific (boolean) propositional symbols, $\mathcal{P}$, with $\mathbb{R} \in [0, 1]$ for each state $s \in \mathcal{S}$. A *labeling function* is a function $\mathcal{F}_L(t) : \mathcal{S} \times \mathcal{A} \times \mathcal{S}' \rightarrow 2^{\mathcal{P}}$, that maps state observations to truth assignments over the vocabulary $\mathcal{P}$ for the time-step t.

Definition 1.2: MDP with a reward machine (MDP-RM)

A Markov decision process with a reward machine is a tuple $\Psi = \langle S, A, p, \gamma, \mathcal{P}, \mathcal{F}_L, U, u_0, G, \delta_u, \delta_r \rangle$ where S, A, p, and γ are defined as in an MDP, $\mathcal{P}$ is a set of domain specific propositional symbols, $\mathcal{F}_L$ is a labelling function (Def. 1.1), and U, u_0, G, δ_u and δ_r are defined as in a reward machine, where U is a set of FSA states, u_0 is the initial state, G is a set of terminal or goal states, and δ_u, δ_r are state and reward transition functions.

B REWARD MACHINE: EXPANDED BACKGROUND AND MOTIVATION

Reward machines were introduced by Icarte et al. Icarte et al. (2018) as a type of finite state machine (FSM) that supports the specification of reward functions while exposing reward function structure. As a form of FSM, reward machines have the expressive power of a regular language and as such, support loops, sequences and conditionals. Additionally, it supports expression of temporally extended linear-temporal-logic (LTL) and non-Markovian reward specification, where the underlying reward received by an agent from the environment is not Markovian with respect to the state.

When an environment dynamics is specified using a reward machine (RM), as an agent acts in the environment, moving from state to state, it also moves from state to state within a reward machine (as determined by high-level events detected within the environment).

After every transition, the reward machine outputs the reward function the agent should use at that time. For example, we might construct a reward machine for '*delivering mail to an office*' using two states. In the first state, the agent does not receive any rewards, but it moves to the second state whenever it gets the mail. In the second state, the agent gets rewards after delivering the mail. Intuitively, defining rewards this way improves *scale efficiency* as the agent knows that the problem consists of two stages and might use this information to speed up learning.

Using the above discussion, we can define a standard RM using mathematical formalism as the following definition 2.1.

Definition 2.1: A standard Reward Machine (RM)

Given a set of propositional symbols $\mathcal{P}$, a set of (environment) states S, and a set of actions A, a reward machine is a tuple: $\mathcal{R}_{\mathcal{P}S\mathcal{A}} = \langle U, u_0, F, \delta_u, \delta_r \rangle$ where U is a finite set of states $u_0 \in U$ is an initial state, F is a finite set of terminal states (where $U \cap F = \emptyset$), δ_u is the state-transition function s.t. $\delta_u : U \times 2^{\mathcal{P}} \rightarrow U \cup F$, and δ_r is the state-reward function, s.t. $\delta_r : U \rightarrow [S \times A \times S \rightarrow \mathbb{R}]$.

B.1 MOTIVATION FOR USING RM IN IRL+POMDP SETTING

Agents in modern, real-world RL datasets (e.g. robotics, embodied-ai Shridhar et al. (2020)) often, if not always, are required to perform tasks that are long-horizon with compositional and/or logical underlying reward structure. The main motivation of SMIRL is to show that imbuing the agent with *apriori* (approximate) structure of the latent reward associated with a task allows solving complex tasks that are hard to learn from only demonstrative expert trajectory data. This motivation makes the IRL+POMDP problem setting ideal for purposes in this paper.

The effectiveness of automata-based memory has long been recognized in the POMDP literature Cassandra et al. (1994), where the objective is to find policies given a complete specification of the environment. The overarching idea in approaches under this umbrella is to encode policies using Finite State Controllers (FSCs), which are FSMs with states associated with one primitive action, and the transitions are defined in terms of low-level observations from the environment. During environment interaction, the agent always selects the action associated with the current state in the FSC. Using such automata-based memory was leveraged to work in the RL setting in work by Meuleau et al. Meuleau et al. (2013b) by exploiting policy gradient to learn policies encoded as FSCs.

RMs can be considered as a generalization of FSC as they allow for transitions using conditions over high-level events and associate complete policies (instead of just one primitive action) to each state. This allows RMs to leverage existing deep RL methods to learn policies from low-level inputs, such as images, which is not achievable by other automata-based approaches like Meuleau et al. (2013b). That said, learning FSMs using other ontologies (e.g. Xu et al. (2020); Zhang et al. (2019)) do exist in concurrent literature. Discussion and delineation with such works are further discussed in the 'Related Work' section (S. C).

C RELATED WORK

Augmenting memory using of Recurrent Neural Networks (RNNs) in combination with policy gradient Jaderberg et al. (2016); Mnih et al. (2016); Schulman et al. (2017) is a common approach in state-of-the-art (SOTA) approaches in the RL+POMDP domain. Other approaches use external neural-based memories Oh et al. (2016); Khan et al. (2017); Hung et al. (2019). Model-Based Bayesian RL and extension approaches Doshi-Velez et al. (2013); Ghavamzadeh et al. (2015); Poupart & Vlassis (2008) under partial observability provide a small binary memory to the agent and a special set of actions to modify it. The motivation and idea behind our work here are largely orthogonal to these aforementioned approaches.

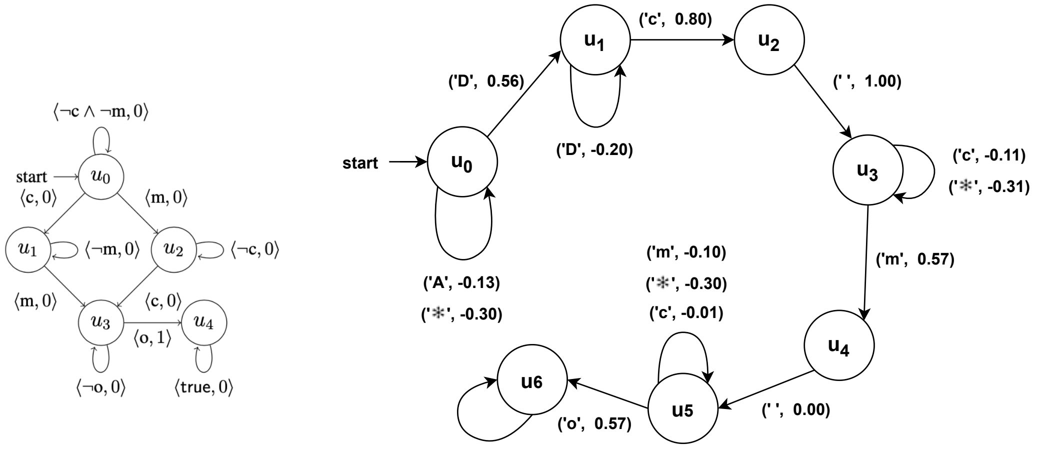
The work that is closest to ours is by Icarte et al. Toro Icarte et al. (2019) – where the authors learn RMs for partially observable RL tasks from trajectories. However, efficacy of the approach is shown in 2D discrete domains, with the authors noting the challenge of showcasing them in 3D continuous domain due to the intractability of state space explosion. While Toro Icarte et al. (2019) motivates our choice to use RM as the chosen structural motif architecture (as opposed to other available FSA ontologies), but to the best our knowledge, learning the motif on continuous 3D domains with complex logic (see SMIRL Algorithm) is not undertaken in prior works. While this difference can be argued as meagre for 2D toy like domains, but is significant for continuous domains, because 'Tabu search' for state space is computationally infeasible in such complex domains. Thus, this insight is more applicable in solving IRL in real-world robotic or embodied-ai domains commensurate with the motivation of our work.

Another related sub-domain of works include literature on *learning logic and automata from demonstrations*. These works by problem definition is slightly different to the IRL problem domain we tackle here. The works in this area (e.g. by Vazquez et al. Vazquez-Chanlatte et al. (2018; 2021)) infers Boolean non-Markovian rewards, or logical properties of available traces (aka. demonstrations). This is achieved by learning probabilistic densities of demonstrations over an **existing, apriori** knowledge pool of candidate specifications. In essence, it is a specifications matching problem, or searching for the most probable specification in a pool of candidate specifications. Our work is **orthogonal to these** in the aspect that we do not have or define the task labels or any apriori structure of the specification.

D OFFICE GRIDWORLD DOMAIN

D.1 DETAILED ILLUSTRATION OF STRUCTURAL MOTIF LEARNING (SMIRL)

Here we examine and illustrate the SMIRL learning algorithm 1 using *Task 3: 'Fetch and deliver coffee and mail'*. The Fig. 8 juxtaposes the perfect reward structure with a FSA structural motif learned using the SMIRL 1 algorithm. Although the learned structure is not optimal (with 6 FSA states as opposed to optimal 4), but if converges to the desired target state.



(a) State diagram of a **perfect** FSA

(b) State diagram of **learned** FSA

Figure 8: Qualitative Evaluation in the *Office GridWorld*. **Left**: Fig. 8a shows the **perfect** reward (FSA) structure for *Task 3: Fetch & Deliver 'Coffee' and 'Mail'* requiring the delivery of both coffee ('`c`') and mail ('`m`') to the office ('`o`') starting with position '`A`' on the map and initial FSA state u_0. **Right**: Fig. 8b shows the **learned** FSA structural motif and weights. The state transition arrow labels (e.g. $u_1 \to u_2$: `('c', 0.80)`)indicate the true propositional symbols [Def. 1.1] – from an agent action a resulting in state transition from s to s', and the underlying FSA to state $u' = \delta_u(u, \mathcal{F}_L(s, a, s')$ – and the correponding reward $r(s, a, s'$, where $r = \delta_r(u)$. We can see some artifacts of the SMIRL algorithm 1 here with nodes u_2 and u_4. The first transition from u_0 to u_1 comes from the fact that the expert trajectories go through `D` while fetching coffee `c`.

Lemma 1: MDP, MDP-RM expected reward equivalency

Given an MDP-RM $\Psi = \langle S, A, p, \gamma, \mathcal{P}, \mathcal{F}_L, U, u_0, G, \delta_u, \delta_r \rangle$ let $\mathcal{M}_\Psi = \langle S', A', r', p', \gamma' \rangle$ be the MDP defined such that $S' = S \times U, A' = A, \gamma' = \gamma$, $p'(\langle s', u' \rangle \mid \langle s, u \rangle, a) = \begin{cases} p(s' \mid s, a) & \text{if } u' = \delta_u(u, \mathcal{F}_L(s')) \\ 0 & \text{otherwise} \end{cases}$, and $r'(\langle s, u \rangle, a, \langle s', u' \rangle) = \delta_r(u, u')(s, a, s')$.

Then any policy for $\mathcal{M}_\Psi$ achieves the same expected reward in Ψ, and vice versa.

D.2 IMPLEMENTATION DETAILS

Generating expert trajectories The process flow for generating expert trajectories in the gridworld domain entails first to train an expert policy, π_e using a perfect FSA reward structure (Fig. 8a), then using the expert policy to generate $\mathcal{D}_e$

The final form of the expert demonstrations is represented by series of state-action transitions, i.e., $(s_1, a_1, s_2, a_2, ..., s_T)$. The actions are represented by single integers from 0 to 3. For the low-level state representations, we include both high-level features of the environment (one-hot encoding of one of the high-level positions, e.g., `{A, B,...,D, c, m,..}` etc.) and the low-level positions (one-hot encoding of the 108 grids of the *Office GridWorld* environment).

The following points detail various conditions adhered to while $\mathcal{D}_e$ generation:

1. There are K max steps (episode horizon) possible in an episode.
2. An episode can end in (t « K) steps if a 'done' or 'game over' condition is hit (like stepping on obstacles).
3. Once an optimal trajectory is traversed for 1 round trip (e.g. *Task 4 – 'Patrol ABCD'*: $u_0 \to u_1 \to u_2 \to u_3$), the agent receives a reward of 1.
4. After that, the agent takes k random steps.

5. If not terminated in k steps, `get_optimal_action()` is invoked (from whichever random position the agent is in, creating another successful reward trip completion.
6. The above step is repeated until max K steps are reached

E REACHER DOMAIN DETAILS

In this section we present pertinent details about the experiments conducted on the continuous MuJoCo Reacher domain (Sec. 4.3).

E.1 REACHERDELIVERY-V0: MODIFIED REACHER DOMAIN

We modify the classic OpenAI Gym's Brockman et al. (2016) MuJoCo Todorov et al. (2012) Reacher environment to our purposes here. The original Reacher environment is a two-jointed robot arm, and the goal is to move the robot's end effector (called *fingertip*) close to a target that is spawned at a random position. The *action space* consists of an *action* (a, b) that represents the torques applied at the hinge joints. The *observation space* consists of the sine, cosine angles of the two arms, coordinates of the target, angular velocities of the arms and a 3D distance vector between the target and the reacher's fingertip. The *reward* consists of two parts: i. *reward_distance* (R_d): a measure of how far the fingertip of the reacher (the unattached end) is from the target, with a more negative value assigned with increasing distance; ii. *reward_control* (R_c): a negative reward for penalising actions that are too large. It is measured as the negative squared Euclidean norm of the action, i.e. as $-\sum action^2$. The total reward returned is: $reward = reward\_distance + reward\_control$ ($R = R_d + R_c$).

ReacherDelivery The following code snippet shows how the MuJoCo asset file was modified to add four target locations as colored balls.

Excerpt from `reacher_delivery.xml` file showing added red ball as a target location

```
<!-- RED -->
<body name="red" pos="0 0 0.01">
      <joint armature="0" axis="1 0 0" damping="0" limited="true" name="
          red_x" pos="0 0 0" range="-.27 .27" ref="0" stiffness="0" type
          ="slide"/>
      <joint armature="0" axis="0 1 0" damping="0" limited="true" name="
          red_y" pos="0 0 0" range="-.27 .27" ref="0" stiffness="0" type
          ="slide"/>
      <geom conaffinity="0" contype="0" name="red" pos="0 0 0" rgba="0.9
          0. 0. 0.3" size=".02" type="sphere"/>
</body>
```

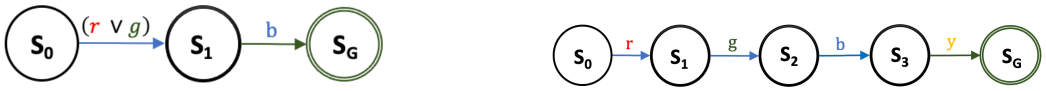
We extended the MuJoCo Reacher class with an environment MDP wrapper and added target goals. Following an agent step in the environment, the state proposition labels are updated using the labelling function. The following code snippet exemplifies this, where a target color location proposition is labeled 'true' if the fingertip is within a threshold distance from it.

The labelling function $\mathcal{F}_L$ for `ReacherDelivery` domain

```
true_props = []
if dist_red < 0.02:
   true_props.append('r')
if dist_green < 0.02:
   true_props.append('g')
if dist_blue < 0.02:
   true_props.append('b')
if dist_yellow < 0.02:
   true_props.append('y')
if cancel:
   true_props.append('c')
```

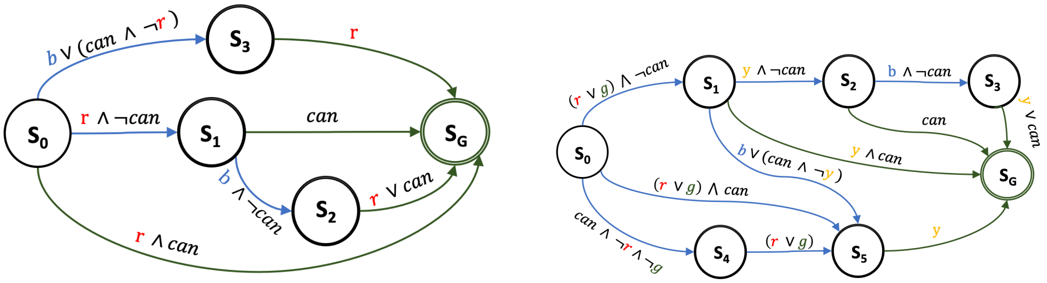
E.2 TASKS REWARD FSA STRUCTURES

Figure 9 shows the (perfect) reward structure for the four tasks along with the task name and the natural language description of the task. The tasks are presented in order of increasing complexity.



(a) **OR**: Touch the red or green ball, then the blue ball

(b) **Sequential**: Touch the red, green, blue and yellow ball in order



(c) **IF**: Touch blue, unless *cancel*; then touch the red ball

(d) **Composite**: Touch red or green, then blue, unless *cancel*, then go to yellow

Figure 9: The four increasingly difficult tasks (`OR, Sequential, IF, Composite`) with the corresponding task descriptions and FSA states

E.3 Implementation Details

This section outlines the various implementation details for the *ReacherDelivery* domain experiments (Sec. 4.3).

Training Details: We use SAC as the underlying RL algorithm throughout. The policy network is a tanh squashed Gaussian with mean and standard deviation parameterized by a (64, 64) ReLU MLP with two output heads. The Q-network is a (64,64) ReLU MLP. For optimization we use Adam with learning rate of 0.003 for both the Q-network and policy network. The replay buffer size was 10000 and we used batch size of 256.

For the baselines f-IRL and MaxEntIRL, we used the f-IRL Ni et al. (2020) authors' official implementation[2]. f-IRL and MaxEntIRL require an estimation of the agent state density. We use kernel density estimation to fit the agent's density, using Epanechnikov kernel with a bandwidth of 0.2 for pointmass, and a bandwidth of 0.02 for Reacher. At each epoch, we sample 1000 trajectories (30000 states) from the trained SAC to fit the kernel density model.

Generating Expert Trajectories: For expert trajectories generation, we first train expert policies imbued with perfect reward structure using SAC for each of the tasks. Fig. 10 shows the training curve and violin plot expert return density curves of training.

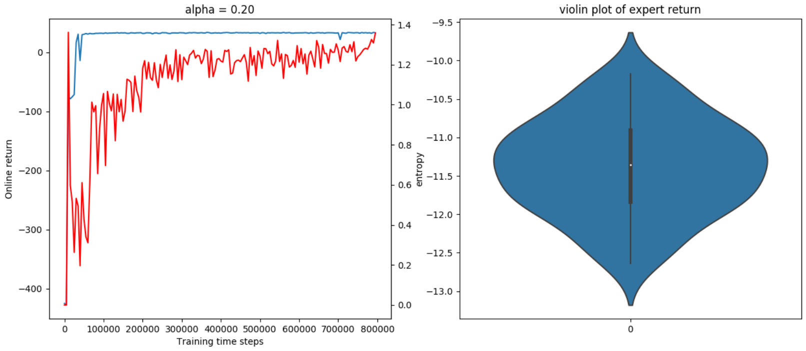
SAC uses the same policy and critic networks with the learning rate set to 0.003. We train using a batch size of 100, a replay buffer of size 1 million, and set the temperature parameter α to be 0.2. The policy is trained for 1 million timesteps on ReacherDelivery. All algorithms are tested on 16 trajectories collected from the expert stochastic policy.

Evaluation We compare the trained policies by the baselines (f-IRL< MaxEntIRL) and SMIRL by computing their returns according to the ground truth return on the ReacherDelivery environment.

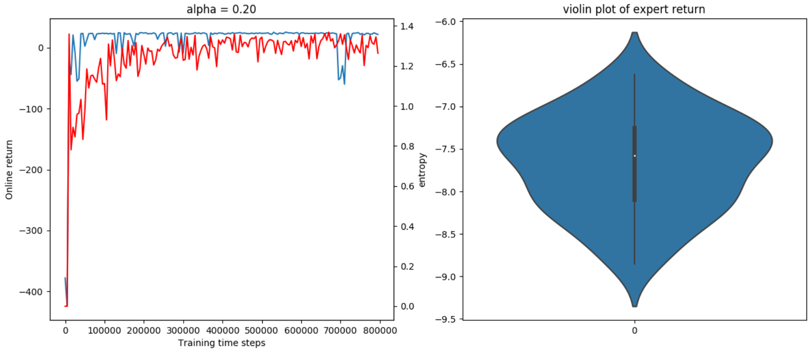
Computational Complexity In general, *ceteris paribus*, SMIRL is more sample efficient (i.e. converges to a solution faster) than baseline IRL approaches. Complexity can arise from two factors: i. the domain complexity, say 2D discrete vs. continuous, and ii. The size and complexity of the reward machine (or, FSA) structural motif. We see implications of both in the paper. First, the sample complexity increases with increasing domain complexity (office gridworld vs. Reacher) and this is intuitive. From our experiments, for the second kind of complexity, i.e. with more intricate underlying RM structure, the bottle-neck seems to emanate from the ability to learn the structure. We see that for the hardest (composite) task, the structure was not learned, and increasing sample complexity wouldn't have helped (it would saturate to a suboptimal level). If the structure is learnable, then

---

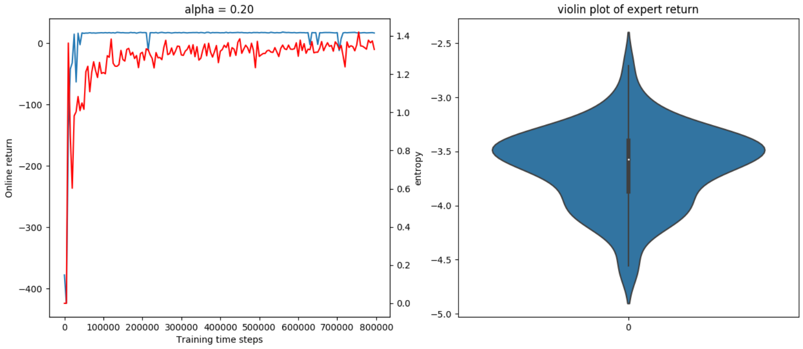
[2] `https://github.com/twni2016/f-IRL`



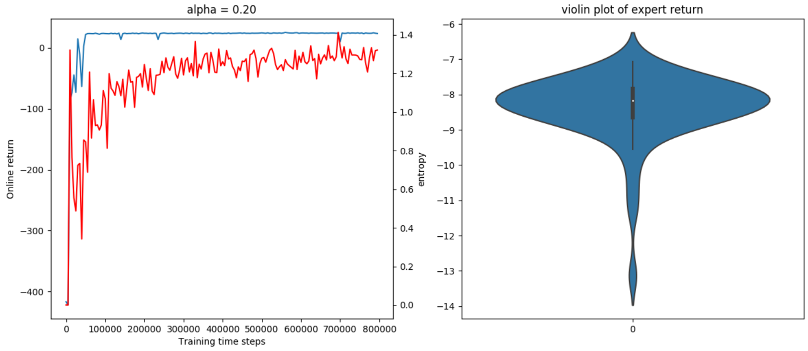
(a) Task: **Sequential** (nF: 5)



(b) Task: **IF** (nF: 5)



(c) Task: **OR** (nF: 3)



(d) Task: **Composite** (nF: 7)

Figure 10: The training curves for expert policy training. The experts were imbued with perfect state based reward structure, where the number of states are given within brackets as 'nF' value. **Left**: The blue curve shows reward and the red curve shows average entropy. **Right**: The violin plot of expert return density.

sample complexity scales with the complexity of the task structure (e.g. ’sequential’ vs. ’OR’ in Fig. 6).

F BASELINE OBJECTIVE FUNCTIONS

f-IRL We train the three variants of f-IRL: forward KL (fkl), reverse KL (rkl) and Jansen-Shannon (js) that represents the f-divergence metric used by the f-IRL algorithm.

f-divergence Ali & Silvey (1966) is a family of distribution divergence metric, which generalizes forward/reverse KL divergence. Formally, let P and Q be two probability distributions over a space Ω, then for a convex and Lipschitz continuous function f such that $f(1) = 0$, the f-divergence of P from Q is defined as:

$$D_f(P \| Q) := \int_\Omega f\left(\frac{dP}{dQ}\right) dQ \tag{10}$$

f-IRL uses state marginal matching by minimizing the f-divergence objective:

$$L_f(\theta) = D_f\left(\rho_E(s) \| \rho_\theta(s)\right) \tag{11}$$

This objective is realized by computing the exact (analytical) gradient of the preceding f-divergence objective w.r.t. the reward parameters θ.

Theorem 6.1: f-divergence analytic gradient (from Ni et al. (2020))

The analytic gradient of the f-divergence $L_f(\theta)$ between state marginals of the expert (ρ_E) and the soft-optimal agent w.r.t. the reward parameters θ is given by:

$$\nabla_\theta L_f(\theta) = \frac{1}{\alpha T} \operatorname{cov}_{\tau \sim \rho_\theta(\tau)} \left(\sum_{t=1}^{T} h_f\left(\frac{\rho_E(s_t)}{\rho_\theta(s_t)}\right), \sum_{t=1}^{T} \nabla_\theta r_\theta(s_t) \right)$$

where $h_f(u) \triangleq f(u) - f'(u)u$, $\rho_E(s)$ is the expert state marginal and $\rho_\theta(s)$ is the state marginal of the soft-optimal agent under the reward function r_θ, and the covariance is taken under the agent’s trajectory distribution $\rho_\theta(\tau)$.[2]