In *Proceedings of the 37th Conference on Artificial Intelligence* (AAAI 2023),
Washington D.C., USA, February 2023

The Perils of Trial-and-Error Reward Design: Misdesign through Overfitting and Invalid Task Specifications

Serena Booth[1,2,3], **W. Bradley Knox**[1,2,5], **Julie Shah**[3],
Scott Niekum[2,4], **Peter Stone**[2,6], **Alessandro Allievi**[1,2]
[1]Bosch, [2]The University of Texas at Austin, [3]MIT CSAIL,
[4]The University of Massachusetts Amherst, [5]Google Research, [6]Sony AI
{serenabooth, julie_a_shah}@csail.mit.edu, {bradknox,pstone}@cs.utexas.edu,
sniekum@cs.umass.edu, alessandro.allievi@us.bosch.com

Abstract

In reinforcement learning (RL), a reward function that aligns exactly with a task's true performance metric is often sparse. For example, a true task metric might encode a reward of 1 upon success and 0 otherwise. These sparse task metrics can be hard to learn from, so in practice they are often replaced with alternative dense reward functions. These dense reward functions are typically designed by experts through an ad hoc process of trial and error. In this process, experts manually search for a reward function that improves performance with respect to the task metric while also enabling an RL algorithm to learn faster. One question this process raises is whether the same reward function is optimal for all algorithms, or, put differently, whether the reward function can be overfit to a particular algorithm. In this paper, we study the consequences of this wide yet unexamined practice of trial-and-error reward design. We first conduct computational experiments that confirm that reward functions can be overfit to learning algorithms and their hyperparameters. To broadly examine ad hoc reward design, we also conduct a controlled observation study which emulates expert practitioners' typical reward design experiences. Here, we similarly find evidence of reward function overfitting. We also find that experts' typical approach to reward design—of adopting a myopic strategy and weighing the relative goodness of each state-action pair—leads to misdesign through invalid task specifications, since RL algorithms use cumulative reward rather than rewards for individual state-action pairs as an optimization target.

Code, data: github.com/serenabooth/reward-design-perils.

1 Introduction

In their authoritative introductory text on reinforcement learning, Sutton and Barto (2018) assert: "The reward signal is your way of communicating to the agent what you want achieved, not how you want it achieved." This statement implies that a reward function should exclusively encode the true task performance metric. Such metrics are often sparse: did the agent succeed at the task or not? Sparse reward functions are rarely used in practice, since it can be hard to learn from sparse signals (for examples, see Yu et al. (2020); Andrychowicz et al. (2020); Knox et al. (2021)). As such, the practice of reward design seldom adheres to this adage.[1] Instead, reward functions are typically designed through an ad hoc process of trial and error. In a survey of 24 expert RL practitioners, we found that 92% reported using trial and error to design their most recent reward function (Apdx. A). This finding echos the literature: Knox et al. (2021) found that, in a survey of RL for autonomous driving, all of the surveyed publications reported designing reward functions by trial and error. Despite the prevalence of trial-and-error reward design, the consequences of this process remain almost completely unexamined by the RL community. It is urgent and crucial for our community to understand the effects of this widespread practice, and ultimately to craft specific guidance for *practical* reward design.

When designing reward functions through trial and error, experts often optimize the reward function by manually searching for a reward function that meets both goals of maximizing the task performance metric and enabling an RL algorithm to learn quickly. This practice raises the question of whether a reward function that is effective with one algorithm can be ineffective with others. In other words, can a reward function be overfit to an algorithm? This question of overfitting in turn raises troubling questions about evaluation in RL. Overfitting a reward function to one RL algorithm undermines comparisons to another algorithm using the same reward function. Consider an ablation study which assesses whether an algorithmic component improves learning. If the reward function is overfit to the algorithm when the component is unablated, observing that learning performance decreases in the absence of the component may merely reflect that the reward function is overfit, giving no clear signal about whether the component is an improvement. Concerns about fair evaluation are already pervasive in RL, and are thought to limit RL's applicability outside of the laboratory (Ibarz et al. 2021). Most such concerns focus on hyperparameters, network architectures, and observed high variance, coupled with the high costs of experimentation (Henderson et al. 2018). This paper adds an additional perspective: that the oft-overlooked design process behind the reward function must also be considered for fair comparisons.

[1]Sutton and Barto disregard their own advice when designing a Dyna-Q+ agent. To encourage exploration, they replace the reward function r with $r + \kappa\sqrt{\tau}$, where κ is a hyperparameter and τ is the number of timesteps (Sutton and Barto 2018, p. 168).

To assess reward function overfitting, we first conduct computational experiments to test whether certain reward functions enable different RL algorithms and hyperparameters to perform better with respect to the true task performance metric. From these experiments, we find evidence that reward functions can indeed be overfit to a particular discount factor, learning rate, or algorithm: in such cases, changing the discount factor, learning rate, or algorithm significantly diminishes the task metric performance. Across numerous experiments, we find that when we rank reward functions by the learned policies' resultant task metric scores, these rankings are largely *uncorrelated* across experiment variations. Though the idea of reward function overfitting may be unsurprising to seasoned RL experts, the extent of this overfitting problem is nonetheless remarkable.

To learn about the implications of trial-and-error reward design, we also conducted a user study. One goal of this user study was to confirm that our computational experiments' findings correspond to practical effects in realistic RL settings. Specifically, we challenged 30 expert RL practitioners to choose an RL algorithm, hyperparameters, and a reward function to train the best agent they could, as measured by the cumulative task performance metric. The majority of experts overfit their reward functions to their choice of algorithms or hyperparameters (68%). More alarmingly, the majority of experts also constructed reward functions which failed to encode the task (53%)—meaning these reward functions encoded optimal policies which significantly deviate from the experts' intent, despite these tests being conducted in a simple gridworld environment. We then applied thematic analysis to qualitatively analyze experts' reward-design process, and we discovered that some types of reward misdesign stem from mismatched perspectives of what the reward function communicates. For the RL algorithm, reward is an additive component that is used to calculate discounted return—the evaluation metric. We find that experts instead typically view reward as a direct evaluation of the relative goodness of each state-action pair. This disparity contributes to reward function misdesign as a consequence of ad hoc trial-and-error reward design.

2 Related Work

Reward Shaping One of the known, common consequences of ad hoc reward design is reward shaping. In reward shaping, the reward function is overloaded to both communicate the underlying performance metric and guide an agent's learning toward a desired policy. Reward shaping can be designed in such a way that the optimal policy is unchanged (Ng, Harada, and Russell 1999). However, ad hoc reward shaping is known to be typically *unsafe*—meaning that a shaped reward function is likely to change the optimal solution to a given reinforcement learning task (Amodei et al. 2016; Knox et al. 2021). Our work affirms that ad hoc reward design amplifies this type of misdesign: the resulting optimal policies are often unrecognizable from the expert's known intent. Our work also contributes a new perspective on how trial-and-error reward design results in reward function overfitting, in which reward functions are unintentionally over-engineered for use with a specific algorithm.

Designing Rewards for Fast Learning Singh, Lewis, and Barto (2009) asked the philosophical question: where do rewards come from? They established a computational framework for quantifying the performance of reward functions, in which they assess whether 'intrinsic' motivation is helpful—i.e., whether reward functions benefit from rewarding subgoals. We build off of this work, especially the computational framework for assessing reward functions.

Similarly, Sowerby, Zhou, and Littman (2022) observe that certain reward functions result in faster learning, and they put forth principles of reward design in accordance with this observation. To find fast-learning reward functions, they use linear programming to construct reward functions which meet a correctness criteria of encoding the optimal policy. To test the fastness of learning from these reward functions, they assess how many training steps are needed for a Q-learning agent to converge to the optimal policy. The authors note this work is preliminary and has mostly been tested with a single Q-learning algorithm with fixed hyperparameters. Our work contributes a related perspective: fast learning may not be an intrinsic property of a good reward function, but also a consequence of the paired choice of algorithm and hyperparameters that were used to test that reward function.

AutoRL is another approach, which is gaining increasing traction (Niekum, Barto, and Spector 2010; Faust, Francis, and Mehta 2019; Chiang et al. 2019; Zheng et al. 2020; Wu et al. 2021; Parker-Holder et al. 2022). AutoRL frames RL as a meta-learning problem, in which the reward function should be learned, perhaps using an evolutionary method. Our work has interesting implications for AutoRL. Currently, these methods usually first optimize a reward function and then fix this function to optimize other RL design choices, such as the neural network architecture. Our work suggests this method—of first fixing the reward function and then optimizing other design choices—may be suboptimal relative to employing a joint optimization strategy.

Inferring Reward Functions Since specifying reward functions is both known to be hard and requires expertise, many research threads explore how to learn reward functions from intuitive signals like demonstrations (Ng, Russell et al. 2000; Ziebart et al. 2008), preferences (Christiano et al. 2017; Knox et al. 2022), and feedback (Knox and Stone 2009; MacGlashan et al. 2017). Inverse reward design is an approach that requires experts to specify reward functions but recognizes that these designed reward functions are only observations about the true goal. As such, inverse reward design works try to infer a true reward function based on these observations (Hadfield-Menell et al. 2017; Ratner, Hadfield-Menell, and Dragan 2018). Our work provides empirical support for this approach, as we find evidence that this assumed human behavior—of designing reward functions as observations and not as true problem specifications—is common in practice. He and Dragan (2021) similarly view reward design is an iterative process. Their work contributes a mechanism for surfacing environments where the reward incentivizes the wrong behavior to the human expert to support them in revising their reward function. Our work reinforces the importance of these types of debugging tools.

3 Preliminaries

Reinforcement Learning In RL, an agent learns a behavioral policy based on experience interacting with its environment. An RL task can be modeled by a Markov decision process (MDP), which is defined by a tuple $\langle S, A, T, \gamma, D_0, r \rangle$. S and A are the sets of states and actions, respectively. T is a transition function, $T : S \times A \times S \rightarrow [0, 1]$. γ is the discount factor and D_0 is the distribution of start states. Lastly, r is a reward function, $r : S \times A \times S \rightarrow \mathbb{R}$. An MDP$\backslash\langle \gamma, r \rangle$ is an MDP with neither a discount factor nor a reward function; we use this formulation for studying humans' reward design processes, wherein we ask humans to select the discount factor and reward function. Actions in an MDP can be prescribed by a policy $\pi : S \times A \rightarrow [0, 1]$, and $\tau_\pi = (s_0, a_0, s_1, ...)$ is a trajectory of states $s_i \in S$ and actions $a_i \in A$ experienced over time by executing π. Discounted return is the discounted sum of reward over a trajectory, $G(\tau) = \sum_{t=0}^{n} \gamma^t r(s_t, a_t, s_{t+1})$. In this work, we learn π by applying one of four algorithms: Q-learning (Watkins and Dayan 1992), PPO (Schulman et al. 2017), DDQN (Mnih et al. 2015), or A2C (Mnih et al. 2016). Unless otherwise noted, we use the hyperparameters in Apdx. D. The shorthand $\pi_{r,D}$ denotes the policy learned with reward function r and some solver D (i.e., an algorithm and hyperparameters).

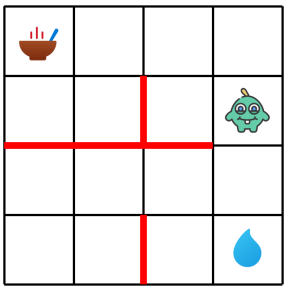
Reward Function Overfitting Let $M : \tau \rightarrow \mathbb{R}$ be the true task performance metric. For example, this metric might encode whether the agent reached a goal state or not. Let a **learning context** be a tuple of an RL algorithm, hyperparameter values, and an MDP$\backslash r$; given a reward function, a learning context can be used to train a policy. We claim a reward function r_1 is overfit with respect to one or more learning contexts, $D_1 \sim \mathcal{D}$, if there exists an alternative reward function r_2 such that the task performance metric is optimized over D_1 but not over the larger distribution, $\mathcal{D}$:

$$\begin{aligned} \mathop{\mathbb{E}}_{\tau \sim \pi_{r_1, D_1}} [M(\tau)] &> \mathop{\mathbb{E}}_{\tau \sim \pi_{r_2, D_1}} [M(\tau)] \\ \mathop{\mathbb{E}}_{\tau \sim \pi_{r_1, \mathcal{D}}} [M(\tau)] &< \mathop{\mathbb{E}}_{\tau \sim \pi_{r_2, \mathcal{D}}} [M(\tau)] \end{aligned} \tag{1}$$

where D_1 is a set of one or more learning contexts, $D_1 = \{d_1, d_2, \ldots, d_n\}$. This definition is adapted from supervised learning overfitting: the hypothesis space corresponds to the space of possible reward functions and the training and test sets correspond to potential RL algorithms, hyperparameters, and environments (Mitchell and Mitchell 1997).

Optimal Reward Functions Optimal reward functions are related to overfitting: these reward functions are the best performing in a given a learning context. A reward function $r^*_{\mathcal{D}}$ is optimal under some distribution $\mathcal{D}$ of learning contexts if it maximizes the expected value of learned policies, i.e., $r^*_{\mathcal{D}} = argmax_r \, \mathbb{E}_{\tau \sim \pi_{r,\mathcal{D}}}[M(\tau)]$.

Hungry Thirsty Domain We use a modified Hungry Thirsty domain (Singh, Lewis, and Barto 2009) as a testbed. This gridworld domain has a fixed time horizon of 200 steps. Food is located in one randomly-selected corner; water in another. Some transitions are blocked by walls (Fig. 1). At each timestep, the agent can choose one of six actions: move in a cardinal direction, eat, or drink. The agent's goal is to have sated hunger for as many timesteps as possible. The agent is hungry if and only if it did not eat in the last timestep. However, the agent can only successfully eat if it is co-located with the food and has quenched thirst. On each timestep, the agent stochastically becomes thirsty with 0.1 probability, and only becomes not thirsty if it drinks while co-located with the water. The agent's state is described by its location, as well as two boolean predicates: $\texttt{H}$ and $\texttt{T}$, corresponding to hunger and thirst. Time remaining is omitted.



"I am hungry and not thirsty."

Figure 1: Hungry Thirsty (4×4 grid). Food and water are each located in a corner. Red walls are impassable. The current reward-relevant state is abbreviated as $\texttt{H} \wedge \neg\texttt{T}$, which corresponds to the agent being hungry and not thirsty. The 6×6 grid variant is depicted in Singh, Lewis, and Barto (2009).

For this task, the performance metric is simply the number of timesteps the agent has sated hunger: $M(\tau) = \sum_{t=1}^{200} \mathbb{1}(\neg\texttt{H} \in s_t)$. This specific metric can be formulated as a sparse Markovian reward function, $r(s, a, s') = \mathbb{1}(\neg\texttt{H} \in s)$.[2] Under the optimal policy for this reward function, the agent alternates between navigating to the water or drinking when thirsty, and navigating to the food or eating when not thirsty. While it is often possible for RL algorithms to learn with this sparse reward function, shaped reward functions that reward the not thirsty ($\neg\texttt{T}$) subgoal or punish time spent hungry ($\texttt{H}$) let many RL algorithms solve this domain faster and more easily. These properties make this domain an interesting testbed for studying reward design.

For our experiments, all reward functions take the form:

$$\begin{aligned} r(\texttt{H} \wedge \texttt{T}) &= a & r(\texttt{H} \wedge \neg\texttt{T}) &= b \\ r(\neg\texttt{H} \wedge \texttt{T}) &= c & r(\neg\texttt{H} \wedge \neg\texttt{T}) &= d \end{aligned}$$

where $a, b, c, d \in \mathbb{R}$. Since there are no reward components for location, opportunities for shaping—and, thereby, overfitting—are limited but still possible. For shorthand, we write reward functions as $[a, b, c, d]$. We say reward functions encode the task when the optimal policy matches the optimal policy derived from the sparse reward function. Singh, Lewis, and Barto (2009) found the highest-performing reward function to be $[-0.05, -0.01, 1.0, 0.5]$ for a continuing version of this domain; this reward function is dense and notably rewards drinking water as a subgoal.

When conducting large experiments, we assign each of a, b, c, and d to a value from the set: $\{\pm 1, \pm 0.5, \pm 0.1, \pm 0.05, 0\}$. We chose these values based on the reward function from Singh, Lewis, and Barto (2009). Reward functions that meet the following criteria

[2] Such reformulation as a Markovian reward function is not universally possible across all task performance metrics.

trivially do not encode the task and are thus excluded: $r(\texttt{H} \wedge \neg\texttt{T}) \geq r(\neg\texttt{H} \wedge \texttt{T})$ and $r(\texttt{H} \wedge \neg\texttt{T}) \geq r(\neg\texttt{H} \wedge \neg\texttt{T})$. Such reward functions encode an incorrect optimal policy of navigating to the water and consistently drinking. This filtering leaves 5,196 reward functions.

4 Computational Experiments

We first assess overfitting in reward functions by conducting large-scale performance comparisons, in which we measure a learning context's ability to optimize the task performance metric given a reward function. As an intuitive example, we speculate that when using an RL algorithm with a high learning rate, a high magnitude reward function might be less likely to lead to convergence within a fixed training duration than a lower magnitude reward function.

To study this relationship between reward function design and hyperparameters empirically, we assess the mean task performance metric accumulated over all 200-timestep episodes of training achieved by learning with different reward functions across varied Q-learning hyperparameters: γ (the environment discount factor) and α (the learning rate). While γ is formally defined as a parameter of the environment, and not the learning algorithm, it is typically selected to construct a viable horizon for applying an RL algorithm (Jiang et al. 2015) and can thus be equally considered a hyperparameter of the learning algorithm. We additionally study whether the reward functions are overfit to the learning algorithm itself by comparing performance metrics with several deep RL methods: A2C (Mnih et al. 2016), DDQN (Mnih et al. 2015), and PPO (Schulman et al. 2017) in the 6×6 Hungry Thirsty domain. Unless otherwise specified, we train 10 agents per experimental setting.

4.1 Hypotheses

Below we list hypotheses concerning the manifestation of reward function overfitting. We substantiate how we test these hypotheses in practice in Sections 4.2 and 4.3.

H1: Reward functions that are effective in one learning context can be ineffective in another. There exist two different learning contexts (D_1 and D_2) and a reward function r_1 such that r_1 achieves high cumulative performance (as measured by the true performance metric) when tested with D_1 but low cumulative performance with D_2. In formal terms, there exists a reward function r_1 such that

$$\mathop{\mathbb{E}}_{\tau \sim \pi_{r_1, D_1}}[M(\tau)] > \beta_1 \text{ and } \mathop{\mathbb{E}}_{\tau \sim \pi_{r_1, D_2}}[M(\tau)] < \beta_2,$$

where β_1 is some high threshold (e.g., performing among the top 25% of reward functions when tested with D_1) and β_2 is some low threshold (e.g., performing among the bottom 25% of reward functions when tested with D_1).

H1 tests whether some reward functions enable successful learning in one learning context but not in another. In other words, this hypothesis assesses whether reward function overfitting can occur. Although the learning context D_2 could be chosen adversarially— i.e., to include an RL algorithm incapable of learning—we assume the algorithm (with its hyperparameters) in D_2 aims to maximize expected return and is generally capable of doing so.

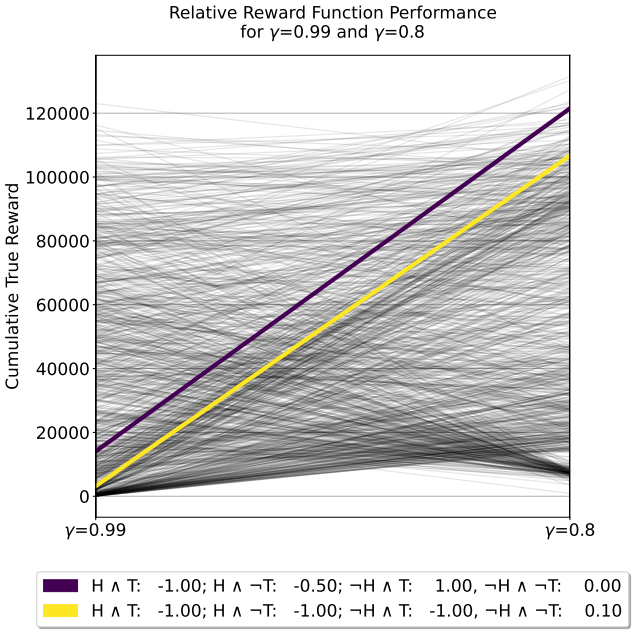


Figure 2: A parallel coordinate plot showing the paired rankings of reward functions. Each line corresponds to a reward function, with cumulative performance averaged over 10 independently-trained agents. The many intersections portray the uncorrelated nature of these rankings. The two reward functions with the largest cumulative difference in performance are highlighted. These reward functions result in low cumulative performance when $\gamma = 0.99$, but high performance when $\gamma = 0.8$. See Apdx. C for more examples.

H2: Reward functions that are optimal in one learning context can be suboptimal in another. Given a reward function $r^*_{D_1}$ that is optimal under a learning context D_1, a different reward function $r^*_{D_2}$ may be optimal with respect to another learning context, D_2. In formal terms, there exist two learning contexts D_1 and D_2 such that $r^*_{D_1} \neq r^*_{D_2}$.

H2 tests tests whether the reward functions which are found to be best-performing are consistently best-performing across multiple learning contexts and whether the best-performing reward functions can be overfit. As in **H1**, we assume that the considered algorithms aim to maximize expected return and are generally capable of doing so.

H3: The performances of different reward functions are uncorrelated across learning contexts. If reward functions are ranked by their average cumulative performance metric scores (e.g., $r_a > r_b > r_c > \ldots$ for a learning context D_1), the ranked reward functions from D_1 will be uncorrelated with the ranked reward functions from D_2. In formal terms, for some set of reward functions $r_1, r_2, \ldots, r_n$, $\mathbb{E}_{\tau \sim \pi_{r_i, D_1}}[M(\tau)]$ will be uncorrelated with $\mathbb{E}_{\tau \sim \pi_{r_i, D_2}}[M(\tau)]$ for $1 \leq i \leq n$.

H3 examines the commonality of reward function overfitting. If this phenomenon is rare, correlation across learning contexts should be high. If it is common, correlation should be low. Of these hypotheses, confirmation of **H3** is most concerning as it indicates extensive reward function overfitting.

Table 1: A comparison of reward function performance assessed over 1000 trials (Q-learning) or 30 trials (deep RL methods). Performance is assessed with the Hoeffding Bound, which is akin to a confidence interval, and a Mann Whitney U-test. This data confirms that the same reward function can lead to very different performance with different hyperparameters or algorithms.

Reward Function	Experiment	Hoeffding Bound	p-value
[-1.0, -1.0, -1.0, 0.1]	$\gamma = 0.99$ $\gamma = 0.8$	[4,965; 20,234] [86,653; 101,922]	< 0.01
[-0.1, 0.2, 0.5, 1.0]	DDQN A2C	[94,790; 182,944] [-29,040; 59,114]	< 0.01

Table 2: Kendall's τ_b correlation over the 5196 tested reward functions for hyperparameter experiments and 107 tested reward functions for algorithm experiments. $\tau_b \in [-1, 1]$. $|\tau_b| < 0.1$ indicates the variables are uncorrelated; $|\tau_b| < 0.2$ indicates a weak correlation. In our experiments, **H3** is supported with low τ_b values, even with high p-values. Almost all comparisons are either uncorrelated or weakly correlated; **H3** is supported for all experiments except PPO vs. A2C. This data confirms that the choice of reward function is highly sensitive for RL algorithm performance.

# of Reward Fns	$\mathcal{D}_1$	$\mathcal{D}_2$	τ_b	p-value
5196	$\gamma = 0.99$	$\gamma = 0.8$	0.07	< 0.01
	$\gamma = 0.99$	$\gamma = 0.5$	0.04	0.07
	$\gamma = 0.8$	$\gamma = 0.5$	0.12	< 0.01
	$\alpha = 0.25$	$\alpha = 0.05$	0.11	< 0.01
107	PPO	A2C	0.25	0.01
	PPO	DDQN	-0.04	0.62
	PPO	QLearn	0.13	0.08
	A2C	QLearn	-0.08	0.29
	A2C	DDQN	-0.01	0.87
	DDQN	QLearn	-0.06	0.41

4.2 Overfitting to Hyperparameters

We first assess whether reward functions can be overfit to either the discount factor, γ, or the learning rate, α. For this experiment, we use a Q-learning agent trained over 2000 episodes. For evaluating overfitting to the discount factor, we vary γ for each learning context: $\gamma = 0.99$, $\gamma = 0.8$, and $\gamma = 0.5$. For evaluating overfitting to the learning rate, we consider $\alpha = 0.05$ and $\alpha = 0.25$. The standard hyperparameters are described in Apdx. D. We average performance, as measured by the cumulative true reward, over 10 trials to account for stochasticity stemming from the environment or from the learning process (i.e., randomized weights).

H1: Reward functions that are effective in one learning context can be ineffective in another. For all experiments, we find some reward functions which result in policies which achieve high task performance scores when trained with one learning context but low task performance scores when trained with a different learning context. Some such reward functions are highlighted in Fig. 2 and Apdx. Fig. 3. To assess whether these differences are not just a consequence of stochastic policy learning, we re-ran these experiments with the reward functions which resulted in maximally different true performance for 1000 additional trials. We then computed the 90% Hoeffding Bound (Hoeffding 1994), which bounds the average cumulative task performance metric across trials with 90% probability, and we separately performed a Mann Whitney U-test (Nachar et al. 2008) to assess whether the mean cumulative true task performance values were drawn from the same underlying distribution. We find, in all cases, we can reject the null hypothesis that these underlying distributions are the same as the observed differences are all statistically significant ($p < 0.05$). We conclude that, across varied hyperparameters, reward functions that are effective in one learning context can be ineffective in another. See Tab. 1 and Apdx. Tab. 3.

H2: Reward functions that are optimal in one learning context can be suboptimal in another. Across each pair of tested learning contexts, we find that the best-performing reward function differs (Apdx, Fig. 4). To test if this difference is just a consequence of stochastic policy learning, we conduct follow-up experiments. Specifically, we fix an experimental learning context and assess whether the best-performing reward function for that learning context outperforms the top-3 reward functions from a different experimental learning context, testing the cumulative task performance metric for each reward function over 1000 trials. For example, the best-performing reward function function for $\gamma = 0.99$ was $[-0.05, -0.05, 0.5, 0.5]$, which outperformed the best-performing reward function for $\gamma = 0.5$, $[-1.0, -1.0, 0.0, 1.0]$. We then compute the 90% Hoeffding Bound for the mean cumulative task performance metric, and we separately conduct a Mann Whitney U-test to assess whether the distribution of the best-performing reward function's performance is greater than that of the alternative tested reward function (which is best-performing for a different experimental condition). We find that we can reject the null hypothesis that these reward functions result in equal performance distributions in 16 of 18 experiments ($p < 0.05$). In general, the best-performing reward function for one learning context outperforms the top-3 reward functions for another learning context. See Apdx. Tab. 4.

H3: The performances of different reward functions are uncorrelated across learning contexts. We compute Kendall's tau rank correlation to assess **H3**; Kendall's tau measures the strength and direction of the monotonic association between rankings, without taking the difference in magnitude of the performance metric into consideration, since some learning contexts may be consistently 'better' or 'worse' in terms of raw performance. We used a nonparametric test, since the cumulative scores were not found to be normally distributed over many trials. In this setting, the null hypothesis is that two random variables are independent (i.e., $\tau_b = 0$). **H3** is supported with low τ_b values, even with high p values. Generally, the closer τ_b is to 0, the more samples are needed to show significance. We find that reward function performance is **uncorrelated** ($|\tau_b| < 0.1$, see Table 2) or **weakly correlated** ($|\tau_b| < 0.2$, see Table 2) in all discount factor and learning rate experiments. We conclude that performance across varying hyperparameters is sensitive to reward function choice.

From these experiments, we find consistent evidence that reward functions can be overfit to RL hyperparameters.

4.3 Overfitting to RL Algorithms

For the Section 4.2 Q-learning experiments with varied hyperparameters, we trained 5196 agents, 10 times each. The protocol for generating these reward functions is described in Section 3. In the deep RL setting, this scale of training is infeasible because training each agent takes between 3 and 11 minutes (Apdx. F). To test overfitting in this setting, we instead consider a restricted set of reward functions to reduce the computational burden. We source these reward functions from the user study; specifically, we consider the set of unique reward functions that experts handcrafted at any point during their sessions and that also encode the desired optimal policy in easy environment configurations (Section 5). In total, we analyze 107 reward functions in this deep RL setting. For each reward function, we train A2C, DDQN, PPO, and Q-learning agents. We train each agent over 5000 episodes, and average performance over 10 trials.

H1: Reward functions that are effective in one learning context can be ineffective in another. We again find that every experiment variation uncovers reward functions which enable successful learning in one experimental learning context, but not the other. For example, the reward function $[-0.1, 0.2, 0.5, 1]$ achieved a high mean cumulative performance metric score of 232055 for DDQN, but a low mean cumulative score of 107—indicative of never learning the optimal policy—for A2C. We then ran this specific test an additional 30 times, and found evidence that we can again reject the null hypothesis that these true task performances were drawn from the same underlying distribution ($p < 0.05$). See Table 1 and Appendix Fig. 5, in which the reward functions that achieve maximally different performance metric measures are highlighted.

H2: Reward functions that are optimal in one learning context can be suboptimal in another. In 5 of 6 experiments varying the RL algorithm, the optimal reward functions differ. The only case in which this is not true is in the PPO and A2C comparisons. In this case, the true performance metric function itself ($[0, 0, 1, 1]$) is the optimal reward function for both algorithms. See Appendix Fig. 5.

H3: The performances of different reward functions are uncorrelated across learning contexts. Again using Kendall's τ_b for assessment, we mostly find evidence that reward functions' cumulative true performance metric scores are **uncorrelated** when varying the RL algorithm ($|\tau_b| < 0.1$, see Table 2). Specifically, we find the PPO vs. DDQN, A2C vs. Q-learning, A2C vs. DDQN, and DDQN vs. Q-learning agents to be mutually uncorrelated. We find evidence of **weak correlation** between PPO and Q-learning ($|\tau_b| < 0.2$). Lastly, we find evidence of **some correlation** ($\tau_b = 0.25$) for the PPO vs. A2C comparison. Statistical significance—which allows us to reject the null hypothesis that the two random variables are independent—is generally not established due to the reduced sample size.

From these experiments, we find consistent evidence that reward functions can be overfit to RL algorithms.

5 Expert Human Subject Experiments

To assess how experts design reward functions and whether this problem of reward function overfitting caries over to realistic settings, we conduct a controlled observation study.

Study Population We conducted 2 pilot studies, followed by 30 studies with expert participants drawn from four US research universities (R1). To qualify as an expert, participants were required to meet one or more of the following criteria: (1) have experience conducting research on RL methods; (2) have used RL methods in research; or (3) have passed a class which covered reinforcement learning in depth. Of the 30 participants, 1 was a post-doctoral scholar, 17 were PhD students, 5 were research-based master's students, and 7 were advanced undergraduates. Participants are each assigned a study ID, ranging from **P0** to **P29**.

Study Protocol The study session took one hour and was primarily conducted in-person (19 of 30 sessions). Participants were compensated $40 USD. The study used a Jupyter notebook, in which participants were required to select a reward function, algorithm, and hyperparameters to train an agent to solve the Hungry Thirsty task (Apdx. B). Participants were asked to speak aloud as they worked, and the experimenter took detailed notes. The experimenter occasionally asked open-ended questions (such as "what are you trying to do now?") to prompt the participant to continue speaking. Five minutes before the end of the session, participants were asked to submit their best configuration consisting of some reward function r_i and some algorithm and hyperparameter selection, D_i. Afterwards, participants were asked to answer five structured questions (Apdx. B.3).

The participants were informed that the research team would independently train an agent using their submitted reward function, algorithm, and hyperparameters, and that if this trained agent performed in the top ten across all participants' agents in terms of cumulative performance, they would receive a $10 USD bonus. Participants were required to train at least three different agents—though the experimenter explicitly noted that they could simply re-train the same agent three times to meet this requirement.

The first 12 participants used a 6×6 grid for the Hungry Thirsty domain. After observing participants struggling to solve this task, we reduced the size of the grid to 4×4 for the remaining 18 participants. The study was IRB approved.

Experts Often Design Invalid Reward Functions The Hungry Thirsty domain has harder and easier environment configurations. In total, there are 12 different configurations, which correspond to different placements of the food and water. In a 6×6 grid, the food and water can either be located 5 steps apart in the best case or 16 steps apart in the worst case. In a 4×4 grid, these distances are 3 and 9 steps in the best and worst cases, respectively. As in the original version of Hungry Thirsty (Singh, Lewis, and Barto 2009), the locations of the food and water are randomly resampled each time the user trains a new agent, but remain consistent throughout the lifetime of the agent. In this study, the user is tasked with designing a reward function which is invariant to any choice of environment configuration.

To determine whether a reward function is valid for a given task configuration (i.e., for fixed food and water positions), we empirically assess whether a policy—learned with value iteration—is the same as the optimal policy under the sparse reward function. Specifically, we use value iteration (with $\theta = 0.01$ as the end criteria) to solve for an approximately optimal policy using the sparse reward function and $\gamma = 0.99$. We then use value iteration to solve for a policy using the user's submitted reward function and choice of γ. We run 100 test episodes for each agent with a fixed random seed, and use the average cumulative undiscounted task performance metric for comparison. If the policy learned with the user's reward function has the same cumulative undiscounted task performance as the policy learned with the sparse reward function, we consider it valid. If the user's reward function is valid for all environment configurations, we say it encodes the task.

The majority of participants (83%) successfully selected reward functions which were valid with the easier placements of the food and water on adjacent corners (10 of 12 in the 6×6 setting; 15 of 18 in the 4×4 setting). However, only 47% of participants selected reward functions which were valid when the food and water are maximally distant, at opposite corners (4 of 12 in the 6×6 setting; 10 of 18 in the 4×4 setting). For example, **P23**'s reward function $[-0.05, 0.5, 0.5, 1.0]$ is valid in the easier adjacent case but not the opposite-corners case, because when the food and water are maximally distant the optimal policy causes the agent to remain in the state $\texttt{H} \wedge \neg \texttt{T}$. Finding this form of misdesign, where reward functions are only valid in some environment configurations, adds support to the research pursuit of inverse reward design methods (Hadfield-Menell et al. 2017), which is built upon the perspective that reward functions should be considered an observation about the expert's intended reward function and not as a perfect specification.

Experts Overfit Reward Functions to Algorithms Even when experts wrote reward functions which encode the task, they typically continued to edit their reward functions. Each expert tried a sequence of reward functions $r_1, r_2, \ldots, r_n$ and finally settled on some reward function r_i where $i \in [1, n]$. The user evaluated each of these reward functions alongside potentially-changing algorithms and hyperparameters, $D_1, D_2, \ldots, D_n$ and settled on some choice D_i. Because every aspect of the user's solution may be changing simultaneously, this setting is messier and harder to evaluate than the purely-computational setting. To evaluate overfitting, we test all of the user's reward functions with standard implementations for DDQN, PPO, and A2C and fixed hyperparameters (Apdx. D). We discard the user's algorithms (i.e., D_i) and exclusively test the user's reward functions.

In this setting, we define overfitting to have occurred if one or more of the user's tested reward functions (r_j, where $j \in [1, n]$ and $r_j \neq r_i$) significantly outperforms their final selection with respect to one or more of the three tested RL algorithms. We define this performance difference threshold to be 20000, accumulated over 5000 training episodes and averaged over 10 trials. This overfitting assessment is different from the computational setting, which requires comparing the rankings and not absolute performance between different reward functions. Since each user tried only a small handful of reward functions (on average, 4.1 unique reward functions), these rankings are less meaningful.

Of the users who tried multiple reward functions and submitted a best-case-valid reward function, 68% (15 of 22) overfit their reward functions. For example, participant **P20** tried the reward function $[-0.1, 0.1, -0.1, 1]$, which achieved a mean cumulative performance of 138,092 using DDQN. In their final selection, this user instead chose the reward function $[-5, 15, 5, 100]$, which achieved a mean cumulative performance of 1,031 using DDQN. We include an alternative metric for overfitting in the user study in Apdx. E.

Assessing the Design Process with Thematic Analysis To analyze not just experts' reward design outcomes, but also their design process, we applied qualitative analysis in the form of *thematic analysis* (Braun and Clarke 2006; Hopkins and Booth 2021). Thematic analysis is a system for extracting patterns from qualitative data by systematically coding and analyzing transcripts. To perform thematic analysis, every statement of each transcript is first assigned a summary (also known as a code). Each of these summaries is then further distilled into a detailed, low-level theme. These low-level themes are finally distilled into high-level themes. Thematic analysis is generally considered successful if the resulting themes are consistent and coherent, and describe the data they incorporate well. In such cases, the extracted themes provide insight into the unstructured data. In our application of thematic analysis, the first summary step of this process generated 990 codes. The second step generated 212 low-level themes. And, finally, the third step resulted in the extraction of 10 high-level themes. We include the full analysis in the supplementary material. Here, we discuss these themes and their implications for reward design.

Experts' Approaches to RL and Reward Design Thematic analysis showed that experts use one or more of the following strategies when tasked with crafting and solving an RL task: folklore-based, intuition-based, trial-and-error-based, hypothesis-based, random-based, or reason-based. For example, **P25** declared, "I've heard that reward scaling is pretty important", and this quote is an example of using a folklore-based process. Concerning this same parameter choice, **P27** declared, "The reward scaling factor must be very large, I think, since you might only see little food," and this quote is an example of using a reason-based process. Experts often switched between two or more strategies.

Trial-and-Error Reward Design is Typical This user study was designed to be naturalistic; participants could choose to focus primarily on any combination of the three axes of choice, between specifying the reward function, algorithm, and hyperparameters. Conventional reward design wisdom suggests that experts should try to align a reward function as closely as possible with the task completion criteria, and should only adjust the reward function if it is found to not encode correct measurements of task outcomes.

93% of experts tried at least two reward functions (only **P2** and **P4** stuck to a single reward function). Experts tried

4.1 unique reward functions on average. In this study setting, shaping was unnecessary: any of the available algorithms could learn from the sparse reward function. Despite this, and almost all users (97%) shaped their reward functions. This finding is compelling: even in the absence of a need to shape, experts gravitate towards doing so.

Analyzing study transcripts, we find that half of experts (**P5-13, P15-17, P20, P23, P27-28**) explicitly noted a perceived error at least once before modifying their reward function (thus employing a reason-based process). For example, **P5** stated: "I realized I'm penalizing the $\texttt{H} \wedge \texttt{T}$ state too much, because the agent knows it will be penalized on the way back [to the water]." In contrast, some experts indicated they were relying on trial and error: **P28** stated, "The worst possible state to be in is $\texttt{H} \wedge \texttt{T}$, so I'm going to assign -1. The best possible state to be in is $\neg\texttt{H} \wedge \neg\texttt{T}$, so I'm assigning that to 1. $\texttt{H} \wedge \neg\texttt{T}$ is not particularly as bad as $\texttt{H} \wedge \texttt{T}$... Setting that to -0.25. Reward for $\neg\texttt{H} \wedge \texttt{T}$: not too close to -1; I'll just assign some arbitrary small value."

A Common Misdesign Cause: Weighing State Goodness Weighing state goodness to design a reward function was a recurring low-level theme. Most experts (83%) stated something to the effect of: "It's best to be $\neg\texttt{H} \wedge \neg\texttt{T}$, so I'll set that to the max, 1. Being $\neg\texttt{T}$ is better than being $\neg\texttt{H}$. Worst is at $\texttt{H} \wedge \texttt{T}$; setting that to -1" (**P25**; this statement corresponded to their invalid reward function [-1.0, 0.3, -0.35, 1.0], for which the optimal policy is to remain drinking water indefinitely). This reward design practice—of using the reward function to rank the goodness of immediate states and/or actions, applying a myopic design strategy without assessing how the reward function will be used as an optimization target for computing expected discounted return—often led to reward misdesign, as it did for **P25**.

Though less often, some experts did recognize the importance of reward accumulation and state visitation frequency (another recurring low-level theme). For example, **P23** stated "A positive reward for $\texttt{H} \wedge \neg\texttt{T}$ is not the way to go. A combination with a negative reward for $\texttt{H} \wedge \texttt{T}$ makes it worse, since it would rather accumulate positive rewards at the water instead of searching for food." This design process—of considering summed reward, which aligns with the RL optimization objective—was relatively rare (i.e., 30% of experts noted something to this effect). From this qualitative analysis, we found this lack of emphasis on reward accumulation and expected discounted return to be the main cause of explicit reward misdesign, wherein reward functions were invalid and did not correctly encode the task.

This observation—that humans assume a myopic interpretation of reward functions, in which reward accumulation is largely ignored—has previously been observed in another setting. Knox and Stone (2015) discovered that when learning a reward function from non-expert human feedback, humans adopt a similarly myopic teaching strategy. Finding that this myopic interpretation is echoed across both non-expert and expert users can inform future efforts to support humans in designing reward functions, and can help reinterpret how humans' reward functions should be used for optimization.

6 Limitations

While we studied the Hungry Thirsty domain in depth in this work, we only evaluated reward design practice in this one domain. Hungry Thirsty is a rich testbed for assessing reward design practice, but understanding this practice across multiple domains with diverse properties equally deserves attention. This domain in particular allows us to assume the existence and specification of a true task performance metric, but in many circumstances, specifying such a metric is itself a challenging problem. In such cases, the methodology we use to study reward design would not readily reapply.

Another limitation of this work concerns the definition of reward function overfitting. Our definition omits a temporal aspect to the distribution ($\mathcal{D}$, consisting of algorithms, hyperparameters, and tasks) that makes samples from $\mathcal{D}$ dependent (i.e., not i.i.d.). For example, if an expert has tested a reward function r with one RL algorithm and a set of hyperparameter values, we suspect such an expert is more likely to next test the reward function r with the *same* algorithm and different hyperparameter values than with a different RL algorithm. This temporal component is omitted from our overfitting definition—as it similarly tends to be in the supervised learning setting—and future work could explore the consequences of this omission.

7 Discussion

Despite the prevalence of trial-and-error reward design, the implications of this widespread practice remain underexplored. In this first analysis of the consequences of this practice, we identify two problems: reward function overfitting and the frequent design of invalid task specifications. In overfitting, reward functions are designed with respect to a fixed algorithm or hyperparameter set, and the resulting reward functions bias toward better learning given these design choices. This finding contributes to concerns around reproducibility in RL: we find the performance of the reward function is often dependent on the choice of algorithm. For RL practitioners, one takeaway from this work is that the reward function—like the discount factor (Jiang et al. 2015)—should be defined twice: once to specify the true problem as part of the MDP, and once as a form of hyperparameter for the RL algorithm to facilitate learning. This separation accommodates the need to design a reward function for successful learning while also supporting fair evaluations.

In addition to overfitting, we find that ad hoc trial-and-error reward design leads to misdesign in the form of invalid task specifications, wherein experts design reward functions which fail to encode the desired task, even in a simple grid-world domain. One candidate cause for this misdesign is that experts typically adopt a myopic interpretation of reward, and this interpretation is at odds with the RL objective of optimizing cumulative, if discounted, rewards. Given this finding, one future direction would assess the systemic errors humans make when designing reward functions, and try to construct better mechanisms for inferring the humans' true intent given these systemic errors. Such a mechanism could build off of inverse reward design (Hadfield-Menell et al. 2017).

While there is great optimism around the flexibility of reward as the optimization target for learning (Silver et al. 2021), this paper contributes to mounting evidence that most people are ineffective reward designers in *current* practice (Amodei et al. 2016; Krakovna et al. 2020; Knox et al. 2021). As future work, we assert that the community should also explore mechanisms to support humans—including experts!—in this reward design endeavor. Specifically, one could develop guidance for the human reward designer's process such that it more directly reflects the RL optimization target of expected discounted return. Additionally, it is worth exploring whether incorporating explanation mechanisms can improve reward design outcomes (for example, by assisting experts in assessing the contributions of decomposed reward components (Juozapaitis et al. 2019)).

Alternative models of reward should also be considered and evaluated both for their propensity for overfitting and for their propensity for other forms of misdesign. Reward machines (Icarte et al. 2018) and hybrid reward architectures (Van Seijen et al. 2017) are two such candidates. In reward machines, reward functions are described as a type of finite state machine instead of directly as a function. While reward machines may elicit feature engineering and are thus taboo in RL, humans may be better able to design reward functions which correctly encode a task if they use sufficient structure for guiding the design process. In hybrid reward architectures, the reward function is decomposed into n different reward functions, each of which is then used to optimize a policy. These policies are subsequently aggregated into a single policy. These methods both induce supporting structures for designing reward functions, and this support may help humans write better reward functions with lowered propensity for overfitting or other misdesign.

8 Acknowledgments

We thank Dylan Hadfield-Menell and many RLDM conference-goers for their input on this work. In addition, the authors would like to thank the user study participants and anonymous reviewers at both AAAI and RLDM.

References

Amodei, D.; Olah, C.; Steinhardt, J.; Christiano, P.; Schulman, J.; and Mané, D. 2016. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565*.

Andrychowicz, O. M.; Baker, B.; Chociej, M.; Jozefowicz, R.; McGrew, B.; Pachocki, J.; Petron, A.; Plappert, M.; Powell, G.; Ray, A.; et al. 2020. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1): 3–20.

Braun, V.; and Clarke, V. 2006. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2): 77–101.

Chiang, H.-T. L.; Faust, A.; Fiser, M.; and Francis, A. 2019. Learning navigation behaviors end-to-end with autorl. *IEEE Robotics and Automation Letters*, 4(2): 2007–2014.

Christiano, P. F.; Leike, J.; Brown, T.; Martic, M.; Legg, S.; and Amodei, D. 2017. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30.

Engstrom, L.; Ilyas, A.; Santurkar, S.; Tsipras, D.; Janoos, F.; Rudolph, L.; and Madry, A. 2019. Implementation matters in deep rl: A case study on ppo and trpo. In *International conference on learning representations*.

Faust, A.; Francis, A.; and Mehta, D. 2019. Evolving rewards to automate reinforcement learning. *arXiv preprint arXiv:1905.07628*.

Hadfield-Menell, D.; Milli, S.; Abbeel, P.; Russell, S. J.; and Dragan, A. 2017. Inverse reward design. *Advances in neural information processing systems*, 30.

He, J. Z.-Y.; and Dragan, A. D. 2021. Assisted robust reward design. *arXiv preprint arXiv:2111.09884*.

Henderson, P.; Islam, R.; Bachman, P.; Pineau, J.; Precup, D.; and Meger, D. 2018. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32.

Hoeffding, W. 1994. Probability inequalities for sums of bounded random variables. In *The collected works of Wassily Hoeffding*, 409–426. Springer.

Hopkins, A.; and Booth, S. 2021. Machine learning practices outside big tech: How resource constraints challenge responsible development. In *Proceedings of the 2021 AAAI/ACM Conference on AI, Ethics, and Society*, 134–145.

Ibarz, J.; Tan, J.; Finn, C.; Kalakrishnan, M.; Pastor, P.; and Levine, S. 2021. How to train your robot with deep reinforcement learning: lessons we have learned. *The International Journal of Robotics Research*, 40(4-5): 698–721.

Icarte, R. T.; Klassen, T.; Valenzano, R.; and McIlraith, S. 2018. Using reward machines for high-level task specification and decomposition in reinforcement learning. In *International Conference on Machine Learning*, 2107–2116. PMLR.

Jiang, N.; Kulesza, A.; Singh, S.; and Lewis, R. 2015. The dependence of effective planning horizon on model accuracy. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, 1181–1189. Citeseer.

Juozapaitis, Z.; Koul, A.; Fern, A.; Erwig, M.; and Doshi-Velez, F. 2019. Explainable reinforcement learning via reward decomposition. In *IJCAI/ECAI Workshop on explainable artificial intelligence*.

Knox, W. B.; Allievi, A.; Banzhaf, H.; Schmitt, F.; and Stone, P. 2021. Reward (mis) design for autonomous driving. *arXiv preprint arXiv:2104.13906*.

Knox, W. B.; Hatgis-Kessell, S.; Booth, S.; Niekum, S.; Stone, P.; and Allievi, A. 2022. Models of human preference for learning reward functions. *arXiv preprint arXiv:2206.02231*.

Knox, W. B.; and Stone, P. 2009. Interactively shaping agents via human reinforcement: The TAMER framework. In *Proceedings of the fifth international conference on Knowledge capture*, 9–16.

Knox, W. B.; and Stone, P. 2015. Framing reinforcement learning from human reward: Reward positivity, temporal

discounting, episodicity, and performance. *Artificial Intelligence*, 225: 24–50.

Krakovna, V.; Uesato, J.; Mikulik, V.; Rahtz, M.; Everitt, T.; Kumar, R.; Kenton, Z.; Leike, J.; and Legg, S. 2020. Specification gaming: the flip side of AI ingenuity. *DeepMind Blog*.

MacGlashan, J.; Ho, M. K.; Loftin, R.; Peng, B.; Wang, G.; Roberts, D. L.; Taylor, M. E.; and Littman, M. L. 2017. Interactive learning from policy-dependent human feedback. In *International Conference on Machine Learning*, 2285–2294. PMLR.

Mitchell, T. M.; and Mitchell, T. M. 1997. *Machine learning*, volume 1. McGraw-hill New York.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, 1928–1937. PMLR.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *nature*, 518(7540): 529–533.

Nachar, N.; et al. 2008. The Mann-Whitney U: A test for assessing whether two independent samples come from the same distribution. *Tutorials in quantitative Methods for Psychology*, 4(1): 13–20.

Ng, A. Y.; Harada, D.; and Russell, S. 1999. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, 278–287.

Ng, A. Y.; Russell, S.; et al. 2000. Algorithms for inverse reinforcement learning. In *Icml*, volume 1, 2.

Niekum, S.; Barto, A. G.; and Spector, L. 2010. Genetic programming for reward function search. *IEEE Transactions on Autonomous Mental Development*, 2(2): 83–90.

Parker-Holder, J.; Rajan, R.; Song, X.; Biedenkapp, A.; Miao, Y.; Eimer, T.; Zhang, B.; Nguyen, V.; Calandra, R.; Faust, A.; et al. 2022. Automated reinforcement learning (autorl): A survey and open problems. *Journal of Artificial Intelligence Research*, 74: 517–568.

Ratner, E.; Hadfield-Menell, D.; and Dragan, A. D. 2018. Simplifying reward design through divide-and-conquer. *arXiv preprint arXiv:1806.02501*.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Silver, D.; Singh, S.; Precup, D.; and Sutton, R. S. 2021. Reward is enough. *Artificial Intelligence*, 299: 103535.

Singh, S.; Lewis, R. L.; and Barto, A. G. 2009. Where do rewards come from. In *Proceedings of the annual conference of the cognitive science society*, 2601–2606. Cognitive Science Society.

Sowerby, H.; Zhou, Z.; and Littman, M. L. 2022. Designing Rewards for Fast Learning. *arXiv preprint arXiv:2205.15400*.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Van Seijen, H.; Fatemi, M.; Romoff, J.; Laroche, R.; Barnes, T.; and Tsang, J. 2017. Hybrid reward architecture for reinforcement learning. *Advances in Neural Information Processing Systems*, 30.

Watkins, C. J.; and Dayan, P. 1992. Q-learning. *Machine learning*, 8(3): 279–292.

Wu, Z.; Lian, W.; Unhelkar, V.; Tomizuka, M.; and Schaal, S. 2021. Learning dense rewards for contact-rich manipulation tasks. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 6214–6221. IEEE.

Yu, T.; Quillen, D.; He, Z.; Julian, R.; Hausman, K.; Finn, C.; and Levine, S. 2020. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. In *Conference on robot learning*, 1094–1100. PMLR.

Zheng, Z.; Oh, J.; Hessel, M.; Xu, Z.; Kroiss, M.; Van Hasselt, H.; Silver, D.; and Singh, S. 2020. What can learned intrinsic rewards capture? In *International Conference on Machine Learning*, 11436–11446. PMLR.

Ziebart, B. D.; Maas, A. L.; Bagnell, J. A.; Dey, A. K.; et al. 2008. Maximum entropy inverse reinforcement learning. In *Aaai*, volume 8, 1433–1438. Chicago, IL, USA.

A RL Practitioner Survey

We invited RL practitioners from 2 Fortune 500 company's AI research divisions and from 2 US Research Universities (R1) to participate in this survey. Practitioners were entered into a raffle for a $15 USD gift card in exchange for participating. Practitioners were screened based on whether or not they had designed a reward function in the past year. Only those who affirmed this were able to proceed with the survey. Multiple options may be selected for any given question. 24 practitioners completed the survey. The survey questions are as follows, and the number of respondents for each option are indicated:

- What domain have you designed a reward function for *most recently*? (Mark all that apply, but only for your most recent domain)
  - A gridworld task: 8 respondents
  - A classic RL task – like CartPole or Mountain Car: 2 respondents
  - A robotics task: 12 respondents
  - A multi-agent task (e.g., Hanabi): 5 respondents
  - An Atari game or other game: 5 respondents
  - Other: please specify: 3 respondents: A continuous control task, a racing game, and a command and control task
- How did you write an initial reward function? (Mark all that apply.)
  - By applying intuition and considering how the agent would learn: 15 respondents
  - By embedding domain knowledge of how the agent should behave: 13 respondents
  - Using a reward function someone else had written (e.g., from a publication or shared implementation): 15 respondents
  - By specifying a performance objective for the task: 10 respondents
  - By inverse reinforcement learning, or some other demonstration-based approach: 6 respondents
  - Other: please specify: 1 respondent: By defining a goal region
- Did you shape your reward function?
  - Yes: 15 respondents
  - No: 6 respondents
  - I don't know: 1 respondent
  - Other: please specify: 2 respondents: Tried, but didn't have much success; Using human feedback to shape
- Did you use trial-and-error to refine your reward function?
  - Yes: 22 respondents
  - No: 2 respondents
  - Other: please specify: 0 respondents
- During trial-and-error reward design, how did you evaluate your reward function(s)?
  - By viewing the agent's behavior after it has finished training (and would not be trained further): 18 respondents
  - By viewing the agent's behavior during a training session (that was continued further): 12 respondents
  - By plotting some performance metric against time or learning iterations: 15 respondents
  - By plotting return from the changing reward function against time or learning iterations: 6 respondents
  - By scoring example trajectories: 2 respondents
  - Other: please specify: 0 respondents
- Did you observe suboptimal behavior after training your agent?
  - Yes - I observed the agent taking advantage of a loophole in the reward function, so I changed the function to remove the loophole: 7 respondents
  - Yes - I observed the agent's behavior plateauing at unsatisfactory performance, so I changed the reward function to help it learn beyond the plateau: 13 respondents
  - Yes - I observed the agent's behavior plateauing at unsatisfactory performance, so I changed the learning algorithm or hyperparameter. : 11 respondents
  - No: 4 respondents
  - Other: please specify: 0 respondents

B User Study Details

B.1 Expert Participant Recruitment

To recruit study participants, we sent recruitment emails to relevant listservs of computer science researchers and to faculty at four US research universities (R1). These faculty members passed the recruitment email along to their research groups.

B.2 User Study Protocol

The next five pages contain a screenshot of the Jupyter notebook used for the first 55 minutes of the study.
The expert participants first read a description of the problem, as follows:

- The goal of the hungry-thirsty domain is to teach an agent to eat as much as possible. There's a catch, though: the agent can only eat when it's not thirsty. Thus, the agent cannot just "hang out" at the food location and keep eating because at some point it will become thirsty and eating will fail.
- The agent always exists for 200 timesteps.
- The grid is 4×4^3. Food is located in one randomly-selected corner, while water is located in a different (random) corner.
- At each timestep, the agent may take one of the following actions: move (up, down, left, right), eat, or drink.
- But actions can fail:
  - The drink action fails if the agent is not at the water location.
  - The eat action fails if the agent is thirsty, or if the agent is not at the food location.
  - The move action fails if the agent tries to move through one of the red barriers (depicted below).
- If the agent eats, it becomes not-hungry for one timestep.
- If the agent drinks, it becomes not-thirsty.
- When the agent is not-thirsty, it becomes thirsty again with 10% probability on each successive timestep.

The experts then run a cell which shows a GIF of a Hungry Thirsty agent.

The experts are then tasked with specifying the reward function, learning algorithm, and hyperparameters, as shown in the following pages. To avoid biasing participants toward trial-and-error reward design, the order in which they were asked to select the reward function or the algorithm choice was randomized.

For the reward function, the experts set the rewards for each state $\texttt{H} \wedge \texttt{T}, \texttt{H} \wedge \neg\texttt{T}, \neg\texttt{H} \wedge \texttt{T}, \neg\texttt{H} \wedge \neg\texttt{T}$ from the range $[-1, 1]$ in 0.05 increments. The hyperparameters vary slightly across algorithms — for example, DDQN uses an ϵ-greedy action selection strategy, while PPO and A2C instead use an entropy term for exploration. Specifically, PPO and A2C use this entropy term in their loss functions, and this requires the user to set an entropy coefficient for regularization.

After selecting their choice of reward function, learning algorithm, and hyperparameters, the expert can start training the agent. As it trains, the Jupyter notebook plots three graphs. The first is the true metric performance for each episode, which corresponds to the stated goal: "The goal of the hungry-thirsty domain is to teach an agent to eat as much as possible." The second graph corresponds to the undiscounted return for each episode, which is based on the expert's own reward function. The third and final graph consists of a state visitation distribution heatmap, which shows where the agent is spending its time. Darker red means more state visits, and lighter red means fewer visits.

After training an agent (or cutting training short), the expert could then review training in three different ways. They could `select_run_and_show_agent()`, which allows them to see any trained agent's performance for a single, randomly-initialized episode. They could instead `view_training_runs()`, which allows them to review the training graphs for any agent. Or they could `review_past_run()`, which allows them to review their reward function, algorithm choice, and hyperparameter selections for any past agent.

Finally, 55 minutes into the study, experts were asked to submit their final, best configuration. They could choose from any of the configurations they tried during the study.

B.3 Follow Up Questions

After 55 minutes, participants submitted their best attempt at training an agent to solve Hungry Thirsty. In the remaining 5 minutes, we asked participants five structured questions:

1. Please describe your approach to training RL agents.
2. How well does this process mimic your past experience of training RL agents?
3. Is there some missing information from this interface which you wished you had access to?
4. Does your submitted agent meet your expectations?
5. Did you shape your reward function?

[3] For the first 12 participants, the grid was instead 6×6

Hungry-Thirsty Domain

The goal of the hungry-thirsty domain is to teach an agent to eat as much as possible.
There's a catch, though: the agent can only eat when it's not thirsty.
Thus, the agent cannot just “hang out” at the food location and keep eating because at some point it will become thirsty and eating will fail.

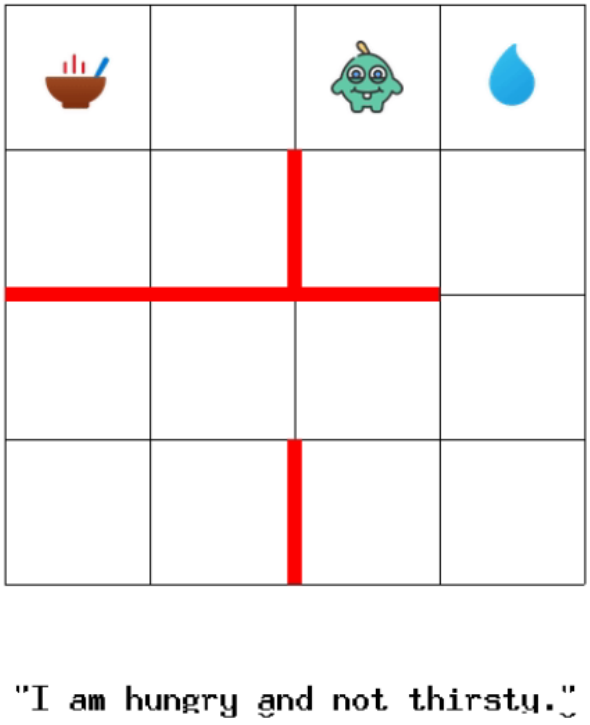
- The agent always exists for 200 timesteps.
- The grid is 4x4. Food is located in one randomly-selected corner, while water is located in a different (random) corner.
- At each timestep, the agent may take one of the following actions: move (up, down, left, right), eat, or drink. But actions can fail:
  - The drink action fails if the agent is not at the water location.
  - The eat action fails if the agent is thirsty, or if the agent is not at the food location.
  - The move action fails if the agent tries to move through one of the red barriers (depicted below).
- If the agent eats, it becomes not-hungry for one timestep.
- If the agent drinks, it becomes not-thirsty.
- When the agent is not-thirsty, it becomes thirsty again with 10% probability on each successive timestep.

See an example of the game below.

In [1]:
```
# import notebooks; do not edit
%run setup_reward_and_learning_alg.ipynb
%run training_and_model_eval.ipynb

# show agent
from IPython.display import Image
Image("../Assets/h-t-small.gif", width=450)
```

Out[1]:



Your Task

We haven't specified the reward function, learning algorithm, or algorithm hyperparameters; you'll need to fill in these details using the code cells below.

Your Study ID

Replace the YOUR_NAME string with your full name to generate a unique study ID.

Note: do *not* re-run the cells after entering data using these Jupyter notebook widgets.

In [2]:
```
set_study_id()
```

Your name: AAAI_2023 Submit

```
Your study ID is: b25b8922b18b04834f444d98afb1d6ea.
```

Design your Reward Function and Select Your RL Algorithm

For your reward function, you need to assign a value r(s) to each state. The state is composed of the thirst and hunger status of the agent. You must assign a value for each state:

- r(Hungry AND Thirsty) = ???; r(Hungry AND Not Thirsty) = ???; r(Not Hungry AND Thirsty) = ???; r(Not Hungry AND Not Thirsty) = ???

For your RL algorithm, you will need to choose your algorithm. Your options are:

- A2C, DDQN, PPO

After choosing your RL algorithm, you will need to select the hyperparameters. If you change your learning algorithm, you may need to re-run the hyperparamater selection (below).

In [3]:
```
# select the learning algorithm and reward function parameters
selectors = reward_and_alg_selector()
selectors
```

Reward for state: hungry AND thirsty -0.50

Reward for state: hungry AND not thirsty -0.25

Reward for state: not hungry AND thirsty 0.75

Reward for state: not hungry AND not thirsty 1.00

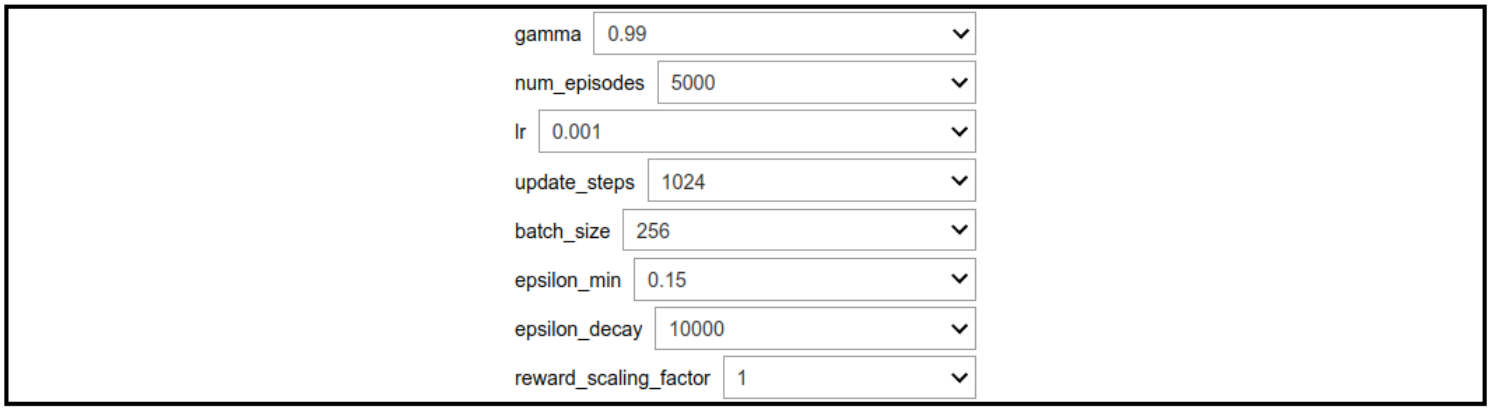
Algorithm Choice DDQN

Hyperparameters

!!! If you change your learning algorithm selection, you will need to rerun the following hyperparameter selection cell as well. !!!

- Hyperparameters Required for All Algorithms:
  - **Gamma**: the discount factor for the environment. A small Gamma means the agent prioritizes only immediate rewards (i.e., the agent is myopic), while a larger Gamma means the agent tends to also consider future rewards.
  - **Num_Episodes**: the number of episodes to train for. A smaller number means the experiments are faster, but contain less experience to learn from.
  - **Learning Rates**: the learning rate is used for training all networks: the Q-network for DDQN, and the actor and critic networks for A2C and PPO. Smaller learning rates make smaller updates to the network weights (and hence optimization is slower), while larger learning rates make larger updates.
  - **reward_scaling_factor**: a mulitplicative factor (σ) applied to the reward function defined above: $r'(s) = \sigma r(s)$. If set to 1, the reward function is unchanged.
- For A2C and PPO:
  - **Entropy_Coeff**: Only applicable to A2C and PPO, this is the entropy regularization coefficient which rewards entropy in the loss function. A smaller value means the loss encourages a less uniform distribution over actions (meaning less exploration, more exploitation).
- For DDQN and PPO:
  - **Update_Steps**: Only applicable to DDQN and PPO, this is the frequency with which to perform updates. A smaller number means more frequent updates, which is slower but more information dense.
- A2C Only:
  - **n_step_update**: How many steps should the agent take before updating the actor-critic network? A smaller number means more frequent updates, which is slower and higher variance. A larger number means less frequent updates, which is faster and lower variance.
- DDQN Only:
  - **Epsilon_Min**: DDQN uses an epsilon-greedy strategy. Epsilon decreases over time to encourage initial exploration (starting at epsilon=1). epsilon_min corresponds to the floor for the epsilon value. A larger epsilon_min means more exploration, less exploitation. A smaller epsilon min means less exploration, more exploitation.
  - **Epsilon_Decay**: Over time, epsilon decreases from 1 to epsilon_min. Every time step, epsilon decreases by 1/epsilon_decay.
  - **Batch_Size**: The number of samples to take from the experience replay buffer from which to calculate the loss and update the deep Q Network. A smaller number is faster to run but contains less experience.
- PPO Only:
  - **Eps_Clip**: In PPO, the estimated advantage function is clipped to handle variance. If the probability ratio between the new policy and the old policy falls outside the range (1 — ε) and (1 + ε), the advantage function is clipped. A smaller eps_clip value is more permissive; a larger eps_clip value is more restrictive and allows for less substantial policy changes.

In [4]:
```
# select the learning algorithm hyperparameters
# !!! If you change the algorithm choice, you will need to re-run this !!!
alg = get_params(widget_ref=selectors)["Algorithm Choice"]
select_learning_alg_params = construct_hyperparam_selector(alg_name = alg)
select_learning_alg_params
```



Training Time!

For evaluating our reward functions, algorithm selection, and hyperparameters, we plot training performance according to *fitness* and *undiscounted return*.

Each episode consists of a trajectory $\tau = [(s_0, a_0, s_1), (s_1, a_1, s_2), \dots]$.

Fitness is computed as the sum of states in which the agent is not hungry:

- Fitness := $\Sigma_{(s,a,s')\in\tau}\ \mathbb{1}$(s[is_hungry] == False)

Undiscounted return is computed using the reward function you specified:

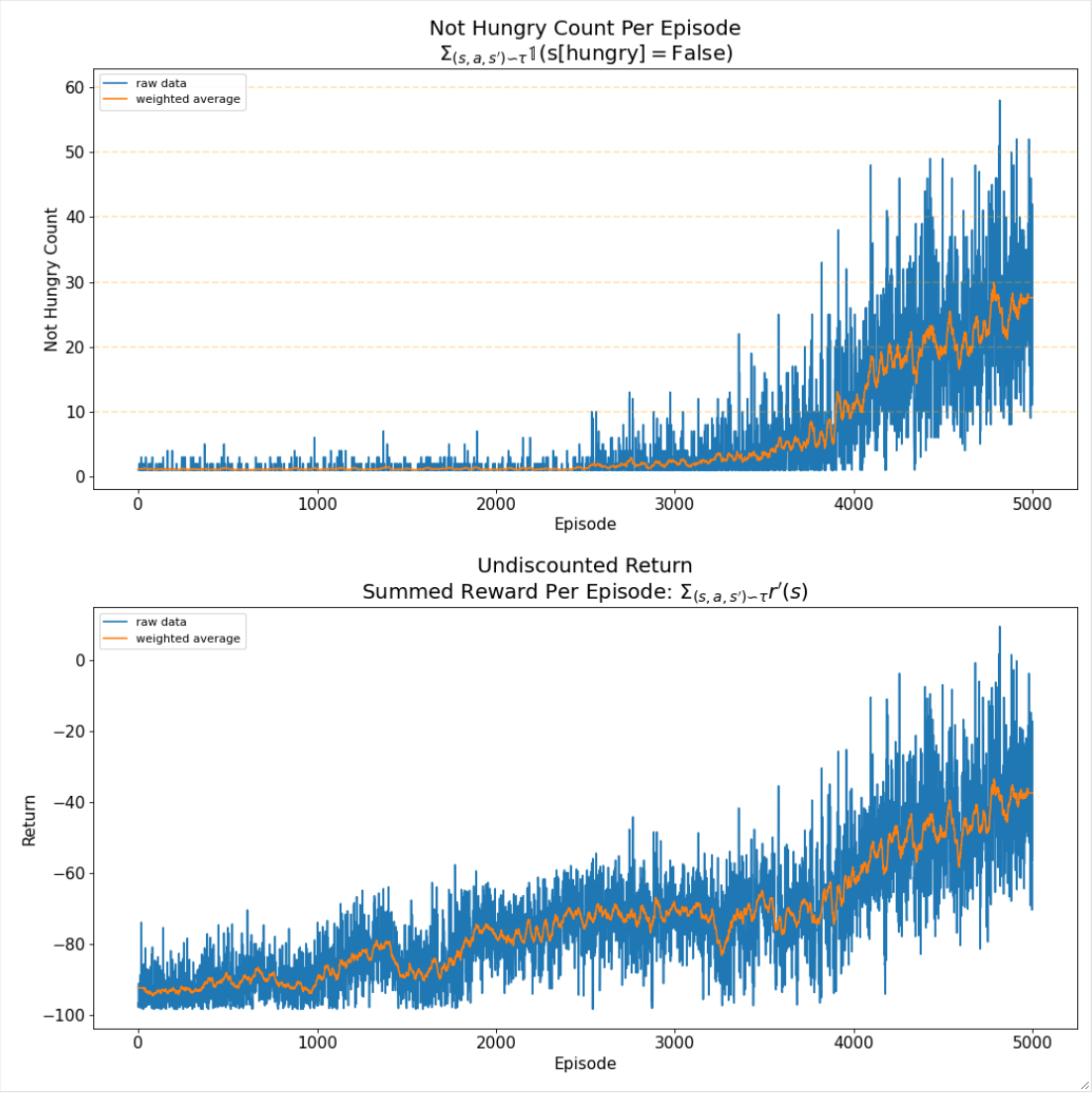
- Undiscounted Return := $\Sigma_{(s,a,s')\in\tau}\sigma r(s) = \Sigma_{(s,a,s')\in\tau} r'(s)$

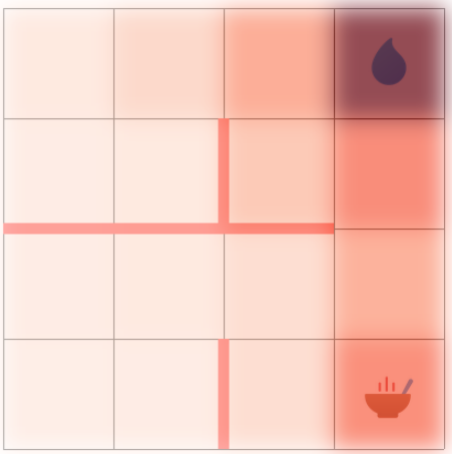
You may wish to go back and change one or more of your reward function, learning algorithm, or learning algorithm parameters.

You can cut training off early, but you won't be able to resume training a partially-trained agent.

In [5]:
```
%matplotlib notebook

train_agent(alg_and_reward_params=get_params(widget_ref=selectors),
            hyper_params=get_params(widget_ref=select_learning_alg_params),
            study_id=study_id)
```





In [ ]:
```
# view the agent
select_run_and_show_agent()
```

In [ ]:
```
view_training_runs()
```

In [ ]:
```
review_past_run()
```

Final Submission

When you are finished training your agent(s) and choosing which agent is best, run this cell and make your selection.

If the agent you submit is a top-10 performer in this user study, we will award you a $10 bonus after the conclusion of our study.

In [ ]:
```
submit_agent()
```

C Computational Experiments

In this section, we include supporting analysis for the computational experiments.

Table 3, Fig. 3, and Fig. 5 correspond to **H1: Reward functions are not universally effective.**

- Table 3 presents the Hoeffding Bound and Mann Whitney U-test p-values to compare the average cumulative mean performances achieved by policies learned with reward functions across varied hyperparameters (e.g., $\gamma = 0.99$ vs $\gamma = 0.8$).
- Fig. 3 presents parallel coordinate plots, highlighting the reward functions which resulted in the largest absolute difference in true cumulative performance across varied hyperparameters (e.g., $\gamma = 0.99$ vs $\gamma = 0.8$).
- Fig. 5 presents parallel coordinate plots, highlighting the reward functions which resulted in the largest absolute difference in true cumulative performance across varied algorithm choices (e.g., PPO vs. DDQN).

Table 4, Fig. 4, and Fig. 6 correspond to **H2: Reward functions are not universally optimal.**

- Table 4 presents the Hoeffding Bound and Mann Whitney U-test p-values to compare the average cumulative mean performances achieved by the best-performing policies learned with reward functions across varied hyperparameters (e.g., $\gamma = 0.99$ vs $\gamma = 0.8$).
- Fig. 4 presents parallel coordinate plots, highlighting the reward functions which result in the highest true cumulative performance for each hyperparameter (e.g., $\gamma = 0.99$).
- Fig. 6 presents parallel coordinate plots, highlighting the reward functions which result in the highest true cumulative performance for each algorithm (e.g., DDQN).

Table 3: **H1: Reward functions are not universally effective.** For each large-scale computational comparison experiment (e.g., $\gamma = 0.99$ vs. $\gamma = 0.5$), we find the 3 reward functions which result in the maximal difference in the cumulative true performance metric. To confirm that these differences are not simply simply due to sampling bias, we retrain agents using each of these reward functions 1000 additional times. We then compute the 90% Hoeffding bound as well as a Mann Whitney U-test over this larger set of data. In all cases, we find that the 90% Hoeffding bounds are non-overlapping, and that the difference in underlying distributions are statistically significant. In sum, these reward functions all result in high performance for one experiment variation, and low performance for another variation.

Reward Function	Experiment	Hoeffding Bound	p-value
[-1.0, -1.0, 0.5, -0.5]	$\gamma = 0.99$ $\gamma = 0.5$	[-2,471; 12,798] [85,830; 101,099]	< 0.01
[-0.05, -0.1, 1.0, 0.5]	$\gamma = 0.99$ $\gamma = 0.5$	[91,486; 106,754] [2,312; 17,580]	< 0.01
[-0.5, -0.5, -0.1, -0.05]	$\gamma = 0.99$ $\gamma = 0.5$	[968; 16,237] [88,656; 103,925]	< 0.01
[-1.0, -0.5, 1.0, 0.0]	$\gamma = 0.99$ $\gamma = 0.8$	[13,800; 29,068] [45,594; 60,863]	< 0.01
[-1.0, -1.0, -1.0, 0.1]	$\gamma = 0.99$ $\gamma = 0.8$	[4,965; 20,234] [86,653; 101,922]	< 0.01
[-0.5, -0.5, 0.0, 0.1]	$\gamma = 0.99$ $\gamma = 0.8$	[14837, 30105] [88,180; 103,449]	< 0.01
[-0.5, -0.05, 0.5, 0.5]	$\alpha = 0.25$ $\alpha = 0.05$	[20,034; 35,303] [70,224; 85,493]	< 0.01
[-1.0, -0.1, -0.1, 1.0]	$\alpha = 0.25$ $\alpha = 0.05$	[15,061; 30,330] [53,716; 68,985]	< 0.01
-0.1, -0.5, -1.0, 1.0]	$\alpha = 0.25$ $\alpha = 0.05$	[23,954, 39,223] [68,009; 83,278]	< 0.01

Table 4: **H2: Reward functions are not universally optimal.** Comparisons of reward function optimality. For each experiment configuration (i.e., $\gamma = 0.99$), we find the 3 best-performing reward functions from the grid search, where each reward function is tested 10 times. As shown in Figure 4, none of the 'optimal' reward functions are shared across experiment variations. To assess whether these relationships are not simply due to sampling bias, we re-run these experiments with these top reward functions for 1000 trials each, and report the results here for each top-3 reward function combinations. Specifically, we fix an experimental test condition (i.e., $\gamma = 0.99$), and assess whether the top reward function for that experiment (i.e., $[-0.05, -0.05, 0.5, 0.5]$) outperforms the top reward functions for alternative test conditions (i.e., [-1.0, -1.0, 0.0, 1.0], which is optimal for $\gamma = 0.5$). To make this assessment, we compute the 90% Hoeffding Bound of the average cumulative reward. With 90% probability, the true mean lies within this bound. Second, we compute a Mann Whitney U-test to assess whether the distribution of performance means corresponding to the optimal reward function is greater than that of its comparison, and we report the p-values from these tests. In this table, all but two comparisons show statistical significance.

Test	Reward Function	Selection	Hoeffding Bound	p-value
$\gamma = 0.99$	[-0.05, -0.05, 0.5, 0.5]	#1 for $\gamma = 0.99$	[90442, 105710]	< 0.01
	[-1.0, -1.0, 0.0, 1.0]	#1 for $\gamma = 0.5$	[52436, 67704]	
$\gamma = 0.99$	[-0.05, -0.05, 0.5, 0.5]	#1 for $\gamma = 0.99$	[90442, 105710]	< 0.01
	[-0.05, -0.05, 0.0, 1.0]	#2 for $\gamma = 0.5$	[85587, 100856]	
$\gamma = 0.99$	[-0.05, -0.05, 0.5, 0.5]	#1 for $\gamma = 0.99$	[90442, 105710]	< 0.01
	[-1.0, -1.0, 0.1, 0.1]	#3 for $\gamma = 0.5$	[9998, 25266]	
$\gamma = 0.5$	[-1.0, -1.0, 0.0, 1.0]	#1 for $\gamma = 0.5$	[91071, 106339]	0.99
	[-0.05, -0.05, 0.5, 0.5]	#1 for $\gamma = 0.99$	[93017, 108286]	
$\gamma = 0.5$	[-1.0, -1.0, 0.0, 1.0]	#1 for $\gamma = 0.5$	[91071, 106339]	< 0.01
	[-0.05, -0.1, 1.0, 0.5]	#2 for $\gamma = 0.99$	[2311, 17580]	
$\gamma = 0.5$	[-1.0, -1.0, 0.0, 1.0]	#1 for $\gamma = 0.5$	[91071, 106339]	< 0.01
	[-0.5, -0.1, 1.0, 0.5]	#3 for $\gamma = 0.99$	[15209, 30478]	
$\gamma = 0.99$	[-0.05, -0.05, 0.5, 0.5]	#1 for $\gamma = 0.99$	[90442, 105710]	< 0.01
	[-0.05, 0.0, 0.05, 0.5]	#1 for $\gamma = 0.8$	[73526, 88795]	
$\gamma = 0.99$	[-0.05, -0.05, 0.5, 0.5]	#1 for $\gamma = 0.99$	[90442, 105710]	< 0.01
	[-0.05, 0.0, 0.5, 0.5]	#2 for $\gamma = 0.8$	[77842, 93111]	
$\gamma = 0.99$	[-0.05, -0.05, 0.5, 0.5]	#1 for $\gamma = 0.99$	[90442, 105710]	< 0.01
	[-1.0, -1.0, 0.1, 1.0]	#3 for $\gamma = 0.8$	[53252, 68520]	
$\gamma = 0.8$	[-0.05, 0.0, 0.05, 0.5]	#1 for $\gamma = 0.8$	[80029, 95298]	1.0
	[-0.05, -0.05, 0.5, 0.5]	#1 for $\gamma = 0.99$	[99649, 114917]	
$\gamma = 0.8$	[-0.05, 0.0, 0.05, 0.5]	#1 for $\gamma = 0.8$	[80029, 95298]	< 0.01
	[-0.05, -0.1, 1.0, 0.5]	#2 for $\gamma = 0.99$	[73935, 89204]	
$\gamma = 0.8$	[-0.05, 0.0, 0.05, 0.5]	#1 for $\gamma = 0.8$	[80029, 95298]	< 0.01
	[-0.5, -0.1, 1.0, 0.5]	#3 for $\gamma = 0.99$	[56372, 71641]	
$\alpha = 0.25$	[-0.1, -0.1, 1.0, 0.05]	#1 for $\alpha = 0.25$	[69028, 84297]	< 0.01
	[-0.05, -0.05, 0.5, 0.5]	#1 for $\alpha = 0.05$	[46150, 61419]	
$\alpha = 0.25$	[-0.1, -0.1, 1.0, 0.05]	#1 for $\alpha = 0.25$	[69028, 84297]	< 0.01
	[-0.05, -0.1, 1.0, 0.5]	#2 for $\alpha = 0.05$	[53966, 69235]	
$\alpha = 0.25$	[-0.1, -0.1, 1.0, 0.05]	#1 for $\alpha = 0.25$	[69028, 84297]	< 0.01
	[-0.5, -0.1, 1.0, 0.5]	#3 for $\alpha = 0.05$	[29049, 44318]	
$\alpha = 0.05$	[-0.05, -0.05, 0.5, 0.5]	#1 for $\alpha = 0.05$	[90442, 105710]	0.01
	[-0.1, -0.1, 1.0, 0.05]	#1 for $\alpha = 0.25$	[79718, 94987]	
$\alpha = 0.05$	[-0.05, -0.05, 0.5, 0.5]	#1 for $\alpha = 0.05$	[90442, 105710]	< 0.01
	[-0.05, 0.0, 1.0, 0.1]	#2 for $\alpha = 0.25$	[69325, 84594]	
$\alpha = 0.05$	[-0.05, -0.05, 0.5, 0.5]	#1 for $\alpha = 0.05$	[90442, 105710]	< 0.01
	[-0.05, -0.05, 0.5, 0.0]	#3 for $\alpha = 0.25$	[64327, 79596]	

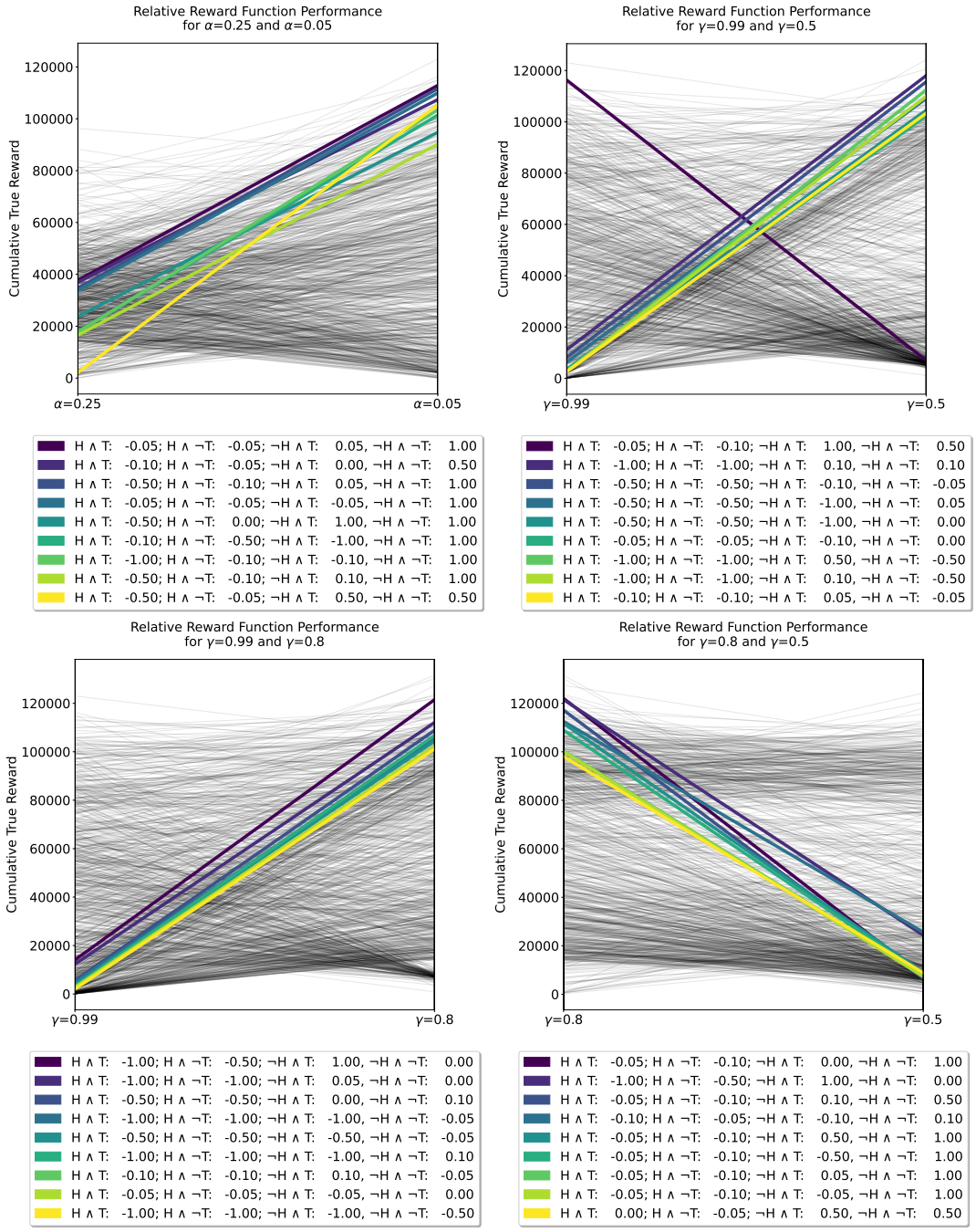


Figure 3: Parallel coordinate plots which correspond to **H1: Reward functions are not universally effective.** This figure shows plots for $\alpha = 0.25$ vs. $\alpha = 0.05$, $\gamma = 0.99$ vs. $\gamma = 0.5$, and $\gamma = 0.99$ vs. $\gamma = 0.8$. Each line represents a reward function's performance, as measured by the true cumulative performance metric achieved by an average policy trained with the hyperparameter shown on the $x-$axis. The reward functions which result in the highest absolute difference in performance are highlighted. For example, $[-0.05, -0.05, 0.05, 1.0]$ resulted in a cumulative performance of approximately $40,000$ when $\alpha = 0.25$, but instead resulted in a cumulative performance of approximately $100,000$ when $\alpha = 0.05$.

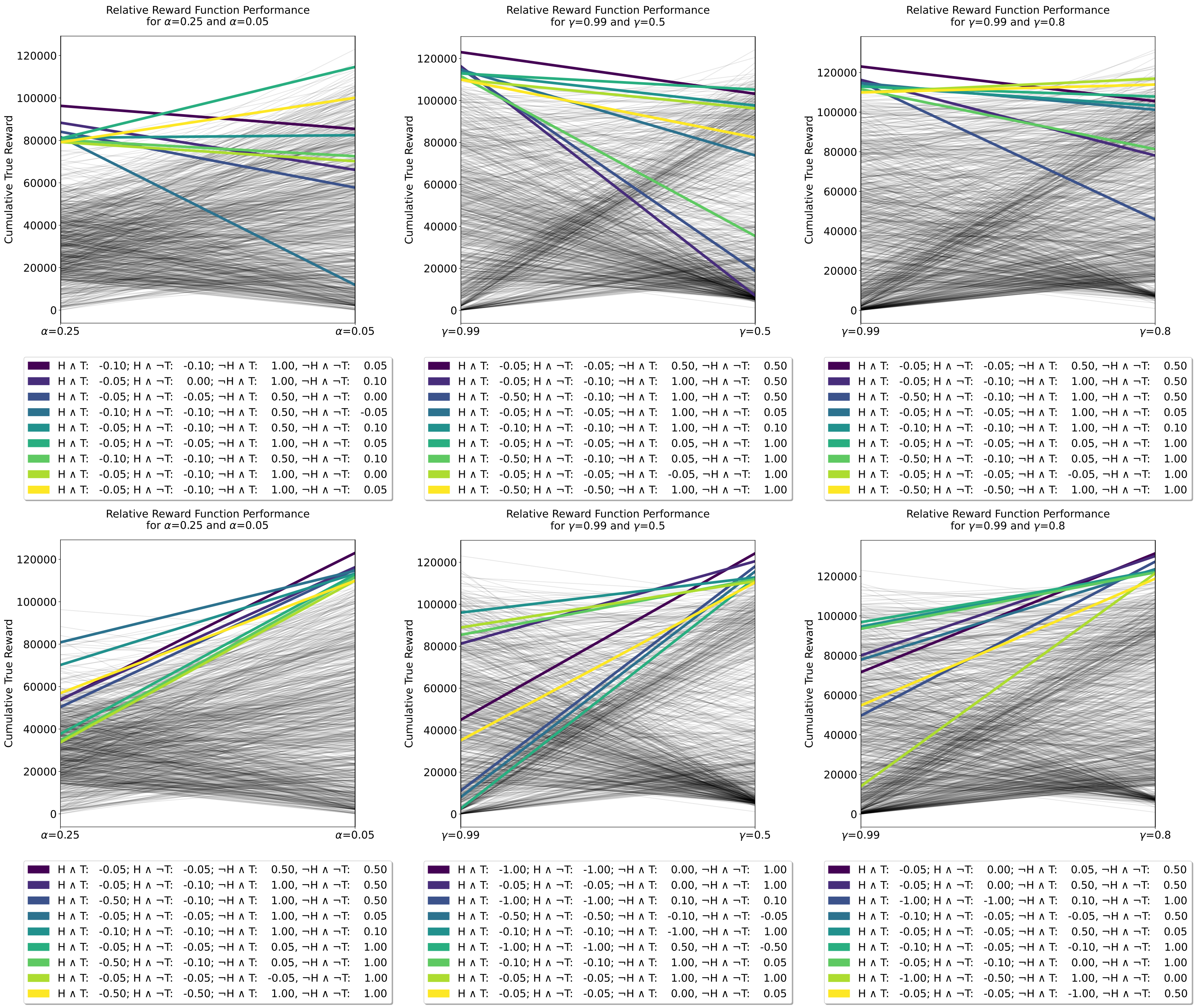


Figure 4: Parallel coordinate plots which correspond to **H2: Reward functions are not universally optimal.** This figure shows plots for $\alpha = 0.25$ vs. $\alpha = 0.05$, $\gamma = 0.99$ vs. $\gamma = 0.5$, and $\gamma = 0.99$ vs. $\gamma = 0.8$. Each line represents a reward function's performance, as measured by the true cumulative performance metric achieved by an average policy trained with the hyperparameter shown on the $x-$axis. In the top row, the reward functions which result in the best cumulative performance for the first condition—i.e., $\alpha = 0.25$, $\gamma = 0.99$, and $\gamma = 0.99$ respectively—are highlighted. In the bottom row, the reward functions which result in the best cumulative performance for the second condition—i.e., $\alpha = 0.05$, $\gamma = 0.5$, and $\gamma = 0.8$ respectively—are highlighted.

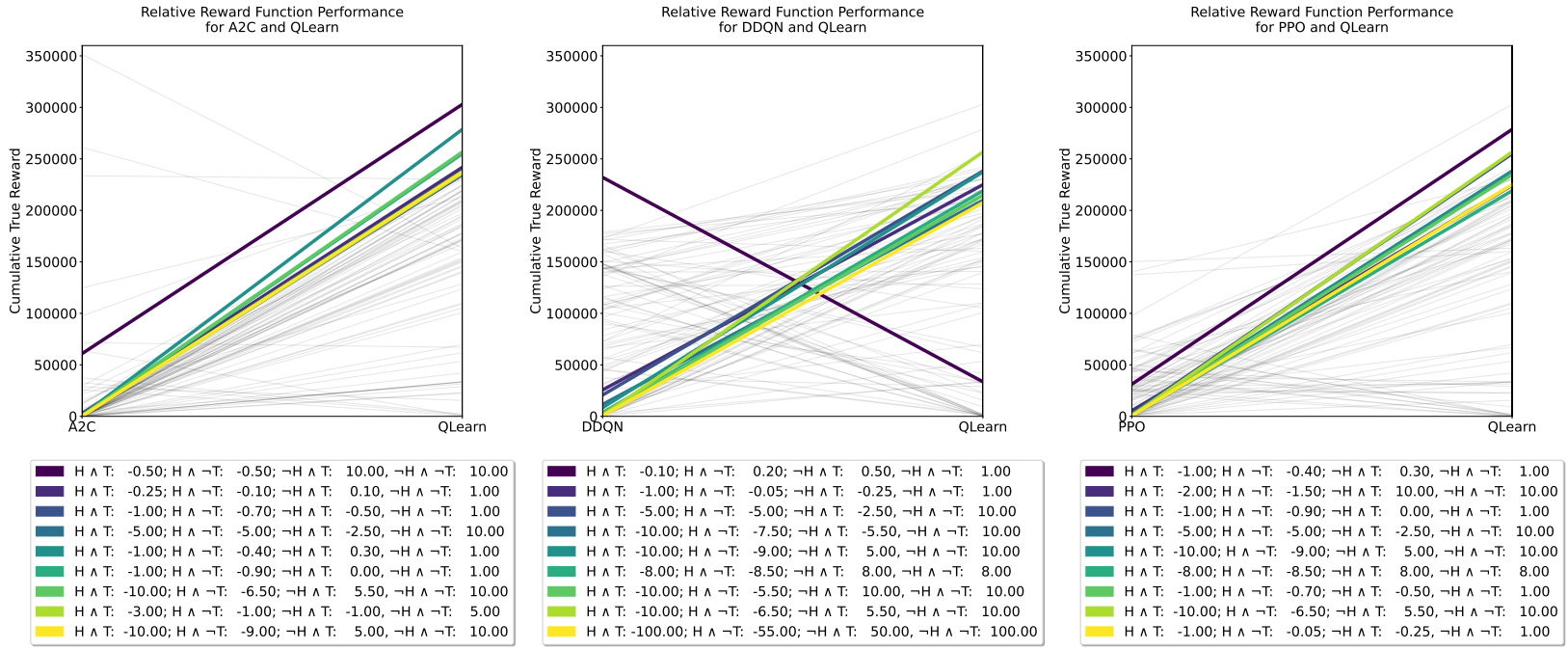


Figure 5: Parallel coordinate plots which correspond to **H1: Reward functions are not universally effective.** This figure shows plots for A2C vs. Q-learning, DDQN vs. Q-learning, PPO vs. Q-learning, A2C vs. DDQN, PPO vs. DDQN, and PPO vs. A2C. Each line represents a reward function's performance, as measured by the true cumulative performance metric achieved by an average policy trained with algorithm shown on the $x-$axis, using the standard hyperparameters as described in Apdx. Section D. The reward functions which result in the highest absolute difference in performance are highlighted. For example, $[-0.50, -0.50, 10.00, 10.00]$ resulted in a cumulative performance of approximately $60,000$ when trained with A2C, but instead resulted in a cumulative performance of approximately $300,000$ with Q-learning. Note that these algorithms are each trained for 5000 episodes (instead of the 2000 used for comparing performance across hyperparameter changes).

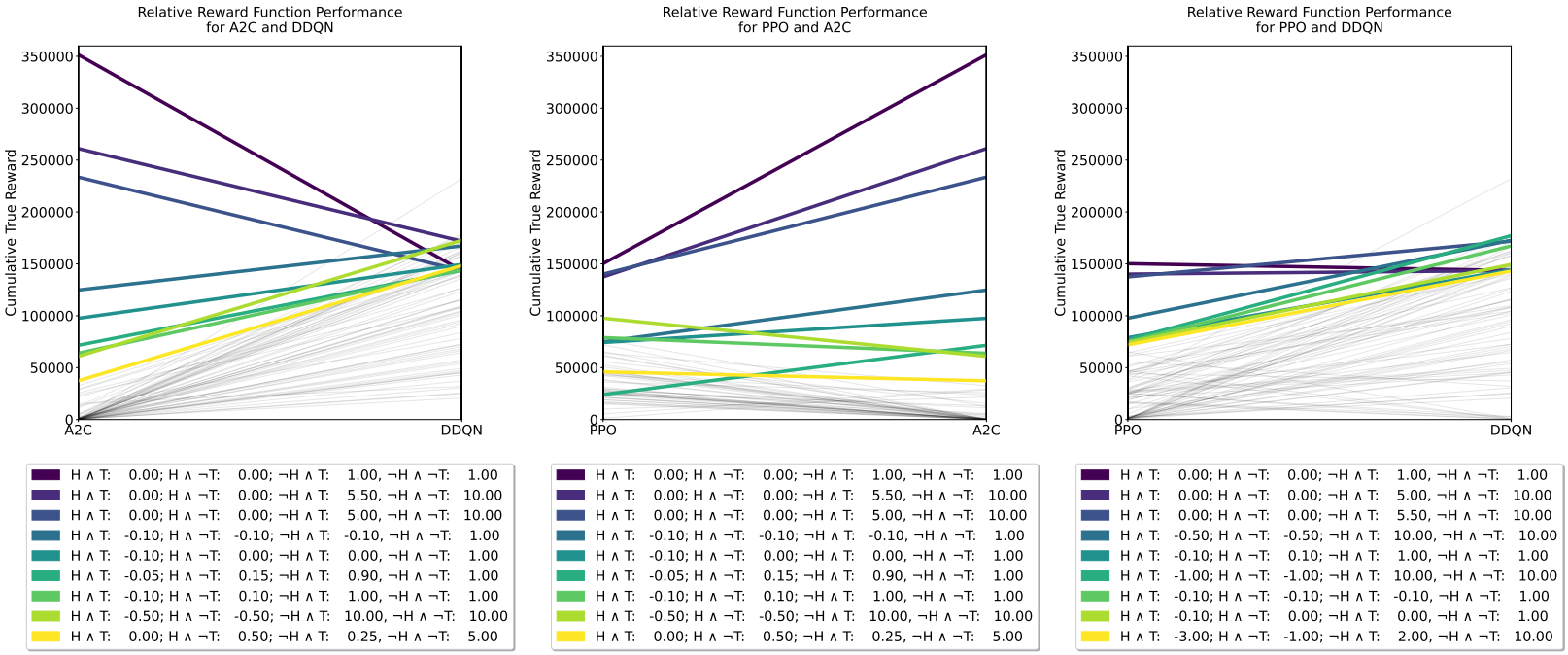


Figure 6: Parallel coordinate plots which correspond to **H2: Reward functions are not universally optimal.** This figure shows plots for A2C vs. DDQN, PPO vs. A2C, and PPO vs. DDQN. Each line represents a reward function's performance, as measured by the true cumulative performance metric achieved by an average policy trained with the algorithm shown on the $x-$axis. In the top row, the reward functions which result in the best cumulative performance for the first condition—i.e., A2C, PPO, and PPO respectively—are highlighted. In the bottom row, the reward functions which result in the best cumulative performance for the second condition—i.e., DDQN, A2C, and DDQN respectively—are highlighted. The best-performing reward functions are different for A2C vs. DDQN and PPO vs. DDQN, but not for PPO vs. A2C. In this last case, the optimal reward function corresponds to the sparse reward function $[0, 0, 1, 1]$.

D Deep RL Implementation Details & Hyperparameters

Alongside this paper, we release all of the code for our experiments—including the implementations of the RL algorithms we use: Q-Learning, A2C, DDQN, and PPO. It is known that these algorithms are not only sensitive to hyperparameters, but also potentially to details of the implementation (Engstrom et al. 2019). Unless specified elsewhere in the text, we use the hyperparameters presented in Table 5.

Table 5: Summary of the standard hyperparameters and related implementation details used with each RL algorithm.

Standard Hyperparameter Value	Q-Learning	A2C	DDQN	PPO
γ, discount factor	0.99	0.99	0.99	0.99
α, learning rate	0.05	0.001	0.001	0.005
number of training episodes	2000	5000	5000	5000
neural net hidden layer size	—	144	144	144
neural net structure	—	Actor/Critic	Q-Network/Q-Network	Actor/Critic
neural net activation function	—	ReLU	ReLU	ReLU
entropy coefficient	—	0.01	—	0.01
ϵ-greedy coefficient	0.15	—	0.15	—
ϵ-decay rate	—	—	10000	—
n-step network updates	—	20	—	—
experience replay buffer size	—	—	5000	—
update steps	—	—	128	800
batch size	—	—	128	80
ϵ clipping coefficient (trust region)	—	—	—	0.2

E User Study Overfitting

Say a user selects reward function r_i, but tested several other reward functions, $r_1, r_2, \ldots, r_n$. In the main text, we assess whether users overfit their reward functions to their selected algorithms and hyperparameters by assessing whether *any* of the user's tested reward functions r_j, where $j \in [1, n]$ and $r_i \neq r_j$, outperformed the user's final reward function significantly using *any* of the studied algorithms (DDQN, PPO, and A2C with fixed hyperparameters). Here we propose an alternative metric for overfitting in this context. If the *average* performance of the user's selected reward function over all three tested algorithms is significantly worse than the *average* performance of one of the user's alternative reward functions, we claim it is overfit. As in the former evaluation, we again define the performance difference threshold to be 20000, accumulated over 5000 training episodes and averaged over 10 trials.

Using this alternative metric, we find 11 of 24 users (46%) overfit their reward functions. Note that 6 users are excluded from this evaluation, as their final reward functions were invalid even in the best-case environment configuration. For example, user **P28** submitted the reward function $[-1.0, -0.05, -0.25, 1.0]$, which achieved an average cumulative performance of 8793 across the implementations of DDQN, PPO, and A2C. This same user tested but did not select the alternative reward function $[-1.0, -0.1, 0.0, 1.0]$, which achieved an average cumulative performance of 35367 across the three tested algorithms. Since the performance difference is more than the selected threshold of 20000, we say this user overfit their reward function.

F Compute

We ran these experiments on a combination of local compute and cloud services. Our local compute consists of a machine with 64GB DDR4, a 12-core AMD Ryzen 9 5900 CPU, and a single NVIDIA GeForce RTX 3080 Ti GPU. We used a cloud service which has 480 nodes with Intel Xeon Platinum 8260 CPUs and 4 GB of RAM per core, with no GPUs available. All computers use Ubuntu as the OS. The description of all packages and the specific versions needed to run the code (e.g., NumPy) is included in the released codebase.

F.1 Deep RL Agents: Time to Train

Without any parallelization, training a DDQN agent for 5000 episodes on local compute with the standard hyperparameters took on average 186 seconds; training a PPO agent took 415 seconds; and, finally, training an A2C agent took 511 seconds.