DiReCL: Learning Differentiable Reward Code with Inverse Reinforcement Learning

Aoran Wang[1,2] **Jingtao Zhang**[3] **Maosen Li**[3] **Zongzhang Zhang**[1,2]*
[1] National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
[2] School of Artificial Intelligence, Nanjing University, Nanjing, China
[3] Yinwang Intelligent Technologies Co., Ltd., Shenzhen, China
wangar@lamda.nju.edu.cn, {zhangjingtao,limaosen}@yinwang.com,
zzzhang@nju.edu.cn

Abstract

In reinforcement learning (RL), reward specification plays a fundamental role in shaping policy optimization. Inverse RL (IRL) offers a data-driven approach to inferring rewards from demonstrations, but widely used neural-network-based methods often produce rewards that are difficult to interpret, inspect, or revise manually. In contrast, human-written or large language model (LLM)-generated reward code provides explicit semantic structure, yet its precise numerical parameters are often difficult to calibrate and prone to specification errors. To bridge this gap, we propose DiReCL (**Di**fferentiable **Re**ward **C**ode **L**earning), a novel framework that unifies these two paradigms by formulating reward learning in a differentiable reward code space. DiReCL first prompts an LLM to synthesize reward code templates, then converts the numeric elements in these templates into differentiable parameters and aligns them with expert demonstrations through IRL. We further introduce a reflection mechanism that uses IRL diagnostics to provide evidence of inadequacies in the reward structure and guide template revision. Experiments on four MuJoCo locomotion tasks and two autonomous driving benchmarks demonstrate that DiReCL achieves over $1.25\times$ the performance of the compared IRL and LLM-assisted reward design baselines, while retaining explicit and inspectable reward semantics.

1 Introduction

Despite significant advances in reinforcement learning (RL), reward specification remains a major obstacle to deploying AI agents in real-world scenarios [1, 2]. Poorly specified rewards can induce undesirable optimization behavior, including reward hacking and weak generalization to unseen environments [3, 4]. A central difficulty is that real-world objectives are often high-dimensional and qualitative, making them difficult to encode as precisely calibrated scalar functions [5]. For example, in autonomous driving, a reward must capture nuanced trade-offs among safety, efficiency, and passenger comfort, such as how strongly the vehicle should brake near a pedestrian crossing [6, 7].

Existing reward specification methods mainly fall into two paradigms, each with important limitations:

- **Data-driven reward learning**, such as Inverse RL (IRL), learns rewards from expert demonstrations [8–11]. Typical IRL methods parameterize rewards with unconstrained neural networks or feature combinations, providing expressive hypothesis spaces. However, these representations can be difficult to interpret and are susceptible to causal confusion [12] and weak generalization [13, 14], limiting their practical use in real-world applications.

*Zongzhang Zhang is the corresponding author.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

- **Heuristic reward design**, including human-designed and large language model (LLM)-assisted reward programming, faces different trade-offs. Manual reward design can incorporate human priors and yield interpretable semantic structures, but it often requires substantial domain expertise and costly trial-and-error refinement [15, 5]. Recent LLM-assisted methods can synthesize reward code from natural-language task descriptions [16–18], but their numerical choices are not directly grounded in expert behavior and often require careful calibration to induce effective policies.

The complementary strengths and limitations of these two paradigms suggest a core challenge in reward specification: how can we combine interpretable semantic reward structures with data-driven numerical calibration? Semantic reward structures provide an explicit organization of task objectives and constraints, whereas learnable parameters enable fine-grained adaptation for effective RL training. To address this challenge, we propose **Di**fferentiable **Re**ward **C**ode **L**earning (DiReCL), which treats reward specification as learning in a differentiable reward code space. DiReCL first uses an LLM to generate reward code, and then optimizes its learnable numerical parameters through differentiable IRL objectives. This yields a trainable reward representation that remains semantically structured while being calibrated from demonstrations.

Concretely, DiReCL parses the generated reward code into an abstract syntax tree (AST), relaxes selected non-differentiable operations into differentiable substitutes, and identifies eligible numeric elements as learnable parameters. These parameters are updated through an IRL optimization loop while the underlying reward code structure is preserved. This design allows DiReCL to improve the behavioral grounding of the reward without discarding its explicit semantic organization.

Beyond numerical parameter alignment, DiReCL addresses cases where numerical calibration alone is insufficient. We introduce a new reward code reflection mechanism that analyzes IRL optimization diagnostics, including policy return, imitation gap, parameter dynamics, and bad-case summaries, to inform reward structure revision. These diagnostics are summarized and provided to the LLM to guide the next round of reward code generation and optimization.

We evaluate DiReCL on robotic control and autonomous driving tasks. The results show that the learned rewards support effective downstream policy optimization and improve robustness under dynamics shifts while retaining explicit code-level semantics.

Our contributions are threefold.

- We propose DiReCL, a novel reward specification framework that formulates reward learning as optimization in a differentiable reward code space, bridging the semantic structure of LLM-generated code with the adaptability of data-driven reward learning.
- We develop an IRL procedure for optimizing eligible numerical parameters inside reward code, and further introduce a diagnostic-driven reflection mechanism that revises reward templates when their structure limits behavioral alignment.
- We empirically evaluate DiReCL on continuous control and autonomous driving benchmarks, where it achieves stronger performance than representative IRL and LLM-assisted reward design baselines while preserving interpretable code-level reward structure.

2 Preliminaries

2.1 Inverse Reinforcement Learning (IRL)

We consider a discounted Markov decision process (MDP) without a specified reward, $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, d_0, \gamma \rangle$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition dynamics, d_0 is the initial state distribution, and $\gamma \in (0, 1]$ is the discount factor. Given a reward $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, the pair $(\mathcal{M}, r)$ induces a standard discounted MDP [1]. In regularized RL [19, 20], the optimal policy π^* for $(\mathcal{M}, r)$ is given by

$$\pi^* \in \arg\max_{\pi} \left\{ \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^t \cdot r(s_t, a_t) \right] + H(\pi) \right\}, \tag{1}$$

where $\tau = \{(s_t, a_t)\}_{t=0}^{\infty}$ denotes a trajectory, and $H(\pi) = \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} -\gamma^t \log \pi(a_t \mid s_t) \right]$.

Given expert demonstrations generated by an expert policy π_E, IRL aims to recover a reward function r under which π_E is optimal for $(\mathcal{M}, r)$ according to Eq. 1. We consider a conventional regularized IRL framework in the following saddle-point form [8]:

$$\min_{r} \max_{\pi} \left\{ H(\pi) + \mathbb{E}_{\tau\sim\pi} \left[\sum_{t=0}^{\infty} \gamma^t \cdot r(s_t, a_t) \right] - \mathbb{E}_{\tau\sim\pi_E} \left[\sum_{t=0}^{\infty} \gamma^t \cdot r(s_t, a_t) \right] \right\}. \tag{2}$$

Intuitively, this formulation searches for a reward that separates expert behavior from the current policy behavior, while the inner RL computes an optimal policy under the current reward.

Recent IRL methods typically parameterize the reward as $r(s, a \mid \theta)$, where θ denotes the parameters of a neural reward network. For example, Maximum-Likelihood Inverse Reinforcement Learning (ML-IRL) [10] learns the reward by solving

$$\begin{aligned} \max_{\theta} \quad & \mathcal{L}(\theta) = \mathbb{E}_{\tau\sim\pi_E} \left[\sum_{t=0}^{\infty} \gamma^t \log \pi_\theta(a_t \mid s_t) \right] \\ \text{s.t.} \quad & \pi_\theta = \arg\max_{\pi} \left\{ \mathbb{E}_{\tau\sim\pi} \left[\sum_{t=0}^{\infty} \gamma^t \cdot r(s_t, a_t \mid \theta) \right] + H(\pi) \right\}. \end{aligned} \tag{3}$$

In this formulation, the reward parameters are optimized so that the entropy-regularized optimal policy induced by $r(s, a \mid \theta)$ assigns high likelihood to expert actions. The optimal solution θ^* in Eq. 3 induces a reward $r(s, a \mid \theta^*)$ corresponding to the optimal solution of the problem in Eq. 2.

2.2 Reward Code

Reward code can be written as $r(s, a \mid \mathcal{C})$, where $\mathcal{C}$ denotes a code fragment written by hand or generated by an LLM. The code takes task inputs and returns scalar reward values. Unlike neural reward networks, reward code exposes an explicit semantic decomposition of the objective through $\mathcal{C}$. For example, a locomotion reward may contain separate terms for forward progress, control cost, posture stability, and contact regularization [21, 22]. Similarly, a driving reward may encode route progress, collision avoidance, lane keeping, and comfort [23, 7].

DiReCL treats reward code as a structured hypothesis class with learnable internal parameters, rather than as a fixed reward function. We therefore represent reward code as an executable program $r(s, a \mid \mathcal{C}, \theta)$, explicitly separating the semantic structure of the reward code from its numerical calibration. The code template $\mathcal{C}$ specifies the semantic structure of the reward, including which task inputs are used and how different reward components are composed. In contrast, the numerical parameters θ specify the learnable values inside the template, such as coefficients, weights, and thresholds. Formally, we write

$$r_{\mathcal{C},\theta} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}, \qquad r_{\mathcal{C},\theta}(s, a) = r(s, a \mid \mathcal{C}, \theta),$$

where $\mathcal{C}$ determines the interpretable reward structure and θ determines its numerical calibration.

This formulation allows us to learn reward from data directly in a programmatic reward space. As a result, the learned reward can preserve an inspectable semantic structure while allowing its internal numerical details to be calibrated from demonstrations.

3 DiReCL: Differentiable Reward Code Learning

We now present **Di**fferentiable **Re**ward **C**ode **L**earning (DiReCL), a framework that learns rewards in a programmatic reward code space rather than in an unconstrained reward-function space. Given task information from a reward-free MDP, DiReCL first queries an LLM to generate human-interpretable reward code. It then transforms eligible numerical elements in the code into learnable parameters and aligns these parameters with expert demonstrations through IRL updates. When the current reward code is structurally insufficient, DiReCL uses learning diagnostics to augment the prompt context and revise the reward template. The overall pipeline is shown in Figure 1.

DiReCL represents the final reward code as a weighted composition of reward components:

$$r_\phi(s, a) = \sum_{n=1}^{N} \alpha_n \cdot r_{\mathcal{C}_n,\theta_n}(s, a), \tag{4}$$

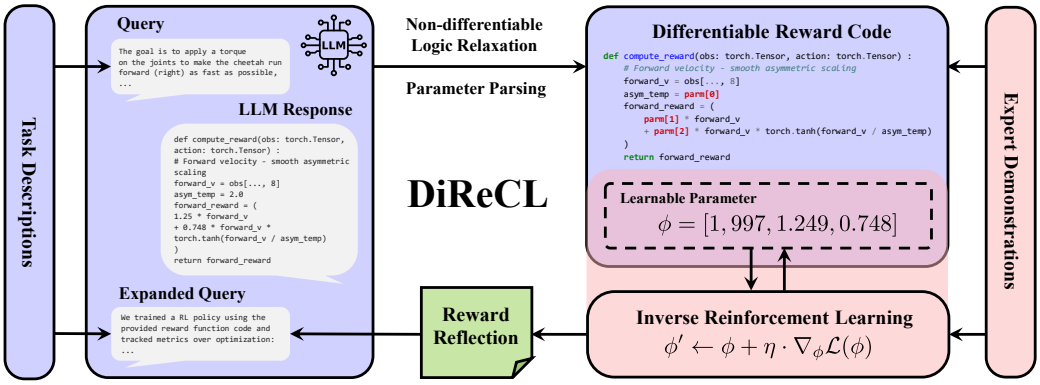


Figure 1: Pipeline of DiReCL. DiReCL queries an LLM with task descriptions to generate reward code, parses eligible numerical elements as learnable parameters (blue), optimizes these parameters with IRL gradients (red), and uses optimization diagnostics to revise the reward code (green).

where $\mathcal{C}_n$ denotes the n-th reward code template, θ_n denotes its learnable numerical parameters, and α_n denotes its component weight. Each component can contain nonlinear interactions. We use $\phi = \{\alpha_n, \theta_n\}_{n=1}^{N}$ to denote all learnable parameters.

3.1 Reward Code Generation

LLM generation Given a task description, environment specification, and observation and action semantics, we construct a prompt that queries the LLM to generate reward code with explicit semantic structure. The prompt encourages the LLM to express the reward through meaningful components, such as progress, stability, safety, or control cost, depending on the task domain. It also requires the generated code to implement a specified function signature so that it can be used directly in the learning loop. Specifically, we use the Python interface

```
def compute_reward(obs, action) -> float,
```

where `obs` and `action` denote the state and action inputs, respectively. This interface allows DiReCL to operate over a broad class of computable reward functions expressible as executable Python code. Appendix A provides the full prompt.

Non-differentiable logic relaxation LLM-generated reward code often contains non-differentiable logic, such as threshold branches, Boolean condition tests, and non-smooth operations such as maximization. Such logic can improve interpretability, but it can obstruct gradient-based optimization, especially when the numerical values inside the non-differentiable logic are intended to be learned. To address this issue, DiReCL replaces selected non-differentiable operations with differentiable surrogates. For example, consider the following threshold-based branch:

$$\psi(s,a) = \begin{cases} u(s,a), & \text{if } \; g(s,a) > m, \\ v(s,a), & \text{otherwise}, \end{cases}$$

where m is the threshold. When m corresponds to a tunable behavioral boundary, it can be promoted to a learnable parameter. If $u(s,a)$, $v(s,a)$, and $g(s,a)$ are differentiable functions, or have already been transformed into differentiable forms, we replace the threshold branch with the relaxation

$$\tilde{\psi}(s,a) = \sigma\left(\beta\left(g(s,a) - m\right)\right) \cdot u(s,a) + \left(1 - \sigma\left(\beta(g(s,a) - m)\right)\right) \cdot v(s,a),$$

where σ is the sigmoid function and β controls the relaxation sharpness. This transformation provides gradients for learnable parameters in $u(s,a)$, $v(s,a)$, and $g(s,a)$, as well as for the threshold m. Appendix A.2 lists all the non-differentiable operations we consider.

Parameter parsing After template generation and non-differentiable logic relaxation, DiReCL parses the resulting reward code into an abstract syntax tree (AST) and identifies eligible numeric literals as learnable parameters. We parameterize numeric literals whose continuous variation

preserves the semantic structure of the reward code. In contrast, we keep structural literals fixed, including indices, dimensions, and exponents of power operations, to prevent gradient updates from breaking the intended semantics of the reward code. For example, consider the following node

$$\texttt{0.5 * (s[i]**2 - 2.0)},$$

where `s[i]**2` denotes the square of the i-th dimension of the variable `s`. DiReCL transforms it into

$$\texttt{para[0] * (s[i]**2 + para[1])},$$

where `para[0]`, `para[1]` are parsed as learnable parameters and initialized to 0.5 and -2.0, while `s[i]**2` remains unchanged.

The LLM determines the number of reward terms N during generation. Applying non-differentiable logic relaxation and AST-based parameter parsing to the n-th generated template yields a differentiable reward code component $r_{\mathcal{C}_n,\theta_n}(s,a)$, where θ_n denotes the learnable parameters parsed from template $\mathcal{C}_n$. If parsing fails, the code is rejected and regenerated, with the error included in the updated LLM context. Finally, all reward code components are combined by the weighted sum in Eq. 4, so that the final reward is a linear composition of semantically meaningful components.

The same transformation procedure can also be applied to hand-crafted rewards, as long as the reward can be expressed as executable Python code. This extends DiReCL beyond LLM-generated rewards and allows it to calibrate manually designed reward codes as well.

3.2 Parameter Alignment with IRL

Given the differentiable reward code in Eq. 4, DiReCL aligns the reward parameters $\{\alpha_n, \theta_n\}_{n=1}^N$ with expert demonstrations through an alternating IRL procedure. The parameters are initialized from the numerical values proposed by the LLM. At each iteration, the current reward is used to train a policy, and the resulting agent trajectories are compared with expert trajectories to update the reward parameters. When the generated template already forms a weighted sum, α_n can be absorbed into θ_n by treating it as an additional learnable parameter. However, we separate α_n from $r_{\mathcal{C}_n,\theta_n}(s,a)$ in Eq. 4 to explicitly represent the final reward as a linear combination of generated components.

Let $\mathcal{D}_E$ denote the expert demonstrations with horizon T. With the current templates fixed, DiReCL optimizes $\phi = \{\alpha_n, \theta_n\}_{n=1}^N$ using the ML-IRL objective [10] in Eq. 3:

$$\begin{aligned} \max_{\phi} \quad & \mathcal{L}(\phi) = \mathbb{E}_{\tau\sim\mathcal{D}_E}\left[\sum_{t=0}^{T-1}\gamma^t \log \pi_\phi(a_t \mid s_t)\right] \\ \text{s.t.} \quad & \pi_\phi = \arg\max_{\pi}\left\{\mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T-1}\gamma^t\left(\sum_{n=1}^{N}\alpha_n \cdot r_{\mathcal{C}_n,\theta_n}(s_t,a_t)\right)\right] + H(\pi)\right\}. \end{aligned} \tag{5}$$

At iteration k, the current policy π_k is optimized by RL under the reward parameterized by ϕ_k. After collecting trajectories $\mathcal{D}$ with π_k, DiReCL updates ϕ_k using the estimated gradient of $\mathcal{L}(\phi_k)$:

$$\phi_{k+1} \leftarrow \phi_k + \eta \cdot \nabla_\phi \mathcal{L}(\phi_k), \tag{6}$$

where η is the learning rate. Specifically, the gradient components for α_n and θ_n in Eq. 6 are

$$\nabla_{\alpha_n}\mathcal{L}(\phi_k) = \mathbb{E}_{\tau\sim\mathcal{D}_E}\left[\sum_{t=0}^{T-1}\gamma^t r_{\mathcal{C}_n,\theta_n}(s_t,a_t)\right] - \mathbb{E}_{\tau\sim\mathcal{D}}\left[\sum_{t=0}^{T-1}\gamma^t r_{\mathcal{C}_n,\theta_n}(s_t,a_t)\right],$$

$$\nabla_{\theta_n}\mathcal{L}(\phi_k) = \alpha_n\left(\mathbb{E}_{\tau\sim\mathcal{D}_E}\left[\sum_{t=0}^{T-1}\gamma^t \nabla_{\theta_n} r_{\mathcal{C}_n,\theta_n}(s_t,a_t)\right] - \mathbb{E}_{\tau\sim\mathcal{D}}\left[\sum_{t=0}^{T-1}\gamma^t \nabla_{\theta_n} r_{\mathcal{C}_n,\theta_n}(s_t,a_t)\right]\right).$$

Thus, DiReCL follows the standard alternating update scheme of IRL, while enabling direct optimization of parameters inside interpretable reward code.

3.3 Reward Template Reflection

Parameter alignment can calibrate the numerical values in a reward template, but it cannot repair a structurally misleading or incomplete template. DiReCL therefore includes a reflection stage that uses IRL optimization diagnostics to revise the reward structure when necessary.

DiReCL invokes reflection when IRL diagnostics indicate that numerical calibration alone is insufficient. It summarizes the following diagnostic evidence.

IRL logs We record the IRL objective $\mathcal{L}(\phi_k)$ and policy-learning metrics such as return under the learned reward, episode length, and KL divergence. These logs indicate whether reward learning and policy optimization are making the intended progress.

Imitation gap We measure expert-agent discrepancies over task-relevant statistics. For a selected set of normalized features $\{f_j\}_{j\geq 0}$, such as forward velocity and control magnitude, we compute

$$\Delta_j = \mathbb{E}_{\tau\sim\mathcal{D}_E}\left[\sum_{t=0}^{T-1}\gamma^t f_j(s_t,a_t)\right] - \mathbb{E}_{\tau\sim\mathcal{D}}\left[\sum_{t=0}^{T-1}\gamma^t f_j(s_t,a_t)\right].$$

A large $|\Delta_j|$ indicates that the current reward-induced policy differs from the expert along feature f_j.

Parameter monitoring We report parameter diagnostics, including parameter changes from initialization, gradient norms, and whether parameters collapse, oscillate, or vanish. We also examine whether a reward component becomes inactive or dominating in the final reward. These diagnostics help identify structural problems such as misspecified or redundant terms.

Bad-case summary Once per reflection round, we query GLM-5.1 separately with task information and statistics from agent and expert trajectories. It uses feature deviations to identify the worst rollout and describes its dominant failure mode. We append its report to the revision prompt to help the code-generation LLM identify possible sources of misalignment.

After each round of IRL training, DiReCL progressively expands the initial context with evidence collected during reflection. The expanded context includes task information, the current reward code, learned parameter values, and the diagnostic evidence above. This turns reward revision from purely heuristic LLM-based generation into a data-informed template refinement process.

Given the expanded context, the LLM proposes a revised set of templates $\{\mathcal{C}'_n\}_{n=1}^{N'}$. DiReCL then reapplies non-differentiable logic relaxation and parameter parsing to obtain new learnable parameters ϕ'. Parameters associated with unchanged or semantically matched components are warm-started from their previous values, while newly introduced parameters are initialized from LLM-proposed heuristics. DiReCL then begins a new round of parameter alignment under the revised reward code.

4 Experiments

We evaluate whether DiReCL can learn rewards that support effective policy optimization while retaining clear semantic structure. Additional experimental details are provided in Appendix B. In the main experiments, we use GLM-5.1 [24] as the backbone for all methods that generate code with an LLM. Appendix B.5 compares two alternative LLM backbones.

4.1 Baselines

We compare DiReCL with representative IRL and LLM-assisted reward design baselines.

AIRL Adversarial Inverse Reinforcement Learning (AIRL) [9] learns a structured neural discriminator within a generative-adversarial imitation learning framework. It serves as a standard IRL baseline that learns rewards from expert demonstrations.

ML-IRL ML-IRL [10] learns neural rewards through a maximum-likelihood formulation.

LLM This baseline directly uses reward code generated by GLM-5.1, including the numeric scalars proposed by the LLM. It serves as an ablation that removes both parameter alignment and reflection.

EUREKA Evolution-driven Universal REward Kit for Agent (EUREKA) [16] uses an LLM to revise reward code iteratively through reflection on training scores. Unlike DiReCL, it does not ground numerical parameters in supervision. This baseline evaluates whether iterative LLM generation with reflection alone can produce well-calibrated reward code.

DiReCL w/o R DiReCL without Reflection (DiReCL w/o R) removes the reward template reflection stage. The LLM-generated reward code is still parameterized and optimized from demonstrations, but its structure is not revised using diagnostic feedback.

Table 1: MuJoCo results: mean ± standard deviation over five independent runs. EUREKA and DiReCL both perform four reflection iterations and evaluate their selected reward code.

Method	Ant	HalfCheetah	Hopper	Humanoid
Expert	5115.5 ± 413.9	10999.1 ± 136.5	3437.6 ± 8.5	5692.0 ± 520.3
AIRL	1508.6 ± 529.5	2718.9 ± 440.2	178.3 ± 68.6	363.1 ± 104.9
ML-IRL	4940.5 ± 297.4	12083.2 ± 264.6	3377.1 ± 40.8	5000.8 ± 287.8
LLM	3669.2 ± 472.4	9203.9 ± 628.3	2652.4 ± 480.2	5071.0 ± 986.4
EUREKA	4643.8 ± 502.3	10215.6 ± 727.9	3153.4 ± 149.9	6401.6 ± 473.4
DiReCL w/o R	4629.8 ± 879.5	11790.6 ± 2049.1	2954.6 ± 604.0	7035.2 ± 472.9
DiReCL	**5908.1 ± 153.2**	**15005.1 ± 1497.6**	**3700.0 ± 219.1**	**7913.0 ± 231.1**

We provide more comparisons with variants of IRL, EUREKA, and DiReCL in Appendix B.3. We do not include Generating Rewards As CodE (GRACE) [18] or intEractive LEarning froM dEmoNstraTion And Language (ELEMENTAL) [25] as baselines because their implementations are not publicly available. We also do not compare with Proximal Inverse Reward Optimization (PIRO) [11] because ML-IRL provides a competitive and more computationally efficient representative of recent state-of-the-art IRL methods on these benchmarks.

4.2 Benchmarks

MuJoCo We evaluate on four MuJoCo locomotion environments: Ant, HalfCheetah, Hopper, and Humanoid [21]. These benchmarks are widely used for evaluating reward learning methods, requiring dense reward signals for high-dimensional continuous control.

CommonRoad-RL We further evaluate on CommonRoad-RL tasks [23] constructed from the highD [26] and inD [27] datasets, which contain naturalistic vehicle trajectories collected by drones over German highways and road intersections. These tasks evaluate whether the learned rewards can induce effective driving policies in realistic road scenarios, where the objective involves safety, progress, comfort, and interaction-aware behavior.

4.3 Experimental Setup

For each MuJoCo task, the expert demonstration set contains 25 trajectories collected by a well-trained Soft Actor-Critic (SAC) [19] policy under the ground-truth environment reward. For CommonRoad-RL tasks, the expert demonstration set contains 100 trajectories collected similarly by a well-trained Proximal Policy Optimization (PPO) [28] policy.

After reward selection, we train a policy from scratch with the same downstream RL configuration within each task and evaluate it using the ground-truth environment reward. DiReCL selects the calibrated candidate with the lowest empirical IRL loss on the given demonstrations. The ground-truth environment reward is excluded from generation, calibration, reflection, and selection; it is used only for evaluation. Each DiReCL seed reruns generation, IRL, and policy training independently. Four MuJoCo and nine CommonRoad-RL reflections yield five and ten generation rounds, respectively.

4.4 Main Results

DiReCL learns effective rewards. Table 1 and 2 show that DiReCL achieves the strongest performance among the compared methods on both benchmarks. On all four MuJoCo tasks, policies trained with DiReCL rewards also achieve higher evaluation returns than the expert policies trained with the original task rewards.

DiReCL outperforms AIRL and ML-IRL when all three methods use the same expert demonstrations and downstream RL backbone. This comparison isolates the effect of the reward hypothesis space: the empirical improvement suggests that learning within a structured reward code space can provide a useful inductive bias compared with learning unconstrained neural rewards.

DiReCL also outperforms the LLM and EUREKA baselines. This result suggests that LLM-generated reward structure alone is not sufficient. Although LLMs can often propose semantically meaningful

Table 2: CommonRoad-RL results: mean ± standard deviation over five independent runs. EUREKA and DiReCL both perform nine reflection iterations and evaluate their selected code.

Method	highD Score	highD Success	inD Score	inD Success
Expert	14066.6 ± 154.6	0.948 ± 0.004	18094.4 ± 283.7	0.918 ± 0.010
AIRL	7351.5 ± 279.7	0.120 ± 0.030	8056.7 ± 438.8	0.078 ± 0.028
ML-IRL	10799.2 ± 409.9	0.626 ± 0.053	13323.1 ± 1089.5	0.514 ± 0.086
LLM	9804.8 ± 488.8	0.530 ± 0.046	13549.8 ± 413.1	0.544 ± 0.039
EUREKA	11399.2 ± 319.3	0.732 ± 0.048	14903.4 ± 321.4	0.676 ± 0.033
DiReCL w/o R	11298.0 ± 822.2	0.716 ± 0.057	14812.2 ± 637.9	0.666 ± 0.067
DiReCL	**12596.7 ± 148.1**	**0.920 ± 0.014**	**17352.0 ± 728.1**	**0.886 ± 0.036**

reward components, the numerical values in these components require grounding in expert behavior. By converting eligible values into learnable parameters and optimizing them with the IRL objective, DiReCL calibrates reward code toward more effective downstream policy learning.

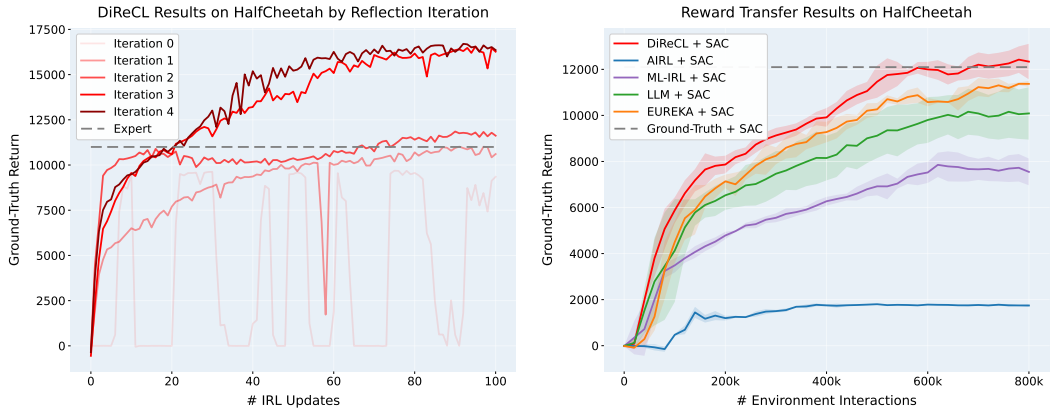
DiReCL reflection improves template quality. Across both benchmarks, DiReCL improves over DiReCL w/o R, indicating that parameter learning alone is insufficient when the initial reward template contains misleading or incomplete components. The reflection stage provides a mechanism for revising the template structure using learning diagnostics, thereby expanding the effective reward code hypothesis space. Figure 2a illustrates the improvements across reflection iterations in one HalfCheetah run, with the revised reward templates supporting progressively stronger IRL curves.

DiReCL learns transferable rewards. We further evaluate whether the learned reward code transfers to environments with modified dynamics. Specifically, we reuse the best reward learned on the source HalfCheetah task and train SAC from scratch on a target HalfCheetah task with altered dynamics, while keeping the ground-truth reward unchanged. Figure 2b reports the resulting evaluation curves. The plotted LLM-assisted rewards transfer more effectively than neural rewards under this dynamics shift, and DiReCL further improves over LLM and EUREKA through demonstration-grounded numerical calibration. Empirically, this suggests that interpretable reward code can capture task-relevant objectives while remaining less dependent on the source dynamics. In contrast, the neural rewards learned by AIRL and ML-IRL overfit to the dynamics and are more sensitive to the shift.

5 Related Work

Data-driven Reward Learning IRL provides a data-driven approach to learning reward functions from expert demonstrations. Although generative-adversarial IRL (e.g., AIRL [9], RAIRL [29]) and likelihood-based IRL (e.g., ML-IRL [10], PIRO [11]) have achieved significant success, they represent rewards as implicit neural functions whose opaque, rigid parameterizations hinder inspection, verification, and manual intervention. Beyond traditional IRL, generative-adversarial imitation methods (e.g., GAIL [30], IQ-Learn [31], AILBoost [32]) and preference-based learning [33, 34] effectively align policy learning using neural representations. However, these learned rewards are inherently non-reusable, limiting their utility for tasks requiring explicit, human-verifiable specifications. To overcome these limitations, we propose DiReCL, which learns rewards in a differentiable code space. By decoupling the semantic structure ((encoded in interpretable code) from the numerical parameters (calibrated from data), DiReCL ensures that the reward function remains accessible for inspection and adaptable to complex environments.

LLM-assisted Reward Design Recent work in LLM-assisted reward design seeks to automate the construction of interpretable rewards. Early approaches such as EUREKA [16] and Text2Reward [17] leverage LLMs to generate executable code from task descriptions, but suffer from numerical instability due to hallucinations and a lack of grounding in expert behavior. Methods based on vision-language models (VLMs), such as RL-VLM-F [35], use visual preferences to reduce annotation costs but retain the limitations of conventional preference-based learning. More recent hybrid methods attempt to bridge this gap: GRACE [18] uses adversarial feedback to guide code mutation, while ELEMENTAL [25] employs IRL to learn weights over features generated by LLMs. However, both methods leave the internal numerical parameters of the generated code fixed, preventing fine-



(a) DiReCL results by reflection iteration.

(b) Reward transfer results.

Figure 2: (a) IRL learning curves for one HalfCheetah run with the initial reward code (iteration 0) and each of four revisions (iterations 1–4). (b) Reward transfer to HalfCheetah with modified dynamics. For each method, we train SAC in the target environment using the best selected reward and evaluate performance over three random seeds.

grained adaptation to specific demonstrations. DiReCL makes numerical values in the generated code differentiable, combining interpretable reward structures with precise calibration from expert.

Interpretable Reward Learning Interpretability has long been an important concern in reward learning. A classical approach represents rewards as linear combinations of features [36, 37]. However, such interpretability depends on the availability and quality of predefined features, and learned neural features are often difficult to inspect. DiReCL instead expresses reward components in human-readable code with an explicit semantic structure.

Symbolic reward learning provides another form of interpretability. Interpretable Apprenticeship Learning with Temporal Logic Specifications [38] infers temporal-logic formulas from demonstrations, while Symbolic Reward Machine (SRM) [39] represents rewards as automata with predicates and performs Bayesian inference from demonstrations. These methods expose the reward structure but require task-specific predicates or atoms. Their logic languages are less flexible than general executable code. DiReCL uses executable code to specify more expressive rewards for complex tasks.

Recent methods draw on the world knowledge of pretrained LLMs and VLMs to generate interpretable rewards [16–18, 25]. These methods reduce manual reward engineering, but hallucination and prompt variation can make the numerical values they generate unreliable. DiReCL complements these approaches by treating the reward templates that LLMs propose and revise as learnable functions.

Generative reward models can also explain their feedback in natural language. For example, EvalPlanner [40] and Reward Modeling from Natural Language Human Feedback (RM-NLHF) [41] generate reward signals with natural-language rationales. However, such models typically require substantial supervision and training. DiReCL focuses on learning a reward code from limited expert demonstrations, and avoids using reward values directly generated by LLMs.

6 Conclusion and Future Work

We introduced DiReCL, a framework for learning differentiable reward code with IRL. Rather than treating reward code as a fixed heuristic, DiReCL represents it as a differentiable hypothesis and learns its parameters from demonstrations. Separating numerical calibration from semantic structure gives DiReCL a practical way to combine data-driven reward learning with heuristic reward design. Experiments on MuJoCo locomotion and autonomous driving benchmarks show that DiReCL learns rewards that support more effective policy learning than well-designed task rewards, while retaining explicit code-level interpretability.

Beyond LLM-generated rewards, DiReCL can also be applied to hand-crafted reward code, making it useful not only as an LLM-assisted reward design method but also as a general tool for calibrating and

improving existing human-written reward specifications. Several limitations suggest directions for future work. For tasks with high-dimensional visual inputs, incorporating VLMs can improve reward code generation and reflection through richer visual understanding. Moreover, our current formulation primarily grounds reward parameters using expert demonstrations. Future work can extend DiReCL to diverse forms of supervision, including preferences, rankings, and natural-language judgments.

Acknowledgments and Disclosure of Funding

This work is supported by the National Science Foundation of China (No. 62276126), the "111 Center" (No. B26023), and the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118).

The authors declare no additional competing interests.

References

[1] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT Press Cambridge, 1998.

[2] Michael Bowling, John D Martin, David Abel, and Will Dabney. Settling the reward hypothesis. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, pages 3003–3020, 2023.

[3] Dylan Hadfield-Menell, Smitha Milli, Pieter Abbeel, Stuart J Russell, and Anca Dragan. Inverse reward design. In *Advances in Neural Information Processing Systems (NIPS)*, pages 6765–6774, 2017.

[4] Alexander Pan, Kush Bhatia, and Jacob Steinhardt. The effects of reward misspecification: Mapping and mitigating misaligned models. In *Proceedings of the 10th International Conference on Learning Representations (ICLR)*, 2022.

[5] Calarina Muslimani, Kerrick Johnstonbaugh, Suyog Chandramouli, Serena Booth, W Bradley Knox, and Matthew E Taylor. Towards improving reward design in RL: A reward alignment metric for RL practitioners. *arXiv preprint arXiv:2503.05996*, 2025.

[6] Marco Cusumano-Towner, David Hafner, Alexander Hertzberg, Brody Huval, Aleksei Petrenko, Eugene Vinitsky, Erik Wijmans, Taylor W Killian, Stuart Bowers, Ozan Sener, et al. Robust autonomy emerges from self-play. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, pages 11710–11737, 2025.

[7] Bernhard Jaeger, Daniel Dauner, Jens Beißwenger, Simon Gerstenecker, Kashyap Chitta, and Andreas Geiger. CaRL: Learning scalable planning policies with simple rewards. In *Proceedings of the 9th Conference on Robot Learning (CoRL)*, pages 5301–5338, 2025.

[8] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI)*, pages 1433–1438, 2008.

[9] Justin Fu, Katie Luo, and Sergey Levine. Learning robust rewards with adversarial inverse reinforcement learning. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*, 2018.

[10] Siliang Zeng, Chenliang Li, Alfredo Garcia, and Mingyi Hong. Maximum-likelihood inverse reinforcement learning with finite-time guarantees. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 10122–10135, 2022.

[11] Yang Chen, Menglin Zou, Jiaqi Zhang, Yitan Zhang, Junyi Yang, Gael Gendron, Libo Zhang, Jiamou Liu, and Michael J Witbrock. Trust region reward optimization and proximal inverse reward optimization algorithm. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 37777–37808, 2025.

[12] Pim De Haan, Dinesh Jayaraman, and Sergey Levine. Causal confusion in imitation learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 11698–11709, 2019.

[13] Joar Max Viktor Skalse, Matthew Farrugia-Roberts, Stuart Russell, Alessandro Abate, and Adam Gleave. Invariance in policy optimisation and partial identifiability in reward learning. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, pages 32033–32058, 2023.

[14] Weichao Zhou and Wenchao Li. Rethinking inverse reinforcement learning: From data alignment to task alignment. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 27647–27688, 2024.

[15] Serena Booth, W Bradley Knox, Julie Shah, Scott Niekum, Peter Stone, and Alessandro Allievi. The perils of trial-and-error reward design: Misdesign through overfitting and invalid task specifications. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence (AAAI)*, pages 5920–5929, 2023.

[16] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

[17] Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2Reward: Reward shaping with language models for reinforcement learning. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

[18] Silvia Sapora, R Devon Hjelm, Omar Attia, Alexander Toshev, and Bogdan Mazoure. GRACE: A language model framework for explainable inverse reinforcement learning. In *Proceedings of the 14th International Conference on Learning Representations (ICLR)*, 2026.

[19] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 1861–1870, 2018.

[20] Matthieu Geist, Bruno Scherrer, and Olivier Pietquin. A theory of regularized Markov decision processes. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, pages 2160–2169, 2019.

[21] Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *Proceedings of the 25th IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5026–5033, 2012.

[22] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4:eaau5872, 2019.

[23] Xiao Wang, Hanna Krasowski, and Matthias Althoff. CommonRoad-RL: A configurable reinforcement learning environment for motion planning of autonomous vehicles. In *Proceedings of the 2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, pages 466–472, 2021.

[24] GLM-5-Team. GLM-5: From vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

[25] Letian Chen, Nina Marie Moorman, and Matthew Craig Gombolay. ELEMENTAL: Interactive learning from demonstrations and vision-language models for reward design in robotics. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, pages 8700–8725, 2025.

[26] Robert Krajewski, Julian Bock, Laurent Kloeker, and Lutz Eckstein. The highD dataset: A drone dataset of naturalistic vehicle trajectories on German highways for validation of highly automated driving systems. In *Proceedings of the 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 2118–2125, 2018.

[27] Julian Bock, Robert Krajewski, Tobias Moers, Steffen Runde, Lennart Vater, and Lutz Eckstein. The inD dataset: A drone dataset of naturalistic road user trajectories at German intersections. In *Proceedings of the 2020 IEEE Intelligent Vehicles Symposium (IV)*, pages 1929–1934, 2020.

[28] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[29] Wonseok Jeon, Chen-Yang Su, Paul Barde, Thang Doan, Derek Nowrouzezahrai, and Joelle Pineau. Regularized inverse reinforcement learning. In *Proceedings of the 8th International Conference on Learning Representations (ICLR)*, 2020.

[30] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *Advances in Neural Information Processing Systems (NIPS)*, pages 4565–4573, 2016.

[31] Divyansh Garg, Shuvam Chakraborty, Chris Cundy, Jiaming Song, and Stefano Ermon. IQ-Learn: Inverse soft-Q learning for imitation. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 4028–4039, 2021.

[32] Jonathan Daniel Chang, Dhruv Sreenivas, Yingbing Huang, Kianté Brantley, and Wen Sun. Adversarial imitation learning via boosting. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

[33] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems (NIPS)*, pages 4302–4310, 2017.

[34] Zhilong Zhang, Tian Xu, Xinghao Du, Xingchen Cao, Yihao Sun, and Yang Yu. Improving reward model generalization from adversarial process enhanced preferences. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, pages 76414–76435, 2025.

[35] Yufei Wang, Zhanyi Sun, Jesse Zhang, Zhou Xian, Erdem Biyik, David Held, and Zackory Erickson. RL-VLM-F: Reinforcement learning from vision language foundation model feedback. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, pages 51484–51501, 2024.

[36] Andrew Y Ng and Stuart Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning (ICML)*, pages 663–670, 2000.

[37] Marwa Abdulhai, Natasha Jaques, and Sergey Levine. Basis for intentions: Efficient inverse reinforcement learning using past experience. *arXiv preprint arXiv:2208.04919*, 2022.

[38] Daniel Kasenberg and Matthias Scheutz. Interpretable apprenticeship learning with temporal logic specifications. In *Proceedings of the 56th Annual Conference on Decision and Control (CDC)*, pages 4914–4921, 2017.

[39] Weichao Zhou and Wenchao Li. A hierarchical Bayesian approach to inverse reinforcement learning with symbolic reward machines. In *Proceedings of the 39th International Conference on Machine Learning (ICML)*, pages 27159–27178, 2022.

[40] Swarnadeep Saha, Xian Li, Marjan Ghazvininejad, Jason E Weston, and Tianlu Wang. Learning to plan & reason for evaluation with thinking-LLM-as-a-judge. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, pages 52565–52583, 2025.

[41] Zongqi Wang, Rui Wang, Yuchuan Wu, Yiyao Yu, Pinyi Zhang, Shaoning Sun, Yujiu Yang, and Yongbin Li. Reward modeling from natural language human feedback. *arXiv preprint arXiv:2601.07349*, 2026.

[42] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-Baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research (JMLR)*, 22:1–8, 2021.

[43] OpenAI. GPT-5.5 system card, April 2026. URL `https://openai.com/index/gpt-5-5-system-card/`.

[44] Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL `https://qwen.ai/blog?id=qwen3.5`.

A DiReCL Details

A.1 Full Prompt

The initial prompt used to query the LLM is as follows:

Initial Prompt

```
You are a reward-code designer for reinforcement learning tasks.

Unlike a standard LLM-assisted reward-design setting that asks the LLM
to directly provide a fully specified reward function, your goal here is
to design a differentiable programmatic reward template. The semantic
structure of the reward should reflect the task objective, while the
numerical coefficients, thresholds, temperatures, and trade-off weights
may later be calibrated from expert demonstrations by inverse
reinforcement learning.

Your reward function should use only information available from the
environment inputs. Depending on the environment, the reward function
signature can be either:

def compute_reward(
    obs: torch.Tensor,
    action: torch.Tensor,
    next_obs: torch.Tensor,
    info: Dict[str, Any]
) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    ...
    return reward, reward_components

or

def compute_reward(
    obs: torch.Tensor,
    action: torch.Tensor,
    info: Dict[str, Any]
) -> Tuple[torch.Tensor, Dict[str, torch.Tensor]]:
    ...
    return reward, reward_components

Please make sure that the code is compatible with PyTorch automatic
differentiation. Do not use NumPy operations, Python-side scalar
conversion such as .item(), or other operations that would detach the
reward from the computation graph. Any new tensor introduced in the
reward function must be created on the same device as the input tensors.

The output of the reward function should consist of two items:
    (1) the total reward as a torch.Tensor,
    (2) a dictionary containing each individual reward component.

When writing the reward function, follow these requirements:

(1) Focus on the semantic structure of the task reward. Decompose the
reward into meaningful components such as task progress, stability,
safety, energy efficiency, smoothness, or constraint satisfaction when
they are relevant to the task.

(2) Use explicit named scalar variables for important numerical
```

```
coefficients, thresholds, temperatures, margins, and trade-off weights.
These variables are expected to be parsed and treated as learnable
reward parameters in a later IRL stage. Avoid hiding important numerical
values inside complex expressions without names.

(3) The numerical values you assign should be reasonable initial values,
not necessarily final calibrated values. Do not overfit the reward by
assuming that the LLM can determine the optimal numerical calibration.

(4) If you transform a reward component, for example using torch.exp,
torch.tanh, or a smooth penalty, introduce a named temperature or scale
parameter inside the transformation. Each transformed component should
have its own named temperature or scale parameter.

(5) Keep the reward differentiable with respect to the learnable
numerical parameters whenever possible. Prefer smooth functions over
hard discontinuous conditions. Avoid non-differentiable branching,
equality checks, clipping, or indicator-style rewards unless they are
unavoidable for representing the task.

(6) Make sure the type of each input variable is correctly specified.
A float input variable should not be specified as torch.Tensor, and a
torch.Tensor input should not be converted into a Python float.

(7) The reward code must use only variables available from the provided
function inputs, namely obs, action, next_obs, and info when applicable.
Do not introduce additional input arguments. Do not assume access to
hidden simulator states, privileged oracle rewards, or variables that
are not described in the task information.

(8) The reward should be written as executable Python code. The code
should be self-contained except for standard imports such as torch and
typing. The returned component dictionary should use clear component
names that make the reward interpretable and useful for later analysis.

The task information is as follows:
< The task information >
```

We adapt the task information from the environment source code, removing the ground-truth reward while retaining the qualitative task description. We use HalfCheetah as an illustrative example below.

Task Information

```
class HalfCheetahEnv(MujocoEnv):

    """MuJoCo HalfCheetah-v4 environment.

    ### Description
    The HalfCheetah is a 2-dimensional robot consisting of 9 links
    and 8 joints connecting them (including two paws). The goal is
    to apply a torque on the joints to make the cheetah run forward
    (right) as fast as possible, with a positive reward allocated
    based on the distance moved forward and a negative reward
    allocated for moving backward. The torso and head of the cheetah
    are fixed, and the torque can only be applied on the other 6
    joints over the front and back thighs (connecting to the torso),
    shins (connecting to the thighs) and feet (connecting to the
    shins).
```

```
### Action Space

The action space is a `Box(-1, 1, (6,), float32)`. An action
represents the torques applied between *links*.

Action details:

- 0: Torque applied on the back thigh rotor
  - Control range: [-1, 1]
  - XML name: bthigh
  - Joint: hinge
  - Unit: torque (N m)

- 1: Torque applied on the back shin rotor
  - Control range: [-1, 1]
  - XML name: bshin
  - Joint: hinge
  - Unit: torque (N m)

- 2: Torque applied on the back foot rotor
  - Control range: [-1, 1]
  - XML name: bfoot
  - Joint: hinge
  - Unit: torque (N m)

- 3: Torque applied on the front thigh rotor
  - Control range: [-1, 1]
  - XML name: fthigh
  - Joint: hinge
  - Unit: torque (N m)

- 4: Torque applied on the front shin rotor
  - Control range: [-1, 1]
  - XML name: fshin
  - Joint: hinge
  - Unit: torque (N m)

- 5: Torque applied on the front foot rotor
  - Control range: [-1, 1]
  - XML name: ffoot
  - Joint: hinge
  - Unit: torque (N m)


### Observation Space

Observations consist of positional values of different body parts
of the cheetah, followed by the velocities of those individual
parts, i.e., their derivatives, with all the positions ordered
before all the velocities.

By default, observations do not include the x-coordinate of the
cheetah's center of mass. It may be included by passing
`exclude_current_positions_from_observation=False` during
construction. In that case, the observation space will have 18
dimensions, where the first dimension represents the x-coordinate
of the cheetah's center of mass.

Regardless of whether `exclude_current_positions_from_observation`
```

was set to true or false, the x-coordinate will be returned in
‘info‘ with key ‘"x_position"‘.

However, by default, the observation is a ‘ndarray‘ with shape
‘(17,)‘, where the elements correspond to the following.

Observation details:

- 0: z-coordinate of the front tip
  - Min/Max: -Inf/Inf
  - XML name: rootz
  - Joint: slide
  - Unit: position (m)

- 1: angle of the front tip
  - Min/Max: -Inf/Inf
  - XML name: rooty
  - Joint: hinge
  - Unit: angle (rad)

- 2: angle of the second rotor
  - Min/Max: -Inf/Inf
  - XML name: bthigh
  - Joint: hinge
  - Unit: angle (rad)

- 3: angle of the second rotor
  - Min/Max: -Inf/Inf
  - XML name: bshin
  - Joint: hinge
  - Unit: angle (rad)

- 4: velocity of the tip along the x-axis
  - Min/Max: -Inf/Inf
  - XML name: bfoot
  - Joint: hinge
  - Unit: angle (rad)

- 5: velocity of the tip along the y-axis
  - Min/Max: -Inf/Inf
  - XML name: fthigh
  - Joint: hinge
  - Unit: angle (rad)

- 6: angular velocity of front tip
  - Min/Max: -Inf/Inf
  - XML name: fshin
  - Joint: hinge
  - Unit: angle (rad)

- 7: angular velocity of second rotor
  - Min/Max: -Inf/Inf
  - XML name: ffoot
  - Joint: hinge
  - Unit: angle (rad)

- 8: x-coordinate of the front tip
  - Min/Max: -Inf/Inf
  - XML name: rootx

```
  - Joint: slide
  - Unit: velocity (m/s)

- 9: y-coordinate of the front tip
  - Min/Max: -Inf/Inf
  - XML name: rootz
  - Joint: slide
  - Unit: velocity (m/s)

- 10: angle of the front tip
  - Min/Max: -Inf/Inf
  - XML name: rooty
  - Joint: hinge
  - Unit: angular velocity (rad/s)

- 11: angle of the second rotor
  - Min/Max: -Inf/Inf
  - XML name: bthigh
  - Joint: hinge
  - Unit: angular velocity (rad/s)

- 12: angle of the second rotor
  - Min/Max: -Inf/Inf
  - XML name: bshin
  - Joint: hinge
  - Unit: angular velocity (rad/s)

- 13: velocity of the tip along the x-axis
  - Min/Max: -Inf/Inf
  - XML name: bfoot
  - Joint: hinge
  - Unit: angular velocity (rad/s)

- 14: velocity of the tip along the y-axis
  - Min/Max: -Inf/Inf
  - XML name: fthigh
  - Joint: hinge
  - Unit: angular velocity (rad/s)

- 15: angular velocity of front tip
  - Min/Max: -Inf/Inf
  - XML name: fshin
  - Joint: hinge
  - Unit: angular velocity (rad/s)

- 16: angular velocity of second rotor
  - Min/Max: -Inf/Inf
  - XML name: ffoot
  - Joint: hinge
  - Unit: angular velocity (rad/s)

### Starting State
All observations start in state (0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0,) with a noise added to the initial state for
stochasticity. As seen before, the first 8 values in the state
are positional and the last 9 values are velocity. A uniform
noise in the range of [-‘reset_noise_scale‘, ‘reset_noise_scale‘]
is added to the positional values while a standard normal noise
```

```
with a mean of 0 and standard deviation of ‘reset_noise_scale‘ is
added to the initial velocity values of all zeros.

### Episode End
The episode truncates when the episode length is greater than 1000.
"""

metadata = {
    "render_modes": [
        "human",
        "rgb_array",
        "depth_array",
    ],
    "render_fps": 20,
}

def __init__(
    self,
    reset_noise_scale=0.1,
    exclude_current_positions_from_observation=True,
    **kwargs,
):
    self._reset_noise_scale = reset_noise_scale
    self._exclude_current_positions_from_observation = (
        exclude_current_positions_from_observation
    )

    if exclude_current_positions_from_observation:
        observation_space = Box(
            low=-np.inf,
            high=np.inf,
            shape=(17,),
            dtype=np.float64,
        )
    else:
        observation_space = Box(
            low=-np.inf,
            high=np.inf,
            shape=(18,),
            dtype=np.float64,
        )

    MujocoEnv.__init__(
        self,
        "half_cheetah.xml",
        5,
        observation_space=observation_space,
        **kwargs,
    )

def step(self, action):
    x_position_before = self.data.qpos[0]
    self.do_simulation(action, self.frame_skip)
    x_position_after = self.data.qpos[0]
    x_velocity = (x_position_after - x_position_before) / self.dt

    observation = self._get_obs()
    terminated = False
    info = {
```

```
            "x_position": x_position_after,
            "x_velocity": x_velocity,
        }

        # Oracle task reward is intentionally hidden.
        reward = 0.0
        return observation, reward, terminated, False, info

    def _get_obs(self):
        position = self.data.qpos.flat.copy()
        velocity = self.data.qvel.flat.copy()

        if self._exclude_current_positions_from_observation:
            position = position[1:]

        observation = np.concatenate((position, velocity)).ravel()
        return observation

    def reset_model(self):
        noise_low = -self._reset_noise_scale
        noise_high = self._reset_noise_scale

        qpos = self.init_qpos + self.np_random.uniform(
            low=noise_low,
            high=noise_high,
            size=self.model.nq,
        )
        qvel = self.init_qvel + (
            self._reset_noise_scale
            * self.np_random.standard_normal(self.model.nv)
        )

        self.set_state(qpos, qvel)

        observation = self._get_obs()
        return observation
```

The reflection prompt used to query the LLM is as follows:

Prompt Expanded by Reflection

```
The current reward code is:
< The current reward >

We trained an RL policy using the current reward code and then updated
the learnable reward parameters using the IRL objective. The following
diagnostic information was collected during this process.

### IRL Logs
< IRL optimization logs >

The IRL logs include the learning scores, the
IRL loss and the policy optimization statistics. These logs describe
whether the reward parameters can be effectively grounded by expert
demonstrations and whether the learned reward provides useful guidance
for policy learning.

### Imitation Gap
< Imitation gap statistics >
```

```
The imitation gap measures the discrepancy between the policy induced
by the current reward code and the expert demonstrations. A small gap
indicates that the current reward code can explain the expert behavior,
whereas a large or stagnant gap suggests that the reward structure may
be missing important task-relevant components.

### Parameter Monitoring
< Parameter monitoring results >

The parameter monitoring results include the parsed learnable
parameters, their initial values, optimized values, gradients, and
update magnitudes. Use this information to identify which reward terms
are effective, which terms are insensitive, which parameters saturate,
and which numerical values may indicate an improper reward structure.

### Bad-case Summary
< Bad-case summary >

The bad-case summary includes representative failure cases observed
when the policy is trained with the current reward code. These cases may
include unsafe behaviors, unstable motions, lack of task progress,
excessive control effort, violation of constraints, or mismatch with
expert behavior...

Please analyze the above feedback and revise the reward code.

Your revision should follow these principles:

(1) If the imitation gap is small and the policy behavior is already
reasonable, keep the current reward structure and only refine the
numerical initialization, smoothness, or component balancing.

(2) If the imitation gap is large, stagnant, or inconsistent with the
IRL loss, diagnose whether the current reward code lacks necessary
semantic components. In this case, rewrite or expand the reward
structure instead of only changing coefficients.

(3) Use the parameter monitoring results to determine which components
should be preserved, weakened, removed, or redesigned. Parameters with
near-zero gradients, saturated values, unstable signs, or extremely
large updates may indicate ineffective or ill-conditioned reward terms.

(4) Use the bad-case summary to add missing task-relevant components.
For example, add terms for safety, stability, progress, smoothness,
constraint satisfaction, or expert-aligned behavior when the bad cases
show that these aspects are not sufficiently captured.

(5) Keep the reward code fully differentiable with respect to the
learnable numerical parameters. Prefer smooth penalties and rewards
over hard thresholds, discontinuous branches, equality checks, or
indicator-style terms.

(6) Use explicit named scalar variables for important coefficients,
thresholds, temperatures, margins, and trade-off weights, since these
values will be parsed as learnable reward parameters in the next IRL
round.

(7) Do not introduce new input arguments. Use only variables available
```

```
from the existing reward-function signature and the provided task
information.

(8) The output should be Python code only. Keep the reward-function
signature consistent with the current reward code, return the total
reward and a dictionary of interpretable reward components, and make the
code compatible with PyTorch automatic differentiation.
```

A.2 Non-differentiable Logic Relaxation

LLM-generated reward programs often contain hard logical constructs, such as threshold comparisons, branches, clipping, or discrete selections. Although common in manually written reward code, these operations may impede or destabilize gradient-based learning of reward parameters because the reward becomes discontinuous or piecewise constant with respect to some learnable parameters. In DiReCL, we retain the semantic decomposition while replacing supported hard logic with smooth surrogates. This allows gradients to flow not only through arithmetic reward weights, but also through learnable thresholds and gating parameters.

Let $\sigma(x) = 1/(1+\exp(-x))$ denote the sigmoid function, and let $\tau > 0$ be a temperature parameter controlling the sharpness of the relaxation. A smaller value of τ gives a closer approximation to the hard logic, whereas a larger value yields smoother gradients. In the reported experiments, we fix the shared temperature at $\tau = 0.1$ without annealing.

Hard comparison A binary comparison such as $\mathbb{I}[x > c]$ is relaxed by a soft gate:

$$\mathbb{I}[x > c] \quad \Longrightarrow \quad g_\tau(x, c) = \sigma\left(\frac{x-c}{\tau}\right).$$

Similarly,

$$\mathbb{I}[x < c] \quad \Longrightarrow \quad \sigma\left(\frac{c-x}{\tau}\right).$$

When the threshold c is learnable, this relaxation permits differentiation with respect to c.

If-else branching A hard branch

$$r(x) = \begin{cases} r_1(x), & x > c, \\ r_2(x), & x \le c, \end{cases}$$

is relaxed as a soft mixture:

$$r(x) \quad \Longrightarrow \quad g_\tau(x, c)\, r_1(x) + (1 - g_\tau(x, c))\, r_2(x),$$

where $g_\tau(x, c) = \sigma((x-c)/\tau)$. The mixture approximates the hard branch away from the threshold and weights the two branches equally at the threshold.

Equality condition Exact equality checks such as $\mathbb{I}[x = y]$ are usually unsuitable for continuous control because exact equality rarely holds for real-valued state variables. We relax equality by a smooth proximity kernel:

$$\mathbb{I}[x = y] \quad \Longrightarrow \quad \exp\left(-\frac{(x-y)^2}{\tau}\right),$$

For tolerance-based conditions, the alternative is the soft gate:

$$\mathbb{I}[|x-y| < \epsilon] \quad \Longrightarrow \quad \sigma\left(\frac{\epsilon - |x-y|}{\tau}\right).$$

This form accommodates an explicit tolerance threshold in the original reward code.

Interval condition We relax the interval indicator $\mathbb{I}[a < x < b]$ by multiplying two soft gates:

$$\mathbb{I}[a < x < b] \quad \Longrightarrow \quad \sigma\left(\frac{x-a}{\tau}\right)\sigma\left(\frac{b-x}{\tau}\right).$$

This relaxation applies to reward terms that encourage velocity, height, heading, or distance to stay within a desired range.

Boolean conjunction and disjunction For hard logical operators, we use differentiable fuzzy-logic relaxations. If $g_1, g_2 \in [0, 1]$ are soft truth values, then

$$g_1 \wedge g_2 \quad \Longrightarrow \quad g_1 g_2,$$

and

$$g_1 \vee g_2 \quad \Longrightarrow \quad 1 - (1 - g_1)(1 - g_2).$$

Negation is relaxed as

$$\neg g \quad \Longrightarrow \quad 1 - g.$$

These operations allow compound logical conditions, such as "within the lane and below the speed limit," to remain differentiable.

Maximum and minimum Hard maximum and minimum operations can introduce nondifferentiable points. We relax them using the log-sum-exp operator:

$$\max_i x_i \quad \Longrightarrow \quad \tau \log \sum_i \exp(x_i/\tau),$$

and

$$\min_i x_i \quad \Longrightarrow \quad -\tau \log \sum_i \exp(-x_i/\tau).$$

Clipping A hard clipping operation

$$\mathrm{clip}(x, l, u) = \min(\max(x, l), u)$$

can be approximated by a difference of softplus functions:

$$\mathrm{clip}(x, l, u) \implies l + \tau \log\left(1 + e^{(x-l)/\tau}\right) - \tau \log\left(1 + e^{(x-u)/\tau}\right), \qquad l < u.$$

This avoids zero gradients outside the clipping interval and is useful for saturating reward terms.

Absolute value The absolute value $|x|$ is nondifferentiable at zero. We use the smooth approximation

$$|x| \quad \Longrightarrow \quad \sqrt{x^2 + \epsilon},$$

where $\epsilon > 0$ is a small constant. This approximation applies to penalties for distance, deviation, lateral error, and action magnitude.

Sign function We approximate the sign function with a hyperbolic tangent:

$$\mathrm{sign}(x) \quad \Longrightarrow \quad \tanh\left(\frac{x}{\tau}\right).$$

This relaxation is useful when the reward depends on the sign of a quantity and gradients are needed near the decision boundary.

Argmax A hard selection

$$i^* \in \arg\max_i z_i, \qquad r = x_{i^*}$$

is relaxed using a softmax-weighted average:

$$\alpha_i = \frac{\exp(z_i/\tau)}{\sum_j \exp(z_j/\tau)}, \qquad r \quad \Longrightarrow \quad \sum_i \alpha_i x_i.$$

This is useful when a reward program selects the nearest object, the most dangerous vehicle, the constraint with the largest violation, or the best-matching target.

Hard termination A reward function may assign a large penalty when the agent violates a health condition, as in:

$$r(x) = \begin{cases} r_{\mathrm{alive}}, & h_{\min} < h < h_{\max}, \\ -r_{\mathrm{fail}}, & \text{otherwise.} \end{cases}$$

Table 3: Additional MuJoCo results (mean ± standard deviation over five runs).

Method	Ant	HalfCheetah	Hopper	Humanoid
Feature-based IRL	4095.5 ± 385.4	10717.6 ± 823.5	3055.9 ± 264.8	6264.5 ± 521.7
DiReCL-Rule	4232.9 ± 441.3	13279.3 ± 1667.0	3137.5 ± 378.2	7073.6 ± 347.9
EUREKA	4643.8 ± 502.3	10215.6 ± 727.9	3153.4 ± 149.9	6401.6 ± 473.4
EUREKA+Demo	5030.0 ± 439.6	12647.8 ± 814.2	3025.4 ± 196.8	5919.3 ± 337.6
DiReCL w/o R	4629.8 ± 879.5	11790.6 ± 2049.1	2954.6 ± 604.0	7035.2 ± 472.9
DiReCL	**5908.1 ± 153.2**	**15005.1 ± 1497.6**	**3700.0 ± 219.1**	**7913.0 ± 231.1**

The relaxed health condition is

$$g_{\text{healthy}} = \sigma\left(\frac{h - h_{\min}}{\tau}\right)\sigma\left(\frac{h_{\max} - h}{\tau}\right),$$

and the corresponding reward is given by

$$r(x) \quad \Longrightarrow \quad g_{\text{healthy}}\, r_{\text{alive}} + (1 - g_{\text{healthy}})(-r_{\text{fail}}).$$

This preserves the intended semantics while providing informative gradients near the failure boundary.

Implementation principle We do not relax every numeric literal in the reward program. Numeric constants whose variation preserves the reward template, such as weights, margins, thresholds, and temperatures, can be converted into learnable parameters. In contrast, structural constants such as tensor indices, array dimensions, and exponents are kept fixed, because varying them can change the semantic structure of the reward program. Thus, DiReCL relaxes hard reward logic only when the relaxation preserves the intended meaning of the generated reward template.

B Experiment Details

B.1 Code Implementation

For AIRL [9], we use the implementation and training configurations from the public PIRO repository [11]. For ML-IRL [10], we use the authors' open-source implementation and the corresponding training configurations. For EUREKA [16], we adapt the public implementation to the environments and RL backbones, using forward velocity in MuJoCo and goal deviation in CommonRoad-RL as the respective fitness measures. EUREKA+Demo instead ranks candidates by the negative imitation gap. All PPO and SAC implementations and configurations follow Stable-Baselines3 [42].

B.2 Compute Resources

We run all experiments on a Linux server equipped with four NVIDIA A100 GPUs. For each MuJoCo task, DiReCL training takes approximately 24 hours on a single A100 GPU. For the highD and inD tasks, DiReCL training takes approximately 60 hours on a single A100 GPU.

B.3 Reward Structure, Reflection, and Supervision Controls

The additional baselines provide stricter controls for reward structure, reflection, and supervision.

Feature-based IRL This IRL baseline uses a manually specified feature library and the same demonstrations. It learns feature weights and internal numerical parameters but does not generate new structure. This serves as a control for the chosen feature library rather than an upper bound on hand-designed reward families.

DiReCL-Rule This DiReCL variant replaces LLM-based reflection with deterministic diagnostic rules that activate, remove, or replace terms from five candidate components: forward progress, control cost, uprightness, height stability, and action smoothness. It uses the same demonstrations, IRL calibration, RL backbone, and interaction budget as DiReCL.

EUREKA+Demo This EUREKA variant retains EUREKA's code-generation and reflection procedure without differentiable numerical calibration, but ranks candidates by the negative imitation gap

computed from the same demonstrations used by DiReCL. This reduces the difference in supervision relative to ordinary EUREKA, which uses forward velocity or goal deviation as its fitness measure.

As shown in Table 3, DiReCL achieves the highest performance across all four MuJoCo tasks.

B.4 Differentiable Relaxation

Table 4 shows the effect of varying the smoothing temperature τ. Smaller temperatures can produce steep gradients near the boundary, whereas larger temperatures can alter the effective condition.

Table 4: HalfCheetah smoothing-temperature sensitivity (mean $\pm$ standard deviation over five runs).

Temperature	HalfCheetah Return
0.01	10898.4 $\pm$ 1894.9
0.05	14909.1 $\pm$ 1796.8
0.10 (default)	15005.1 $\pm$ 1497.6
0.20	9812.2 $\pm$ 1423.0
0.50	9056.8 $\pm$ 1912.3

B.5 Dependence on the LLM Backbone

On HalfCheetah, we compare GPT-5.5-high [43] and Qwen3.5 [44] with GLM-5.1 while holding the prompt, reflection budget, demonstrations, and RL backbone fixed (Table 5).

Table 5: HalfCheetah results using different LLMs (mean $\pm$ standard deviation over five runs).

Method	GLM-5.1	GPT-5.5-high	Qwen3.5
LLM	9203.9 $\pm$ 628.3	9120.8 $\pm$ 522.2	6394.9 $\pm$ 811.2
EUREKA	10215.6 $\pm$ 727.9	11210.3 $\pm$ 633.6	8176.8 $\pm$ 975.2
DiReCL w/o R	11790.6 $\pm$ 2049.1	11716.7 $\pm$ 1611.1	9868.0 $\pm$ 1846.9
DiReCL	15005.1 $\pm$ 1497.6	15836.4 $\pm$ 1914.8	13953.8 $\pm$ 1764.6

B.6 Sensitivity to Demonstration Count

Table 6 compares DiReCL and ML-IRL using the same subsets of HalfCheetah demonstrations.

Table 6: HalfCheetah demonstration-count sensitivity (mean $\pm$ standard deviation over five runs).

Method	1 trajectory	5 trajectories	10 trajectories	25 trajectories
ML-IRL	7320.9 $\pm$ 1191.5	7744.9 $\pm$ 890.6	11013.4 $\pm$ 557.1	12083.2 $\pm$ 264.6
DiReCL	9115.6 $\pm$ 2068.7	13137.6 $\pm$ 1734.1	13825.5 $\pm$ 1592.7	15005.1 $\pm$ 1497.6

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state the main contribution as learning numerical parameters in differentiable reward-code space from demonstrations, while preserving explicit code-level reward structure.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The paper discusses limitations including the absence of VLMs from the current framework and the current focus on demonstration-based supervision.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper primarily proposes an algorithmic framework and empirical evaluation rather than formal theoretical guarantees.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: The paper describes the information needed to reproduce the main experimental results. Additional implementation and hyperparameter details are provided in the appendix and supplemental material.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We provide anonymized code and configuration files in the supplemental material. The experiments use publicly available benchmark environments and datasets, and the supplemental material provides scripts to process them.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: The paper specifies the benchmark environments, training and evaluation protocol, baselines, policy optimizer, reward-parameter optimizer, use of demonstrations, number of seeds, evaluation frequency, and major hyperparameters.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: The experimental results are reported across multiple random seeds. Curves and tables report the mean performance with variability across seeds.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The appendix reports the compute resources used for the main experiments, including the type of GPUs and approximate runtime.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We comply with the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: The paper discusses potential positive impacts such as improving interpretability of learned rewards. It also discusses possible negative impacts, including the risks of reward misspecification.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not release high-risk pretrained generative models, scraped datasets, or systems intended for direct real-world deployment.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The paper cites the benchmark environments, datasets, and baseline implementations used in the experiments, and respects their terms of use.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.

- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The paper introduces research code documented in the supplementary material.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing experiments or research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing, user studies, or human-subject research, and therefore IRB approval or equivalent review is not applicable.

    Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are used as part of the proposed framework to generate and revise programmatic reward-code templates. The paper describes this usage as a core methodological component, and all final claims, results, and analyses are verified by the authors.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.