DiPRL: Learning Discrete Programmatic Policies via Architecture Entropy Regularization

Chengpeng Hu[1] **Yingqian Zhang**[1] **Hendrik Baier**[1,2]
[1]Eindhoven University of Technology, Eindhoven, the Netherlands
[2]Centrum Wiskunde & Informatica, Amsterdam, the Netherlands

Abstract

Programmatic reinforcement learning (PRL) offers an interpretable alternative to deep reinforcement learning by representing policies as human-readable and -editable programs. While gradient-based methods have been developed to optimize continuous relaxations of programs, they face a significant performance drop when converting the continuous relaxations back into discrete programs. Post-hoc discretization can discard optimized branches and parameters in a program, which results in a collapse of policy expressivity and lowered task performance, leading in turn to a need for additional fine-tuning. To overcome these limitations, we propose Differentiable Discrete Programmatic Reinforcement Learning (DiPRL), a method that learns programmatic policies that become nearly discrete during training, avoiding a separate post-hoc fine-tuning stage. We first analyze the inherent risks of performance drop introduced by post-hoc discretization of gradient-based methods. Then, we introduce programmatic architecture entropy regularization, which enables smooth, differentiable training that encourages convergence toward a discrete program. DiPRL maintains the efficiency of gradient-based optimization while mitigating the risks of post-hoc discretization. Our experiments across multiple discrete and continuous RL tasks demonstrate that DiPRL can achieve strong performance via interpretable programmatic policies.

1 Introduction

Programmatic reinforcement learning (PRL) represents policies as human-readable programs, using explicit control flow (e.g., if-else) to choose primitive actions (e.g., "left" and "right" in CartPole-v1). PRL has achieved performance comparable to the neural policies produced by deep reinforcement learning [1] across domains such as games [2–6] and continuous control [7–9], while offering interpretability. Fig. 1 illustrates an example of a simple program to control CartPole-v1 with only one "if-else" statement.

A common approach to learning programmatic policies is local search (LS), which initializes a random program and iteratively modifies it using the production rules of a language grammar [5]. However, LS is inefficient because transitions collected during program evaluations are discarded. LS also requires additional parameter optimizers such as Bayesian Optimization (BO) for domains like continuous control, where program conditions defined on state features have parameters that must be learned [7]. To alleviate these problems, π-PRL [9] relaxes a discrete program into a continuous, differentiable derivation tree, which allows the use of collected transitions for policy gradient training of both program architecture and parameters. This tree assigns probabilities to program branches and actions, and aggregates all possible execution paths to compute an action distribution. After policy gradient training, a discrete resulting program architecture has to be sampled from the tree via maximum likelihood, which then requires further fine-tuning.

Preprint.

However, a significant issue of the differentiable derivation tree is the risk of performance drop during post-hoc discretization. Converting a continuous derivation tree into a discrete program inevitably discards some program branches, which can potentially result in the loss of important parameters and branches that contribute to the performance of the policy. In extreme cases, the derivation tree can collapse into a trivial program architecture that severely limits its expressivity. In this case, even additional fine-tuning may not improve the extracted program any further. We present this phenomenon as a motivating example for our work in Section 4, where a program with fine-tuning never recovers its performance after the discretization in the continuous task, Ant RandomGoal.

Our main contributions are as follows:

(1) We show that post-hoc discretization of continuous derivation trees can cause significant performance drops by discarding useful program branches, and that additional fine-tuning does not always recover the lost performance.

(2) We propose **Differentiable Discrete Programmatic Reinforcement Learning (DiPRL)**, which augments the standard differentiable derivation trees via *program architecture regularization*. DiPRL enables end-to-end learning of programs by encouraging convergence to discrete architectures during training, which eliminates the need for a separate post-discretization fine-tuning stage.

(3) We conduct experiments on discrete and continuous RL tasks to compare DiPRL with baselines. The results show that DiPRL gradually converges to discrete programmatic policies throughout training, allowing it to achieve performance comparable to neural policies while maintaining the interpretability inherent in programmatic representations.

2 Related Work

Programmatic reinforcement learning encodes policies as programs [10, 11]. Fig. 1 presents an example of a programmatic policy for CartPole-v1. While often providing competitive performance to neural policies [9], PRL enables humans to interpret the learned policies, edit them, and verify their correctness [12]. Its performance has been demonstrated in various domains such as games [2–6, 13–15], Karel tasks [16–19], robotics [7–9, 20], and real-world optimization problems [21, 22]. Decision tree (DT) policies can also be expressed as programs, for example by converting a tree into equivalent `if-else` statements. However, DTs are most commonly learned in supervised settings [23–25]. In RL, most DT approaches rely on imitation learning [7, 26], which can suffer from distillation gaps [9]. Recent differential DTs such as [27, 28] have limited performance.

Due to the discrete nature of programs, search-based methods such as local search [5, 14, 4] and Monte Carlo Tree Search (MCTS) [21] are well suited for generating candidate programs by applying production rules that extend or mutate existing programs. In domains where control flow is governed by fixed, discrete predicates (such as "FrontIsClear" in Karel [16]), search over program architecture alone can optimize effectively. In contrast, in continuous-control settings [7], control flow is often determined by parameterized conditions that map continuous features (such as velocities or angles) to boolean decisions.

However, these approaches are limited to gradient-free optimization due to the discreteness of programs, and they often fail to fully leverage collected experience, leading to poor sample efficiency. To address these limitations, π-PRL [9] relaxes discrete programs into continuous, differentiable derivation trees, by assigning probabilities to different program branches. This enables policy

```
if (−0.16 − 0.31 · f0 − 2.1 · f1 − 6.6 · f2 − 2.3 · f3 > 0) :
    Left
else
    Right
```

Figure 1: A programmatic policy discovered by DiPRL for CartPole-v1. f_0, f_1, f_2, f_3 denote symbolic features corresponding to cart position, cart velocity, pole angle, and pole angular velocity, respectively. The program splits the state space into two regions and selects either the "left" or "right" action. It achieves the maximum score of 500 on CartPole-v1.

gradient methods to jointly optimize program architecture and predicate parameters. π-PRL has been extended by replacing its linear conditions with recurrent neural networks, improving generalization across tasks [20]. During training, this approach samples different tasks in each epoch and updates program-architecture weights and parameters sequentially.

A key challenge of using differentiable program relaxations is how to recover a final discrete policy. Post-hoc discretization can discard useful branches and parameters and lead to significant performance degradation, even if followed by additional fine-tuning. We present an example of such performance collapse in Sec. 4, highlighting the motivation for gradual discretization during training as opposed to post-hoc discretization after training.

3 Background

Domain-specific language. Following prior work [9], we represent programmatic policies using the context-free domain-specific language (DSL) shown in Figure 2. The DSL specifies the syntactic space of programmatic policies by defining how programs are composed from three core components: *perceptions* (state-dependent predicates), *control flow* (branching), and *actions* (terminal decisions). A policy program E is either a primitive action A or a conditional statement of the form *if* B *then* A *else* E, where the condition B is defined by a state predicate $\phi_{\boldsymbol{w}}(s) > 0$, parameterized by $\boldsymbol{w}$. We use linear functions of state features for $\phi_{\boldsymbol{w}}(s)$. Conditions determine choices between program branches, such as following the `if` or `else` branch. Given the DSL, a programmatic policy is constructed by starting from an initial (incomplete) program E and iteratively expanding non-terminal nodes. This expansion process continues until all non-terminal nodes have been expanded, at which point the resulting program architecture is specified but still leaves predicates $\boldsymbol{w}$ to be chosen. We define a maximal program depth, at which we enforce the expansion of actions (terminal nodes) instead of additional non-terminals.

$$\text{Program } E := A \mid \text{ if } B \text{ then } A \text{ else } E$$
$$\text{Condition } B := \phi_{\boldsymbol{w}}(s) > 0$$

Figure 2: Domain-specific language (DSL) for programmatic policies. A denotes a terminal action. $\phi_{\boldsymbol{w}}(s)$ is a parameterized predicate, specifically a linear function of state features.

Markov decision processes. A Markov decision process (MDP) [29] is defined as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$, where $\mathcal{S}$ is the set of states (e.g., velocity and angle in CartPole-v1), $\mathcal{A}$ is the set of actions (e.g., left and right actions), $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ is the reward function, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$ is the transition probability function, and γ is the discount factor. A policy π is a mapping from states to probability distributions over actions, where $\pi(a_t|s_t)$ is the probability of taking action a_t in state s_t. The goal is to optimize a parameterized policy π_θ that maximizes the discounted cumulative reward, formulated as: $\max_\theta \ J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t, s_{t+1})]$, where s_0 is an initial state, and $\tau \sim \pi_\theta$ denotes a trajectory $(s_0, a_0, s_1, a_1, \ldots, s_t, a_t, s_{t+1})$.

4 Motivation: Failure mode of the post-hoc discretization

We present a failure mode of the post-hoc discretization in π-PRL, as shown in Fig. 3. Both π-PRL and DiPRL perform similarly before 50% training progress. However, after applying post-hoc discretization (following the original setting of π-PRL), its performance suddenly drops. Even after another 50% of training, its performance is never recovered. The reason for this failure comes from the sudden step from the derivation tree that is trained via policy gradient, a parameterized distribution over multiple program architectures, to the eventually required output of a single discrete program. The post-hoc discretization used by π-PRL extracts this program using maximum likelihood among all possible programs, thus losing all learned contributions from other programs in the derivation tree. The extracted program loses too many program branches,

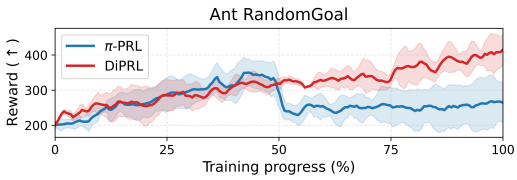

Figure 3: Motivating example of the performance drop caused by post-hoc discretization.

which affects the expressivity of the policy. Even further fine-tuning therefore cannot recover the optimal performance.

5 Differentiable Discrete Programmatic RL

We motivate DiPRL by showing that applying post-hoc discretization to relaxed programmatic policies can lead to an unrecoverable performance drop even with fine-tuning (as presented in Fig. 3). DiPRL mitigates this issue by introducing *program architecture regularization* to π-PRL [9], a continuous differentiable derivation tree. As shown in Figure 4, the tree is built by recursively expanding non-terminal nodes according to the DSL in Figure 2. It represents a distribution over possible programs, so the final policy combines contributions from multiple program architectures. To encourage the relaxed tree to become discrete during training, DiPRL adds program architecture entropy as a regularizer and anneals it with automatic coefficient tuning, eliminating the need for a post-hoc training stage.

5.1 Relaxation into differentiable derivation tree

We follow the same procedure of π-PRL [9] to relax the program into a differential derivation tree. We denote the programmatic policy relaxed by the differentiable derivation tree as $\pi^{\mathcal{T}}_{\mathcal{W}}$, where $\mathcal{T}$ is the relaxed derivation tree used and $\mathcal{W}$ is the set of predicate and action parameters. The tree $\mathcal{T} = \{\mathcal{V}, \mathcal{E}\}$ consists of a node set $\mathcal{V}$ and an edge set $\mathcal{E}$. A node $v \in \mathcal{V}$ can be either a non-terminal, e.g., an `if-else` node, or a terminal action node. An edge $\langle v_i, v_j \rangle \in \mathcal{E}$ connects two nodes, and the expansion probability $P_{\mathcal{T}}(v_j \mid v_i)$ specifies how a non-terminal is expanded by the DSL. These expansion probabilities are state independent and induce a distribution over program paths.

Let $\mathcal{P}$ be the set of program paths in the derivation tree. A program path $\zeta = \langle v_0, v_1, \cdots, v_n \rangle$ starts from the root and follows expansion decisions to a terminal program form. For each consecutive pair in the path, v_{k+1} is a successor of v_k. The state-independent probability of ζ is the product of all expansions along the path: $P_{\mathcal{T}}(\zeta) = \prod_{k=0}^{n-1} P_{\mathcal{T}}(v_{k+1} \mid v_k)$.

Predicate choices along a program path are state dependent. At an `if-else` node with predicate $\phi_{\boldsymbol{w}}(s_t)$, we relax the hard branch indicator $\mathbb{1}[\phi_{\boldsymbol{w}}(s_t) > 0]$ to $\sigma(\phi_{\boldsymbol{w}}(s_t))$, where $\sigma(x) = 1/(1+e^{-x})$. For a predicate-branch edge $v_i \to v_j$, let $b_{ij} = 1$ for the `if` branch and $b_{ij} = 0$ for the `else` branch. Its relaxed predicate-gating probability is $p_{ij}(s_t) = \sigma(\phi_{\boldsymbol{w}}(s_t))^{b_{ij}}(1 - \sigma(\phi_{\boldsymbol{w}}(s_t)))^{1-b_{ij}}$. Note that this probability is internal to an `if-else` node and determines whether `if` or `else` is taken, depending on the state. It is distinct from the program expansion probability $\Pr(v_j^i \mid v_i)$, which determines how a non-terminal is expanded by the DSL (e.g., into an action node or another non-terminal node).

To enable policy gradient, we parameterize each action node with a parameterized action policy $\pi_{\boldsymbol{w}_\zeta}(a_t \mid s_t)$, specifically a categorical distribution over discrete actions or a Gaussian distribution for continuous actions. The state-dependent action distribution contributed by program path ζ is

$$\pi_{\zeta,\mathcal{W}}(a_t \mid s_t) = \left(\prod_{\langle v_i, v_j \rangle \in \zeta} p_{ij}(s_t) \right) \pi_{\boldsymbol{w}_\zeta}(a_t \mid s_t). \tag{1}$$

Then, the relaxed derivation tree that considers both state-independent architecture and state-dependent action distribution is denoted as follows:

$$\pi^{\mathcal{T}}_{\mathcal{W}}(a_t \mid s_t) = \sum_{\zeta \in \mathcal{P}} P_{\mathcal{T}}(\zeta) \pi_{\zeta,\mathcal{W}}(a_t \mid s_t). \tag{2}$$

5.2 Program architecture regularization

While the relaxation described in Sec. 5.1 enables differentiability, it does not specify how to return a discrete policy after training. As discussed in Sec. 4, a significant performance drop may occur if the learned program remains a derivation tree and is discretized post-hoc, e.g., if a discrete program is extracted by selecting the most likely outgoing edge at each node based on $\max_{v_j} \Pr(v_j \mid v_i)$. Post-hoc discretization discards trained program branches whose contributions to performance may

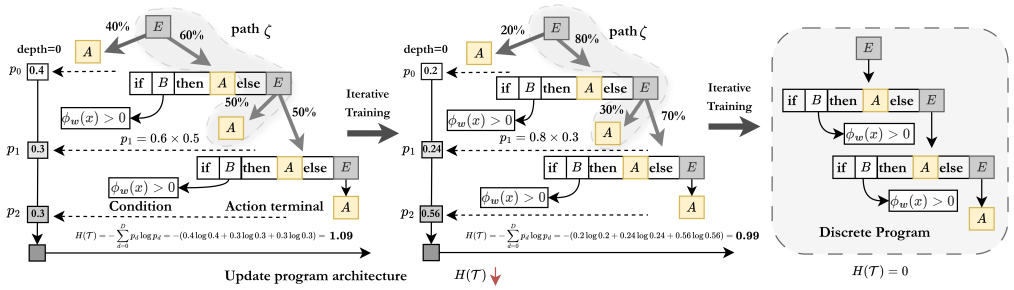

Figure 4: Workflow of DiPRL. E denotes a program node. A denotes an action node, marked in yellow. B denotes a condition in an incomplete program `if` B `then` A `else` E. The maximal depth D_m is set to 2 in the figure. A programmatic policy is relaxed into a continuous derivation program tree by fully expanding and assigning expansion probabilities to program branches. During the training, the program architecture entropy $H(\mathcal{T})$ decreases and encourages the relaxed architecture to approach a discrete program.

not be recoverable even with additional fine-tuning. To avoid this, we propose a novel regularization during training, gradually reducing the *uncertainty* of the program architecture.

Program architecture entropy Recall that the program expansion probabilities $\mathrm{P}_{\mathcal{T}}(v_j \mid v_i)$ determine how each node in the derivation tree is expanded. These probabilities therefore characterize the uncertainty of the program architecture. A program becomes discrete when the expansion probabilities at each node collapse to a one-hot vector, so that only one expansion can be selected.

To quantify this uncertainty, we consider the architecture-dependent probability distribution over program paths, denoted by $\Pr(\zeta)$, where ζ is a complete path in the derivation tree. Let $\mathcal{P}$ denote the set of all possible paths. The generic program architecture entropy is defined as

$$\hat{H}(\mathcal{T}) = -\sum_{\zeta \in \mathcal{P}} \Pr(\zeta) \log \Pr(\zeta). \tag{3}$$

Equivalently, if P is the random variable representing the sampled program path, then $\hat{H}(\mathcal{T}) = H(P)$. We then express this entropy in terms of program depth. Let D be the random variable denoting the depth of a sampled path. Since the depth is determined by the path, we have $D = f(P)$. For a maximum depth D_m, define the probability of terminating at depth d as $p_d \triangleq \Pr(D = d) = \sum_{\zeta \in \mathcal{P}_d} \Pr(\zeta), \sum_{d=1}^{D_m} p_d = 1$, where $\mathcal{P}_d$ is the set of all paths with depth d. The corresponding depth-based architecture entropy is $H(\mathcal{T}) = H(D) = -\sum_{d=1}^{D_m} p_d \log p_d$. In the DSL used in this work, the depth uniquely determines the corresponding program path. Therefore, the conditional entropy term vanishes and the depth-based entropy is not merely a surrogate but exactly matches the generic path entropy, $H(\mathcal{T}) = \hat{H}(\mathcal{T})$. We use this entropy as a regularizer during training to gradually reduce the uncertainty of the program architecture and encourage convergence toward a discrete program.

Regularized policy gradient High entropy indicates a larger uncertainty in the program architecture, while low entropy indicates high confidence in a particular architecture. By introducing this architecture regularization, we reformulate the MDP that maximizes the returns while converging to a discrete program as follows:

$$\max_{\mathcal{T}, \mathcal{W}} \; J(\mathcal{T}, \mathcal{W}) \quad \text{s.t.} \quad H(\mathcal{T}) \leq \bar{H}, \tag{4}$$

where $\bar{H}$ is the target architecture entropy. A fully discrete architecture would ideally have zero entropy, but setting $\bar{H} = 0$ can force premature convergence to a poor program before useful branches have been explored. We therefore use a small positive target entropy and discuss its choice in Appendix D. Using a Lagrangian relaxation, we relax the objective:

$$L(\mathcal{T}, \mathcal{W}, \alpha) = \mathbb{E}_{\tau} \left[\sum_{t=0}^{T} \log \pi_{\mathcal{W}}^{\mathcal{T}}(a_t \mid s_t) \hat{A}_t \right] - \alpha(H(\mathcal{T}) - \bar{H}), \tag{5}$$

where $\alpha \geq 0$, $\tau \sim \pi_{\{\mathcal{T},\mathcal{W}\}}$ and $\hat{A}_t$ denotes the advantage estimate that measures how good a_t is in state s_t, following the policy gradient [30]. We discuss strength of α in Appendix E.

Our regularization is different from entropy-based reinforcement learning methods such as Soft Actor-Critic (SAC) [31, 32]. SAC encourages exploration by maximizing the entropy of the action distribution. However, our regularization operates directly on the program architecture, which aims at shaping the distribution over the derivation tree rather than over actions.

5.3 Why post-hoc discretization can fail

We provide a theoretical analysis (detailed in Appendix B) for the failure mode of post-hoc program extraction. Let $\tilde{\mathcal{P}} \subseteq \mathcal{P}$ be the program paths retained after extracting the discrete program $\tilde{\mathcal{T}}$, and let $\tilde{\pi}$ denote the policy induced by this extracted program. For a program path ζ, define the path-induced value:

$$Q^{\pi}_{\zeta}(s) = \sum_{a \in \mathcal{A}} \pi_{\zeta,\mathcal{W}}(a \mid s) Q^{\pi}(s,a), \tag{6}$$

and let$m_{\mathcal{T}} = \sum_{\zeta \notin \tilde{\mathcal{P}}} P_{\mathcal{T}}(\zeta)$ be the deleted program-path mass. The performance difference between π and $\tilde{\pi}$ is

$$J(\tilde{\pi}) - J(\pi) = \frac{1}{1-\gamma} \mathbb{E}_{s \sim \tau^{\tilde{\pi}}} \left[\sum_{\zeta \in \tilde{\mathcal{P}}} \left(P_{\tilde{\mathcal{T}}}(\zeta) - P_{\mathcal{T}}(\zeta) \right) Q^{\pi}_{\zeta}(s) - \sum_{\zeta \notin \tilde{\mathcal{P}}} P_{\mathcal{T}}(\zeta) Q^{\pi}_{\zeta}(s) \right]. \tag{7}$$

Let $Q_{\mathrm{del}}(s)$ be the average $Q^{\pi}_{\zeta}(s)$ value of deleted program paths, and let $Q_{\mathrm{keep}}(s)$ be the average value of retained program paths receiving the removed mass. Then

$$J(\tilde{\pi}) - J(\pi) = \frac{1}{1-\gamma} \mathbb{E}_{s \sim \tau^{\tilde{\pi}}} \left[m_{\mathcal{T}} \left(Q_{\mathrm{keep}}(s) - Q_{\mathrm{del}}(s) \right) \right]. \tag{8}$$

Suppose there exists a $\kappa(s) > 0$, such that $Q_{\mathrm{del}}(s) - Q_{\mathrm{keep}}(s) \geq \kappa(s)$, then

$$J(\tilde{\pi}) - J(\pi) \leq -\frac{1}{1-\gamma} \mathbb{E}_{s \sim \tau^{\tilde{\pi}}} [m_{\mathcal{T}} \kappa(s)]. \tag{9}$$

Thus, the discretization drop is affected by the amount of deleted program-path mass $m_{\mathcal{T}}$ and the value gap between deleted and retained program paths. DiPRL reduces this failure mode by making the program-path distribution concentrate during training.

5.4 Why architecture entropy regularization helps

The above section shows that post-hoc extraction is harmful when it removes probability mass from useful program paths. Architecture entropy regularization addresses this term directly. Let $p_{\max} = \max_{\zeta \in \mathcal{P}} P_{\mathcal{T}}(\zeta)$ be the probability of the program path selected during discretization, and let $m_{\mathcal{T}} = 1 - p_{\max}$ be the total path probability mass assigned to deleted paths. A low-entropy program-path distribution must concentrate on at least one path. Since Shannon entropy upper-bounds minimal entropy, $H(\mathcal{T}) \geq -\log p_{\max}$, and therefore $p_{\max} \geq \exp(-H(\mathcal{T}))$. Thus

$$m_{\mathcal{T}} = 1 - p_{\max} \leq 1 - \exp(-H(\mathcal{T})) \leq H(\mathcal{T}). \tag{10}$$

If the entropy regularizer drives $H(\mathcal{T}) \leq \epsilon, \epsilon > 0$ before discretization, then the mass that can be discarded by selecting the maximum-probability path is at most ϵ. Let Δ be a uniform upper bound on the retained-versus-deleted path-value gap, i.e., $|Q_{\mathrm{keep}}(s) - Q_{\mathrm{del}}(s)| \leq \Delta$, then

$$|J(\tilde{\pi}) - J(\pi)| \leq \frac{\Delta}{1-\gamma} m_{\mathcal{T}} \leq \frac{\Delta}{1-\gamma} \epsilon. \tag{11}$$

Architecture entropy regularization does not assume that deleted paths have low value, instead, it reduces the total amount of program-path mass that can be deleted. When $H(\mathcal{T})$ is small, the relaxed derivation tree is already close to a single discrete program path in architecture space, so the policy evaluated after extraction is closer to the relaxed policy optimized during training.

Table 1: Rewards (mean $\pm$ std) on discrete benchmark tasks across three runs; higher is better. $\pi_{\text{cont.}}$-PRL, $\pi_{\text{disc.}}$-PRL, and π-PRL denote the relaxed policy before discretization, the discretized policy before fine-tuning, and the final fine-tuned policy, respectively. The best performance among policies (DTSemNets, π-PRL and DiPRL) without oracle is bold for each tasks.

Algorithm	CartPole-v1	Acrobot-v1	MountainCar-v0	DoorKey	LunarLander-V2
PPO	500.00 ± 0.00	-62.57 ± 0.56	-98.57 ± 0.23	0.97 ± 0.00	283.11 ± 1.14
VIPER (PPO)	500.00 ± 0.00	-67.83 ± 3.63	-97.13 ± 2.08	0.55 ± 0.17	164.33 ± 54.25
DTSemNets	326.76 ± 119.77	-78.98 ± 0.68	-199.99 ± 0.07	0.00 ± 0.00	257.61 ± 8.92
$\pi_{\text{cont.}}$-PRL	$\mathbf{500.00 \pm 0.00}$	-80.31 ± 2.12	-172.50 ± 38.89	0.63 ± 0.44	235.72 ± 19.50
$\pi_{\text{disc.}}$-PRL	$\mathbf{500.00 \pm 0.00}$	-86.63 ± 2.67	-178.58 ± 30.30	0.32 ± 0.32	239.90 ± 30.28
π-PRL	$\mathbf{500.00 \pm 0.00}$	-89.37 ± 4.83	-171.83 ± 39.83	0.66 ± 0.39	257.21 ± 4.63
DiPRL (ours)	$\mathbf{500.00 \pm 0.00}$	-79.93 ± 3.37	$\mathbf{-110.79 \pm 4.07}$	$\mathbf{0.95 \pm 0.01}$	$\mathbf{260.21 \pm 13.61}$

6 Experiments

Domains We evaluate DiPRL and baselines on domains with discrete tasks, namely classic control tasks [33] including CartPole-v1, Acrobot-v1, and MountainCar-v0, as well as DoorKey and LunarLander-v2. We also compare algorithms on continuous tasks [9] like HalfCheetah Hurdle, Pusher2D, Ant RandomGoal and Ant CrossMaze.

Baselines We compare DiPRL with PPO [30] as a neural policy baseline, VIPER [34], and DTSemNets [28] as decision tree policy baselines, and π-PRL [9] as a differentiable programmatic policy baseline. In addition, we add $\pi_{\text{cont.}}$-PRL, which denotes the program derivation tree trained by π-PRL before the post-hoc training stage. $\pi_{\text{disc.}}$-PRL denotes the extracted discrete program from $\pi_{\text{cont.}}$-PRL. π-PRL is the final programmatic policy after fine-tuning.

VIPER extracts a decision tree policy based on the neural policy trained by PPO, while DTSemNets directly trains a differentiable decision tree policy without an oracle. π-PRL [9] does not need an oracle, but requires post-hoc discretization and fine-tuning. The post-hoc discretization keeps the dominant branches according to the learned architecture distribution and replaces the relaxed sigmoid with a hard threshold. DiPRL does not need fine-tuning, and the final discretization is used only for evaluating the discrete program. The maximal depth of DT [7] and programs is set to 6, as suggested in [9]. All algorithms are trained with the same budget on each task across three seeds. Implementation and experiment configuration are detailed in appendix C.

6.1 Discrete tasks

Table 1 summarizes the averaged rewards of DiPRL and the baselines on discrete tasks. We include PPO as a neural-policy reference, since it achieves strong performance across all environments. VIPER extracts decision-tree policies from a trained neural policy oracle; in our experiments, we use the PPO agent as this oracle. Although VIPER matches PPO on CartPole-v1 and performs well on Acrobot-v1 and MountainCar-v0, its performance is less stable on more challenging tasks, reflecting the distillation gap commonly observed in imitation-based policy extraction [9]. In particular, VIPER achieves only 0.55 on DoorKey and 164.33 on LunarLander-v2, despite access to PPO policies that obtain 0.97 and 283.11, respectively.

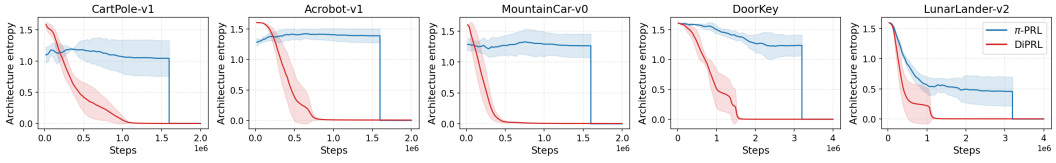

Figure 5: Architecture entropy curves in discrete tasks. The architecture entropy of π-PRL is forced to zero only via post-hoc discretization. DiPRL discretizes the derivation tree relaxation over time.

DTSemNets uses a specialized policy architecture that is semantically equivalent to an oblique decision tree. However, its architecture must be manually specified during training. We use the default policy setting from Panda et al. [28]. As shown in Table 1, DTSemNets performs competitively on Acrobot-v1 and LunarLander-v2, but it underperforms on CartPole-v1, MountainCar-v0, and DoorKey. For example, it obtains only 326.76 on CartPole-v1, collapses to nearly the minimum return on MountainCar-v0 with -200.19 , and fails to solve DoorKey, achieving 0.00.

The most closely related method to DiPRL is π-PRL, which also optimizes a continuous relaxation of a programmatic policy. However, the post-hoc discretization stage required by π-PRL can destabilize the final discrete policy by discarding useful learned branches. This effect is visible on DoorKey, where the continuous relaxed policy $\pi_{\text{cont.}}$-PRL achieves 0.63, but the discretized $\pi_{\text{disc.}}$-PRL policy drops to 0.32. DiPRL mitigates this issue by driving the relaxed architecture close to a discrete program during training, achieving 0.95 on DoorKey. We present the architecture entropy curve in Fig. 5, where π-PRL still has a large entropy before the post-hoc discretization. Fig. 6 also shows that DiPRL maintains very small entropy, e.g., 0.004, whereas $\pi_{\text{cont.}}$-PRL still has a large entropy > 0.25 before the post-hoc discretization.

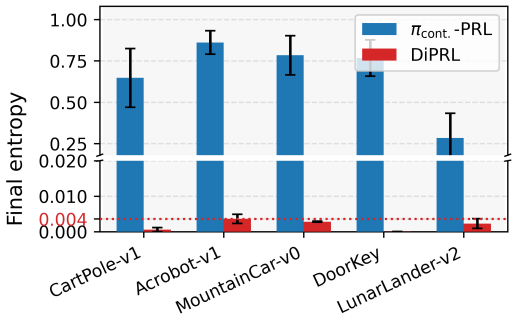

Figure 6: Normalized architecture entropy before discretization on discrete tasks. Lower values indicate architectures closer to discrete programs.

This result aligns with our theoretical analysis. π-PRL keeps a large $m_{\mathcal{T}}$, which aggravates the instability of the policy and may lead to a performance drop. Our DiPRL encourages gradual discretization over the course of the training, where DiPRL's architecture entropy eventually converges to zero, i.e., reducing $m_{\mathcal{T}}$. DiPRL allows alternative program branches to be gradually pruned during training, without requiring a post-hoc training stage. As a result, DiPRL avoids the significant performance drop typically observed in π-PRL, which even additional fine-tuning cannot always alleviate.

6.2 Continuous tasks

For the continuous tasks introduced by [9], we report both episodic reward and goal distance; higher reward and lower goal distance indicate better performance. Table 2 summarizes the averaged rewards of DiPRL and the baselines, and Fig. 7 presents the goal-distance curves. These continuous tasks are known to be challenging for vanilla PPO [9]. The relaxed policy $\pi_{\text{cont.}}$-PRL performs well, especially on Ant CrossMaze, where it obtains 363.44 compared with 220.25 for DiPRL. However, π-PRL shows severe performance drop after discretization. On HalfCheetah Hurdle and Ant CrossMaze, its reward drops from 250.36 to -848.84 and from 363.44 to -506.32, respectively. Even with further fine-tuning, π-PRL hardly recovers from this drop, as also shown in Fig. 7. In contrast, DiPRL remains stable and achieves the best final discrete programmatic policy performance.

Table 2: Rewards (mean $\pm$ std) on the continuous tasks across three runs; higher is better.

Algorithm	HalfCheetah Hurdle	Pusher2D	Ant RandomGoal	Ant CrossMaze
PPO	-780.43 ± 758.89	-87.07 ± 2.77	174.28 ± 31.39	-657.03 ± 64.87
VIPER (PPO)	-1107.49 ± 928.25	-87.85 ± 2.79	328.89 ± 13.37	-996.54 ± 55.43
DTSemNets	-4790.10 ± 22.09	-79.13 ± 4.80	313.47 ± 17.84	-1010.04 ± 37.50
$\pi_{\text{cont.}}$-PRL	250.36 ± 478.97	-82.44 ± 2.84	336.56 ± 45.56	$\mathbf{363.44 \pm 180.23}$
$\pi_{\text{disc.}}$-PRL	-848.84 ± 626.73	-102.65 ± 0.66	237.05 ± 24.48	-506.32 ± 105.49
π-PRL	-443.80 ± 915.83	-80.36 ± 1.48	264.75 ± 61.80	-223.58 ± 226.08
DiPRL (ours)	$\mathbf{723.88 \pm 52.00}$	$\mathbf{-78.62 \pm 1.59}$	$\mathbf{413.12 \pm 47.62}$	220.25 ± 75.94

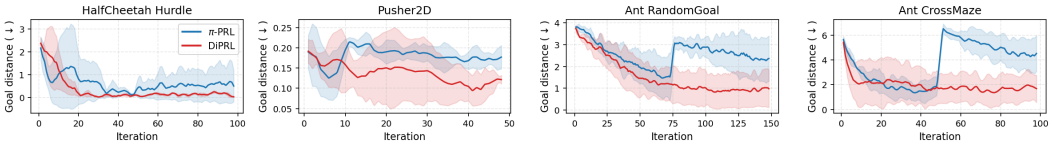

Figure 7: Goal distance curves in continuous tasks in a comparison with π-PRL and DiPRL; smaller is better. π-PRL collapses on all tasks except Pusher2D and its performance hardly recovers.

6.3 Ablation study on regularization strength

There is a trade-off in the choice of α: a small coefficient may leave the final policy too relaxed, while a large coefficient may force premature convergence to a poor program. We evaluate the effect of α by comparing fixed coefficients $\alpha \in \{0, 0.001, 0.01, 0.1, 0.5\}$ with automatic tuning, which adapts α during training. All α settings reach optimal performance in CartPole-v1, as shown in Fig. 8. On harder tasks, fixed coefficients show clear weaknesses. Small coefficients can leave useful behavior spread across multiple program branches, so final discretization removes part of the learned policy. Large coefficients can reduce this discretization gap, but they also push the architecture toward a discrete program too early and limit exploration. As a result, no fixed α is reliable across all tasks. Automatic tuning is more stable because it keeps architecture exploration flexible early in training and increases the pressure toward a discrete program later. Detailed results are reported in Appendix E.

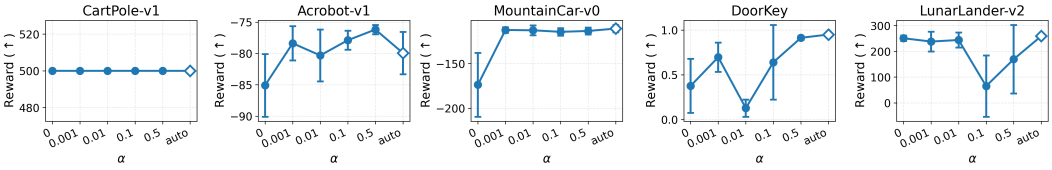

Figure 8: Ablation on regularizer α selected from set $\{0, 0.001, 0.01, 0.1, 0.5\}$ and automatic tuning. Automatic tuning (marked as rhombus ◇) is overall stable.

7 Conclusion

In this work, we first demonstrate a performance drop that can occur when extracting discrete programs post-hoc from a continuous program derivation tree in state-of-the-art PRL approaches. Even with further fine-tuning, it might be unrecoverable. We also theoretically analyze the failure mode. To address the issue, we propose Differentiable Discrete Programmatic Reinforcement Learning (**DiPRL**) by introducing program architecture regularization. DiPRL does not require a separate post-hoc training stage. DiPRL uses gradient-based optimization over both program architecture and parameters. Instead of extracting a discrete program from a continuous relaxation after training, DiPRL integrates a program architecture entropy regularization that enables gradual discretization during training. The differentiable continuous derivation tree gradually converges to a discrete program, thereby mitigating the performance drop caused by the post-hoc discretization. Experiments on discrete and continuous tasks demonstrate the strong performance of DiPRL compared to differentiable decision tree policy and programmatic policy baselines. We also present an ablation study to verify the strength of the regularization.

8 Limitation and future work

Programmatic policies are designed to be human-readable programs. It is important to verify if the derived program by π-PRL and DiPRL indeed improves interpretability. We leave the human-centered evaluation and study as important future work. It is interesting that DiPRL discovers strong policies grounded in a simple DSL with only `if-else` control flow. If an optimal strategy requires constructs this DSL lacks, such as loops and temporal operators for long-horizon tasks, DiPRL may converge to an interpretable but suboptimal program. In future work, we want to extend the DSL with temporal/iterative structure to better capture repetitive behaviors. We will also explore DiPRL on real-world problems such as job-shop scheduling.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[2] Abhinav Verma, Hoang Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[3] Mohammadhosein Hasanbeig, Natasha Yogananda Jeppu, Alessandro Abate, Tom Melham, and Daniel Kroening. Deepsynth: Automata synthesis for automatic task segmentation in deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 7647–7656, 2021.

[4] David S Aleixo and Levi HS Lelis. Show me the way! bilevel search for synthesizing programmatic strategies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 4991–4998, 2023.

[5] Julian RH Marino, Rubens O Moraes, Tassiana C Oliveira, Claudio Toledo, and Levi HS Lelis. Programmatic strategies for real-time strategy games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 381–389, 2021.

[6] Yushi Cao, Zhiming Li, Tianpei Yang, Hao Zhang, Yan Zheng, Yi Li, Jianye Hao, and Yang Liu. Galois: boosting deep reinforcement learning via generalizable logic synthesis. *Advances in Neural Information Processing Systems*, 35:19930–19943, 2022.

[7] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, pages 5045–5054. PMLR, 2018.

[8] Jeevana Priya Inala, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama. Synthesizing programmatic policies that inductively generalize. In *8th International Conference on Learning Representations*, 2020.

[9] Wenjie Qiu and He Zhu. Programmatic reinforcement learning without oracles. In *The Tenth International Conference on Learning Representations*, 2022.

[10] Dongkyu Choi and Pat Langley. Learning teleoreactive logic programs from problem solving. In *International Conference on Inductive Logic Programming*, pages 51–68. Springer, 2005.

[11] Kamal Acharya, Waleed Raza, Carlos Dourado, Alvaro Velasquez, and Houbing Herbert Song. Neurosymbolic reinforcement learning and planning: A survey. *IEEE Transactions on Artificial Intelligence*, 5(5):1939–1953, 2023.

[12] Yuning Wang and He Zhu. Verification-guided programmatic controller synthesis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 229–250. Springer, 2023.

[13] Zahra Bashir, Michael Bowling, and Levi HS Lelis. Assessing the interpretability of programmatic policies with large language models. *arXiv preprint arXiv:2311.06979*, 2023.

[14] Rubens O. Moraes, David S. Aleixo, Lucas N. Ferreira, and Levi H. S. Lelis. Choosing well your opponents: how to guide the synthesis of programmatic strategies. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, 2023. ISBN 978-1-956792-03-4.

[15] Rubens O Moraes and Levi HS Lelis. Searching for programmatic policies in semantic spaces. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, pages 5990–5998, 2024.

[16] Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J Lim. Learning to synthesize programs as interpretable and generalizable policies. *Advances in Neural Information Processing Systems*, 34:25146–25163, 2021.

[17] Guan-Ting Liu, En-Pei Hu, Pu-Jen Cheng, Hung-Yi Lee, and Shao-Hua Sun. Hierarchical programmatic reinforcement learning via learning to compose programs. In *International Conference on Machine Learning*, pages 21672–21697. PMLR, 2023.

[18] Max Liu, Chan-Hung Yu, Wei-Hsu Lee, Cheng-Wei Hung, Yen-Chun Chen, and Shao-Hua Sun. Synthesizing programmatic reinforcement learning policies with large language model guided search. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=8DBTq09LgN`.

[19] Yu-An Lin, Chen-Tao Lee, Chih-Han Yang, Guan-Ting Liu, and Shao-Hua Sun. Hierarchical programmatic option framework. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[20] Tianyi Wu, Liwei Shen, Zhen Dong, Xin Peng, and Wenyun Zhao. Synthesizing programmatic policy for generalization within task domain. In *Thirty-Third International Joint Conference on Artificial Intelligence*, pages 5217–5225, 8 2024.

[21] Yin Gu, Kai Zhang, Qi Liu, Weibo Gao, Longfei Li, and Jun Zhou. π-light: Programmatic interpretable reinforcement learning for resource-limited traffic signal control. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 21107–21115, 2024.

[22] Yin Gu, Kai Zhang, Qi Liu, Runlong Yu, Xin Lin, and Xinjie Sun. Procc: Programmatic reinforcement learning for efficient and transparent TCP congestion control. In *Eighteenth ACM International Conference on Web Search and Data Mining*, pages 963–972, 2025.

[23] Sicco Verwer and Yingqian Zhang. Learning optimal classification trees using a binary linear program formulation. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 1625–1632, 2019.

[24] Daniël Vos and Sicco Verwer. Efficient training of robust decision trees against adversarial examples. In *International Conference on Machine Learning*, pages 10586–10595. PMLR, 2021.

[25] Alvin Wan, Lisa Dunlap, Daniel Ho, Jihan Yin, Scott Lee, Suzanne Petryk, Sarah Adel Bargal, and Joseph E. Gonzalez. NBDT: Neural-backed decision tree. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=mCLVeEpplNE`.

[26] Tom Silver, Kelsey R Allen, Alex K Lew, Leslie Pack Kaelbling, and Josh Tenenbaum. Few-shot bayesian imitation learning with logical program policies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 10251–10258, 2020.

[27] Andrew Silva, Taylor Killian, Ivan Jimenez, Sung-Hyun Son, and Matthew Gombolay. Optimization methods for interpretable differentiable decision trees applied to reinforcement learning. In *International Conference on Artificial Intelligence and Statistics*, pages 1855–1865. PMLR, 2020.

[28] Subrat Prasad Panda, Blaise Genest, Arvind Easwaran, and Ponnuthurai Nagaratnam Suganthan. Vanilla gradient descent for oblique decision trees. In *27th European Conference on Artificial Intelligence*, volume 392, pages 1140–1147, 2024.

[29] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

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

[31] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870. PMLR, 2018.

[32] Ziqi Wang, Chengpeng Hu, Jialin Liu, and Xin Yao. Negatively correlated ensemble reinforcement learning for online diverse game level generation. In *The Twelfth International Conference on Learning Representations*, 2024.

[33] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

[34] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. *Advances in Neural Information Processing Systems*, 31:2499–2509, 2018.

[35] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL `http://jmlr.org/papers/v22/20-1364.html`.

[36] Jiayi Weng, Huayu Chen, Dong Yan, Kaichao You, Alexis Duburcq, Minghao Zhang, Yi Su, Hang Su, and Jun Zhu. Tianshou: A highly modularized deep reinforcement learning library. *Journal of Machine Learning Research*, 23(267):1–6, 2022.

[37] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

A Broad Impact

Our DiPRL extends differential program derivation tree for training an interpretable programmatic policy with mitigating the performance drop of the post-hoc discretizations. A learned programmatic policy helps human users understand why an agent chooses an action in a given state. Besides, it can support policy debugging and failure detection before deployment. However, we note that a short program can still produce undesirable behavior, given biased states, or fail outside the training distribution. Over-trusting a policy is also a potential risk because its syntax is readable, even when its behavior has not been validated. These concerns are especially important if programmatic policies are used in real-world control, scheduling, robotics, or other decision-making systems.

B Theoretical analysis on the failure mode of the post-hoc discretization

We provide a theoretical analysis of the failure mode of post-hoc discretization while using program paths as the basic object. Let $\pi = \pi_{\mathcal{W}}^{\mathcal{T}}$ be the relaxed policy before discretization. Let $\mathcal{P}$ be the set of program paths and let $P_{\mathcal{T}}(\zeta)$ be the state-independent probability of program path $\zeta \in \mathcal{P}$. The policy contribution induced by program path ζ is

$$\pi_{\zeta,\mathcal{W}}(a \mid s) = \left(\prod_{\langle v_i, v_j \rangle \in \zeta} p_{ij}(s) \right) \pi_{\boldsymbol{w}_\zeta}(a \mid s), \tag{12}$$

and the relaxed derivation-tree policy is

$$\pi(a \mid s) = \sum_{\zeta \in \mathcal{P}} P_{\mathcal{T}}(\zeta) \pi_{\zeta,\mathcal{W}}(a \mid s). \tag{13}$$

Let $\tilde{\mathcal{P}} \subseteq \mathcal{P}$ be the program paths retained after extracting the discrete program $\tilde{\mathcal{T}}$. The extracted policy $\tilde{\pi}$ uses the retained architecture distribution $P_{\tilde{\mathcal{T}}}$:

$$\tilde{\pi}(a \mid s) = \sum_{\zeta \in \tilde{\mathcal{P}}} P_{\tilde{\mathcal{T}}}(\zeta) \pi_{\zeta,\mathcal{W}}(a \mid s), \qquad P_{\tilde{\mathcal{T}}}(\zeta) = 0 \text{ for } \zeta \notin \tilde{\mathcal{P}}. \tag{14}$$

For any policy μ, define the discounted visited-state distribution as

$$d^{\mu}(s) = (1-\gamma) \sum_{t=0}^{\infty} \gamma^t \Pr(s_t = s \mid \mu). \tag{15}$$

By the performance-difference identity,

$$J(\tilde{\pi}) - J(\pi) = \frac{1}{1-\gamma} \mathbb{E}_{s \sim \tau^{\tilde{\pi}}} \left[\sum_{a \in \mathcal{A}} \left(\tilde{\pi}(a \mid s) - \pi(a \mid s) \right) Q^{\pi}(s,a) \right]. \tag{16}$$

Define the path-induced expected value under the reference value function Q^{π} as

$$Q_{\zeta}^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi_{\zeta,\mathcal{W}}(a \mid s) Q^{\pi}(s,a). \tag{17}$$

Then,

$$J(\tilde{\pi}) - J(\pi) = \frac{1}{1-\gamma} \mathbb{E}_{s \sim \tau^{\tilde{\pi}}} \left[\sum_{\zeta \in \tilde{\mathcal{P}}} \left(P_{\tilde{\mathcal{T}}}(\zeta) - P_{\mathcal{T}}(\zeta) \right) Q_{\zeta}^{\pi}(s) - \sum_{\zeta \notin \tilde{\mathcal{P}}} P_{\mathcal{T}}(\zeta) Q_{\zeta}^{\pi}(s) \right]. \tag{18}$$

We define the deleted architecture mass as

$$m_{\mathcal{T}} = \sum_{\zeta \notin \tilde{\mathcal{P}}} P_{\mathcal{T}}(\zeta). \tag{19}$$

Since $P_{\mathcal{T}}(\cdot)$ and $P_{\tilde{\mathcal{T}}}(\cdot)$ are both probability distributions over program paths, the total mass removed from deleted paths equals the total extra mass assigned to retained paths:

$$m_{\mathcal{T}} = \sum_{\zeta \in \tilde{\mathcal{P}}} \left(P_{\tilde{\mathcal{T}}}(\zeta) - P_{\mathcal{T}}(\zeta) \right). \tag{20}$$

Let $Q_{\mathrm{del}}(s)$ be the average $Q^{\pi}_{\zeta}(s)$ value of deleted program paths:

$$Q_{\mathrm{del}}(s) = \frac{\sum_{\zeta \notin \tilde{\mathcal{P}}} P_{\mathcal{T}}(\zeta) Q^{\pi}_{\zeta}(s)}{m_{\mathcal{T}}}. \tag{21}$$

Let $Q_{\mathrm{keep}}(s)$ be the average value of retained program paths receiving the extra probability mass:

$$Q_{\mathrm{keep}}(s) = \frac{\sum_{\zeta \in \tilde{\mathcal{P}}} \left(P_{\tilde{\mathcal{T}}}(\zeta) - P_{\mathcal{T}}(\zeta) \right) Q^{\pi}_{\zeta}(s)}{m_{\mathcal{T}}}. \tag{22}$$

Then the performance difference can be rewritten as

$$J(\tilde{\pi}) - J(\pi) = \frac{1}{1-\gamma} \mathbb{E}_{s \sim \tau^{\tilde{\pi}}} \left[m_{\mathcal{T}} \left(Q_{\mathrm{keep}}(s) - Q_{\mathrm{del}}(s) \right) \right]. \tag{23}$$

This shows that post-hoc extraction decreases performance when the deleted program paths are more valuable than the retained program paths receiving their probability mass. More precisely, suppose there exists $\kappa(s) > 0$ such that $Q_{\mathrm{del}}(s) - Q_{\mathrm{keep}}(s) \geq \kappa(s)$. Then

$$J(\tilde{\pi}) - J(\pi) \leq -\frac{1}{1-\gamma} \mathbb{E}_{s \sim \tau^{\tilde{\pi}}} \left[m_{\mathcal{T}} \kappa(s) \right]. \tag{24}$$

Finally, if the extracted program path is the maximum-probability path, $p_{\max} = \max_{\zeta \in \mathcal{P}} P_{\mathcal{T}}(\zeta)$ and $m_{\mathcal{T}} = 1 - p_{\max}$. Since Shannon entropy upper-bounds min-entropy,

$$H(\mathcal{T}) \geq -\log p_{\max}, \qquad m_{\mathcal{T}} \leq 1 - \exp(-H(\mathcal{T})) \leq H(\mathcal{T}). \tag{25}$$

Let ϵ be any upper bound on the achieved entropy before extraction, $H(\mathcal{T}) \leq \epsilon$. It can be the target $\bar{H}$. Let Δ be a uniform bound, $|Q_{\mathrm{keep}}(s) - Q_{\mathrm{del}}(s)| \leq \Delta$ for states under $\tau^{\tilde{\pi}}$. Then

$$|J(\tilde{\pi}) - J(\pi)| \leq \frac{\Delta}{1-\gamma} m_{\mathcal{T}} \leq \frac{\Delta}{1-\gamma} \epsilon. \tag{26}$$

DiPRL targets the post-hoc extraction failure mode by making the architecture distribution concentrate during training, so that the deleted path mass $m_{\mathcal{T}}$ is small before the final discrete program is evaluated.

C Implementation details

This section summarizes the implementation and training settings used for the methods reported in the main tables. All aggregate results use three random seeds, $\{123, 321, 456\}$, and report the mean and standard deviation across seeds under 30 evaluation episodes.

C.1 π-PRL

We train π-PRL with the program derivation relaxation used by the original implementation [9]. During Phase 1, the policy alternates between architecture search and program-parameter optimization. The architecture step updates the distribution over candidate program depths, while the program step updates the predicates and action weights inside the candidate programs.

Post-hoc discretization and fine-tuning in π-PRL. After Phase 1, π-PRL converts the program derivation relaxation into a single discrete program. Concretely, the learned architecture distribution is frozen and the most likely program depth is selected by argmax over the architecture probabilities. The selected program then copies the corresponding learned predicates and action weights from the trained fusion program. We report this extracted policy before additional training as π_{disc}-PRL. Standard π-PRL [9] then performs a Phase-2 fine-tuning stage on this fixed extracted program. The architecture is no longer searched, but the program parameters and value baseline continue to be optimized. Phase 2 uses 20% of the total training budget for discrete tasks. For continuous tasks, we use 50% of the total budget on HalfCheetah Hurdle, Ant RandomGoal, and Ant CrossMaze, and 80% on Pusher2D, following the original settings of π-PRL [9].

C.2 DiPRL

DiPRL uses the same program-derivation-graph parameterization as π-PRL, but introduces architecture entropy regularization. $H(\mathcal{T})$ measures architecture uncertainty, which is determined by the maximal depth D_m, i.e., $H(\mathcal{T}) = H(D) = -\sum_{d=1}^{D_m} p_d \log p_d$. Thus, $\log D_m$ provides the natural scale of the uncertainty. The target entropy is $0.1 \log D_m$. DiPRL is trained as a single phase and skips the post-extraction Phase-2 fine-tuning used by π-PRL. The final policy is the extracted discrete program from this single phase run. DiPRL uses automatic tuning via the dual update for α with a learning rate 10^{-4}. We initialize $\log \alpha = -10$, i.e., $\alpha_{\text{initial}} = \exp(-10)$, and clamp $\log \alpha$ to $[-10, 10]$ during training.

C.3 PPO

The PPO baselines use the PPO implementation of Stable-Baselines3 [35].

- **Discrete tasks.** PPO is trained with 2M steps on classic control, while DoorKey and LunarLander are trained for 4M steps. These runs use a learning rate 10^{-3} and a minibatch size of 64.
- **Continuous tasks.** PPO is trained for 250k steps for Pusher2D, 5M for HalfCheetah Hurdle, 6M for Ant RandomGoal, and 5M for Ant CrossMaze. These runs use a learning rate 10^{-3} and a minibatch size of 64.

Note that both π-PRL and DiPRL keep the same hyperparameter setting with PPO but replace the neural policy with a programmatic policy.

C.4 VIPER

VIPER [34] is trained as a post-hoc distillation baseline from the PPO oracle for the same task and seed. VIPER uses Q-Dagger with a decision-tree classifier of maximum depth 6 and 100k sampled timesteps per run.

C.5 DTSemNets

DTSemNets is trained using the original implementation [28]. Budgets for all tasks are set to be the same as DiPRL.

C.6 Computational Resource

All algorithms are trained and tested on a 128-CPU server.

- CPU: AMD Rome 7H12 (2x) 64 Cores/Socket 2.6GHz 280W
- CPU memory: 256 GiB DRAM (2 GiB per core)
- DIMMs: 16 x 16GiB 3200MHz, DDR4

Table 3: List of licenses for assets used in this work

Resource	Type	Link	License
Tianshou [36]	Code	https://github.com/thu-ml/tianshou/tree/master?tab=readme-ov-file	MIT License
OpenAI'Gym	Code	https://github.com/openai/gym	Available for academic use
Qiu and Zhu [9]	Code	https://github.com/RU-Automated-Reasoning-Group/pi-PRL	Available for academic use
Stable-Baselines3 [35]	Code	https://github.com/DLR-RM/stable-baselines3	Available for academic use
π-PRL	Code	https://github.com/RU-Automated-Reasoning-Group/pi-PRL	Available for academic use
DTSemNets [28]	Code	https://github.com/CPS-research-group/dtsemnet	Available for academic use

D Ablation study on target entropy

There is a trade-off in setting $\bar{H}$. For example, setting $\bar{H} = 0$ may cause premature collapse, resulting in a trivial program with limited expressivity. $H(\mathcal{T})$ measures architecture uncertainty, which is determined by the maximal depth D_m, i.e., $H(\mathcal{T}) = H(D) = -\sum_{d=1}^{D_m} p_d \log p_d$. Thus, $\log D_m$ provides the natural scale of the uncertainty. In preliminary experiments, we studied the effect of the target entropy by considering $\bar{H} \in \{0, 0.1 \log D_m, 0.5 \log D_m, \log D_m\}$. We consider $\log D_m$ as $H_{\max}$. Based on Tab. 4 and Fig. 9, we found that the performance degenerates with larger target entropy. It is intuitive since large entropy has less effect in encouraging a discrete program, while zero might lead to a premature collapse. Thus, we select $0.1 \log D_m$ as the target entropy for DiPRL.

Table 4: Ablation on target entropy selected from $\bar{H} \in \{0, 0.1 \log D_m, 0.5 \log D_m, \log D_m\}$.

Target entropy $\bar{H}$	CartPole-v1	Acrobot-v1	MountainCar-v0	DoorKey	LunarLander-v2
0	500.00 ± 0.00	-78.72 ± 1.74	-115.56 ± 4.25	0.61 ± 0.43	259.96 ± 7.89
$0.1 \log D_m$	500.00 ± 0.00	-79.93 ± 3.37	-110.79 ± 4.07	0.95 ± 0.01	260.21 ± 13.61
$0.5 \log D_m$	500.00 ± 0.00	-83.42 ± 5.07	-115.64 ± 2.69	0.41 ± 0.39	243.48 ± 35.23
$\log D_m$	500.00 ± 0.00	-83.14 ± 5.09	-114.76 ± 3.73	0.19 ± 0.17	213.95 ± 34.17

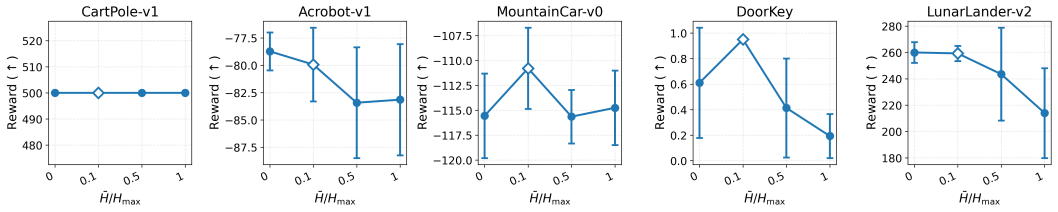

Figure 9: Ablation on target entropy selected from $\bar{H} \in \{0, 0.1 \log D_m, 0.5 \log D_m, \log D_m\}$.

E Ablation study on regularization strength

The coefficient α controls how strongly DiPRL penalizes architecture entropy. If α is too small, the relaxed policy may still rely on several program branches when the final program is extracted. If α is too large, the architecture may become discrete before the policy has explored useful branches and parameters.

Automatic coefficient tuning. It is hard to manually select a fixed penalty coefficient α for $H(\mathcal{T})$ (see Eq. 5) across different tasks. We therefore tune α during training. The dual objective with respect to α is defined as $g(\alpha) = \max_{\mathcal{T},\mathcal{W}} \mathcal{L}(\mathcal{T}, \mathcal{W}, \alpha)$. We update α by approximating the dual gradient according to [37]:

$$\nabla_\alpha g(\alpha) = \bar{H} - H(\mathcal{T}), \tag{27}$$

In practice, we optimize $\beta = \log \alpha$ to ensure positivity. This update increases α when the architecture entropy is above the target and decreases it when the entropy is already low enough. Thus, automatic tuning allows exploration in the early stage and encourages convergence to a discrete program later.

Ablation results. Table 5 compares each policy before and after discretization. The $\times$ rows evaluate the relaxed policy before extracting the final program, while the ✓ rows evaluate the extracted discrete program. No fine-tuning is applied. This comparison shows whether a choice of α already learns a stable discrete architecture during training or still depends on a relaxed mixture of program branches.

Small coefficients do not consistently control the discretization gap. For example, $\text{DiPRL}_{0.001}$ reaches 0.94 on DoorKey before discretization, but drops to 0.70 after extraction. $\text{DiPRL}_{0.01}$ shows an even larger DoorKey drop, from 0.62 to 0.13. DiPRL_0 also degrades on MountainCar-v0, from -148.17 to -173.74. These drops indicate that weak regularization can leave important behavior spread across several branches of the relaxed derivation tree. When the final program is extracted, some of these branches are removed, and the resulting discrete program is worse than the relaxed policy.

Table 5: Ablation of the regularization coefficient α on discrete-control tasks. In the Discretization column, $\times$ denotes the relaxed policy before extraction and ✓ denotes the extracted discrete program. No fine-tuning is applied.

Algorithm	Discretization	CartPole-v1	Acrobot-v1	MountainCar-v0	DoorKey	LunarLander-v2
DiPRL$_{0}$	$\times$	500.00 ± 0.00	-80.99 ± 2.15	-148.17 ± 36.87	0.63 ± 0.45	250.05 ± 13.40
	✓	500.00 ± 0.00	-85.09 ± 5.01	-173.74 ± 35.73	0.38 ± 0.30	250.82 ± 10.89
DiPRL$_{0.001}$	$\times$	500.00 ± 0.00	-77.89 ± 3.51	-111.37 ± 3.85	0.94 ± 0.00	253.36 ± 21.41
	✓	500.00 ± 0.00	-78.38 ± 2.72	-112.36 ± 3.12	0.70 ± 0.16	237.98 ± 38.33
DiPRL$_{0.01}$	$\times$	500.00 ± 0.00	-81.23 ± 4.15	-111.90 ± 4.05	0.62 ± 0.44	254.59 ± 4.42
	✓	500.00 ± 0.00	-80.32 ± 4.14	-112.71 ± 5.53	0.13 ± 0.10	244.04 ± 29.27
DiPRL$_{0.1}$	$\times$	500.00 ± 0.00	-78.59 ± 1.46	-116.64 ± 1.57	0.92 ± 0.02	61.03 ± 124.29
	✓	500.00 ± 0.00	-77.89 ± 1.52	-114.44 ± 4.12	0.64 ± 0.42	65.03 ± 118.85
DiPRL$_{0.5}$	$\times$	500.00 ± 0.00	-80.05 ± 3.82	-111.24 ± 3.94	0.92 ± 0.03	165.26 ± 144.34
	✓	500.00 ± 0.00	-76.19 ± 0.73	-113.46 ± 3.99	0.91 ± 0.02	169.42 ± 133.09
DiPRL$_{\text{auto}}$	$\times$	500.00 ± 0.00	-79.08 ± 2.77	-113.40 ± 3.55	0.95 ± 0.01	248.54 ± 2.00
	✓	500.00 ± 0.00	-79.93 ± 3.37	-110.79 ± 4.07	0.95 ± 0.01	260.21 ± 13.61

Large fixed coefficients reduce this discretization gap in some cases, but they can hurt learning before extraction. DiPRL$_{0.5}$ is stable on DoorKey after discretization, changing only from 0.92 to 0.91, but it performs poorly on LunarLander-v2 compared with the automatic setting. DiPRL$_{0.1}$ has the same problem: it remains far below the best LunarLander-v2 reward and still drops on DoorKey after discretization. This suggests that forcing discreteness too strongly can reduce the exploration of useful program branches and parameters.

DiPRL$_{\text{auto}}$ avoids choosing a single fixed coefficient for all tasks. It keeps DoorKey unchanged after discretization at 0.95, improves MountainCar-v0 after extraction from -113.40 to -110.79, and achieves the best LunarLander-v2 result after discretization. Automatic tuning keeps architecture exploration flexible early in training, then increases pressure toward a stable discrete program when the learned policy is ready to be extracted.

F Additional experiment results

F.1 Program depth comparison between π-PRL and DiPRL

Tab 6 presents the depth of the final discrete program of π-PRL and DiPRL. π-PRL has to discard branches, thus has few chance to explore deeper programs. DiPRL encourages discrete program convergence during training. Better programs with deeper depth can have higher chance to be found.

Table 6: Final discrete program depth (avg. $\pm$ std) for π-PRL and DiPRL.

Task	π-PRL	DiPRL (ours)
CartPole-v1	4.00 ± 0.82	4.67 ± 0.47
Acrobot-v1	2.33 ± 0.47	4.33 ± 0.47
MountainCar-v0	1.33 ± 0.47	3.67 ± 0.47
DoorKey-6x6	3.67 ± 1.89	4.67 ± 0.47
LunarLander-v2	4.33 ± 0.47	4.33 ± 0.47
HalfCheetah Hurdle	1.67 ± 0.47	1.67 ± 0.94
Pusher2D	1.67 ± 0.47	2.33 ± 0.47
Ant RandomGoal	3.00 ± 1.41	3.33 ± 1.25
Ant CrossMaze	2.33 ± 1.89	2.67 ± 0.47

F.2 Architecture entropy comparison between $\pi_{\text{cont.}}$-PRL and DiPRL

Table 7 reports the final normalized architecture entropy before discretization. An interesting point is that π-PRL drops to a 0.004 architecture entropy in HalfCheetah Hurdle, but it still experiences a performance drop. This is because the reported architecture entropy measures the uncertainty of the program architecture. After extraction, π-PRL instantiates a new program policy for Phase 2 fine-tuning. The stochastic action distribution, considering the parameterized action terminal, is still unstable with hard thresholdings. Even when the architecture is nearly deterministic, the behavior of the discrete program may be identical to the relaxed policy. That is why π-PRL introduces the post-hoc fine-tuning. DiPRL consistently drives the architecture entropy close to zero while avoiding cases like this, whereas $\pi_{\text{cont.}}$-PRL often remains high-entropy on almost discrete and continuous tasks. This supports the claim that DiPRL learns a near-discrete architecture during training, reducing the mismatch introduced by discretization.

Table 7: Final normalized architecture entropy before discretization. Values are mean $\pm$ std across three seeds. Lower values indicate that the relaxed architecture is closer to a discrete program.

Task	$\pi_{\text{cont.}}$-PRL	DiPRL
CartPole-v1	0.647 ± 0.178	0.001 ± 0.001
Acrobot-v1	0.861 ± 0.070	0.004 ± 0.001
MountainCar-v0	0.784 ± 0.119	0.003 ± 0.000
DoorKey	0.767 ± 0.109	0.000 ± 0.000
LunarLander-v2	0.284 ± 0.149	0.002 ± 0.001
HalfCheetah Hurdle	0.004 ± 0.006	0.000 ± 0.000
Pusher2D	0.407 ± 0.328	0.000 ± 0.000
Ant RandomGoal	0.490 ± 0.071	0.000 ± 0.000
Ant CrossMaze	0.785 ± 0.063	0.024 ± 0.033

F.2.1 Discrete tasks

Fig. 10 shows that DiPRL keeps competitive reward while gradually driving the architecture entropy to zero. This trend is important because the final policy is a discrete program. Low entropy means that the learned derivation tree has already concentrated on one architecture before evaluation. In contrast, π-PRL often keeps high architecture entropy for most of training and only becomes discrete after the post-hoc extraction step. The reward curves are similar on simple tasks such as CartPole-v1

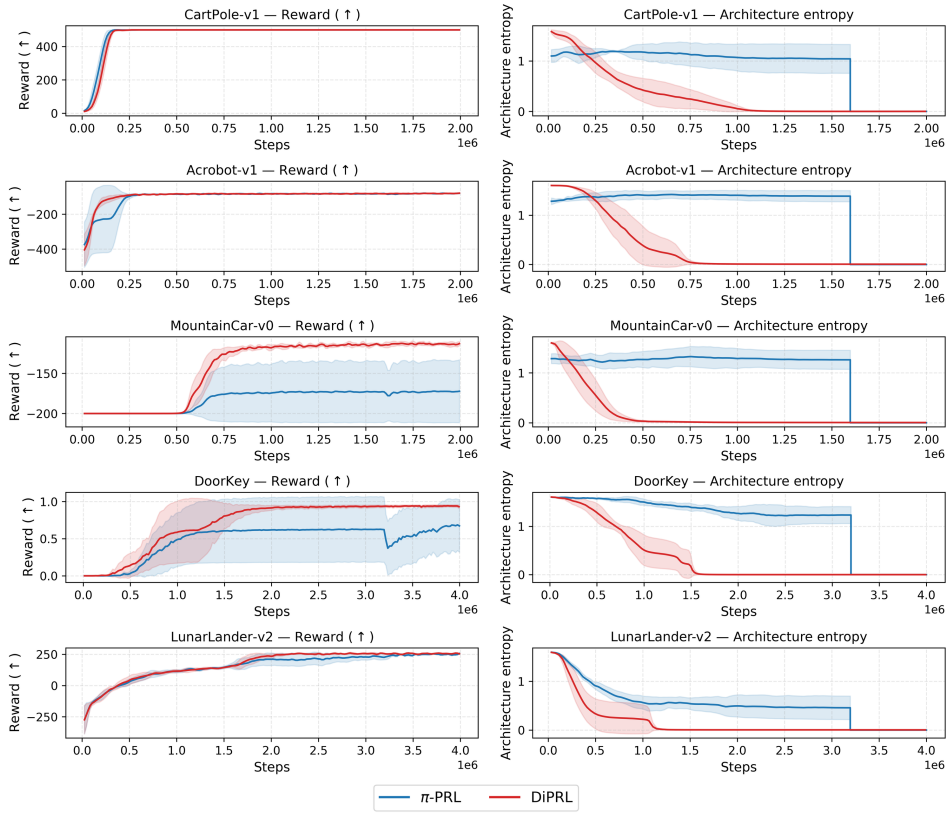

Figure 10: Training curves on discrete tasks. The left column reports reward, where higher is better, and the right column reports architecture entropy. DiPRL reduces architecture entropy during training while maintaining competitive reward.

Table 8: Goal distance curves in continuous tasks; smaller is better.

Algorithm	HalfCheetah Hurdle	Pusher2D	Ant RandomGoal	Ant CrossMaze
PPO	0.53 ± 0.72	0.66 ± 0.17	3.80 ± 0.48	6.65 ± 0.06
VIPER (PPO)	0.60 ± 0.56	0.59 ± 0.24	2.93 ± 0.33	8.17 ± 0.74
DTSemNets	10.27 ± 0.69	0.14 ± 0.06	2.23 ± 0.68	8.37 ± 0.59
$\pi_{\text{cont.}}$-PRL	0.26 ± 0.16	0.15 ± 0.05	1.84 ± 0.90	2.79 ± 0.57
$\pi_{\text{disc.}}$-PRL	0.36 ± 0.39	0.21 ± 0.02	3.05 ± 0.31	6.29 ± 0.42
π-PRL	0.57 ± 0.78	0.18 ± 0.02	2.35 ± 1.05	4.43 ± 1.43
DiPRL (ours)	$\mathbf{0.07 \pm 0.04}$	$\mathbf{0.12 \pm 0.01}$	$\mathbf{0.99 \pm 0.83}$	$\mathbf{1.76 \pm 1.11}$

and Acrobot-v1, but DiPRL gives clearer gains on harder tasks such as MountainCar-v0 and DoorKey, where stable discretization is more important.

F.2.2 Continuous tasks

Fig. 11 reports reward, goal distance, and architecture entropy on continuous-control tasks. Tab. 8 presents the goal distance. DiPRL generally improves reward while reducing goal distance, especially on HalfCheetah Hurdle and Ant RandomGoal. The entropy curves show that DiPRL also converges toward a discrete architecture during training. This supports the same conclusion as in the discrete tasks. The policy does not need a sudden post-hoc discretization step, because the architecture is already close to discrete when training ends.

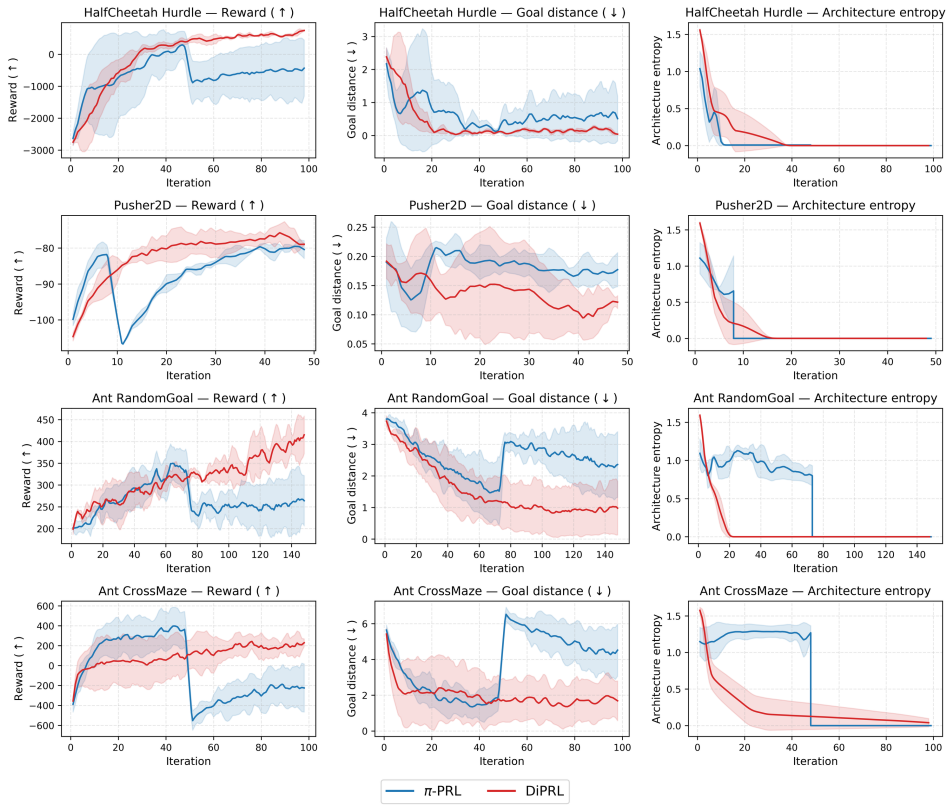

Figure 11: Training curves on continuous tasks. Each row reports reward, goal distance, and architecture entropy for one task. Higher reward and lower goal distance indicate better performance.

G Domain

We validate DiPRL and the baselines on both discrete and continuous tasks.

G.1 Discrete tasks

We evaluate on five finite-action tasks: three classic-control domains (CartPole-v1, Acrobot-v1, and MountainCar-v0), one MiniGrid navigation domain (DoorKey), and one Box2D control domain (LunarLander-v2).

G.1.1 CartPole-v1

The agent controls a cart moving on a one-dimensional track with a pole hinged to the cart. The state is

$$s_t = (x_t, \dot{x}_t, \theta_t, \dot{\theta}_t), \tag{28}$$

where x_t is cart position and θ_t is pole angle. The action space is $\mathcal{A} = \{\text{LEFT}, \text{RIGHT}\}$, corresponding to horizontal forces applied to the cart. The task is to keep the pole upright while the cart remains within the track boundary. The return is the number of balanced time steps, capped at 500, so higher is better. Each step gives a reward for reaching the goal.

G.1.2 Acrobot-v1

Acrobot is an underactuated two-link pendulum. Only the joint between the two links is actuated, and the objective is to swing the free end above a target height. The observation contains a trigonometric encoding of both joint angles and their angular velocities,

$$s_t = (\cos\theta_{1,t}, \sin\theta_{1,t}, \cos\theta_{2,t}, \sin\theta_{2,t}, \dot{\theta}_{1,t}, \dot{\theta}_{2,t}). \tag{29}$$

The action space is $\mathcal{A} = \{-1, 0, +1\}$, representing negative, zero, or positive torque. The environment gives a per-step penalty until the goal is reached, so better policies have less negative returns.

G.1.3 MountainCar-v0

MountainCar controls an underpowered car that must build momentum to reach the goal at the top of a hill. The state is

$$s_t = (p_t, \dot{p}_t), \tag{30}$$

where p_t is position and $\dot{p}_t$ is velocity. The action space is $\mathcal{A} = \{\text{LEFT}, \text{COAST}, \text{RIGHT}\}$. The canonical task gives a penalty at each time step until the goal is reached or the episode times out, so higher returns correspond to reaching the goal in fewer steps. During training, distance-based reward shaping is used in the experimental setup; evaluation reports the task return.

G.1.4 DoorKey

DoorKey is a MiniGrid navigation task. The agent must navigate a grid, pick up a key, unlock a door, and reach the goal cell. The action space contains discrete navigation and object-interaction actions, including turning, moving forward, picking up the key, and toggling the door. The task has sparse success feedback: unsuccessful episodes receive zero return, while successful episodes receive a positive reward that is larger when the goal is reached earlier.

G.1.5 LunarLander-v2

LunarLander-v2 is a discrete-action Box2D landing task. The lander state includes its position, velocity, angle, angular velocity, and leg-contact indicators. The action space is

$$\mathcal{A} = \{\text{NOOP}, \text{LEFT ENGINE}, \text{MAIN ENGINE}, \text{RIGHT ENGINE}\}. \tag{31}$$

The objective is to land softly between the landing flags while controlling fuel use and avoiding crashes. Higher episodic return indicates a more stable and efficient landing policy.

G.2 Continuous tasks

We also evaluate on four continuous tasks introduced by Qiu and Zhu [9]: HalfCheetah Hurdle, Pusher2D, Ant RandomGoal, and Ant CrossMaze. These tasks use continuous action spaces and require locomotion or manipulation policies that make progress toward task-specific goals. In addition to reward, we report goal distance for these domains, where lower distance indicates better goal reaching.

G.2.1 HalfCheetah Hurdle

HalfCheetah Hurdle is a locomotion task in which a half-cheetah robot must move forward while clearing a sequence of hurdles. The observation contains the robot's proprioceptive state together with task-specific information about the next hurdle. The action is a continuous six-dimensional motor command,

$$a_t \in \mathbb{R}^6. \tag{32}$$

The reward combines forward progress, jumping behavior, distance to the target position, hurdle-related shaping, a large completion bonus, and collision penalties. The task is solved by reaching the target region beyond the hurdles; higher reward and lower final goal distance indicate better performance.

G.2.2 Pusher2D

Pusher2D is a planar manipulation task. A three-joint arm must push a puck to a fixed goal location. The observation includes trigonometric joint encodings, joint velocities, the end-effector position, and the object position. The action is a continuous three-dimensional control vector,

$$a_t \in \mathbb{R}^3. \tag{33}$$

The reward is the negative weighted sum of the arm-object distance, the object-goal distance, and a control penalty. Consequently, better policies keep the arm close to the object, move the object toward the goal, and use smoother controls.

G.2.3 Ant RandomGoal

Ant RandomGoal is a quadruped locomotion task with a randomly sampled target position in a bounded circular arena. The observation contains the ant's proprioceptive state and the current goal position. The action is a continuous eight-dimensional torque command,

$$a_t \in \mathbb{R}^8. \tag{34}$$

The agent receives dense reward for reducing its distance to the sampled goal, with additional healthy-survival reward and control/contact costs. Episodes terminate when the ant reaches the goal or leaves the allowed area. We measure both episodic reward and the final distance to the sampled goal.

G.2.4 Ant CrossMaze

Ant CrossMaze uses the same ant morphology but places the agent in a maze-like environment with walls and candidate goal locations. The benchmark variant used in our experiments samples the goal from a small set of maze targets. The observation contains the ant state and the active goal position, and the action is again a continuous eight-dimensional torque command,

$$a_t \in \mathbb{R}^8. \tag{35}$$

The task requires both locomotion and navigation: the ant must move through the maze to the active goal while avoiding poor routes and unstable contacts. Reward is based on goal progress, survival, and control/contact penalties; lower final goal distance indicates more reliable navigation.

H Declaration of LLM usage

We used generative AI for proofreading. All scientific content, claims, and results were produced and verified by the authors.