Inductive Generalization in Reinforcement Learning from Specifications

Vignesh Subramanian[1], Rohit Kushwah[2], Subhajit Roy[2], and Suguman Bansal[1]

[1] School of Computer Science, Georgia Institute of Technology, USA
{vignesh,suguman}@gatech.edu
[2] Department of Computer Science, Indian Institute of Technology Kanpur, India
{krohitk,subhajit}@cse.iitk.ac.in

Abstract. We present a novel *inductive generalization framework* for RL from logical specifications. Many interesting tasks in RL environments have a natural inductive structure. These *inductive tasks* have similar overarching goals but they differ inductively in low-level predicates and distributions. We present a generalization procedure that leverages this inductive relationship to learn a higher-order function, a *policy generator*, that generates appropriately *adapted* policies for instances of an inductive task in a zero-shot manner. An evaluation of the proposed approach on a set of challenging control benchmarks demonstrates the promise of our framework in generalizing to unseen policies for long-horizon tasks.

1 Introduction

Formal methods community has contributed strongly to *reinforcement learning (RL)*, especially from formal specifications [1, 7, 9, 10, 13, 14, 20, 25, 38, 40]. These techniques may not provide strong guarantees. In fact, their inability to offer rigorous guarantees has been proven [2, 39]. Nevertheless, these methods provide a principled approach for handling learning over long-horizon tasks.

Generalization remains one of the fundamental challenges in RL. While RL agents can achieve impressive performance on individual tasks, they often struggle to transfer learned behaviors to even slightly modified scenarios. Most RL approaches lack formal mechanisms to capture and exploit structural relationships between tasks, instead relying on implicit generalization through neural network function approximation. This often leads to superficial generalization that fails to capture deeper task similarities. Recent work has attempted to address these challenges through meta-learning and goal-conditioned learning, but developing RL algorithms that can generalize remains an open challenge.

While the challenge of overall generalization is too large to address, this work presents a novel notion of generalization, which we call *inductive generalization*. This is based on leveraging inductive similarities between tasks to generalize. Inductive relationships are fundamental to computational tasks because they capture how complex behaviors can be built from simpler ones through systematic transformation. They appear naturally whenever tasks exhibit natural recursion or iteration. Our insight is that once we understand how to transform from step

i to step $i+1$, we can systematically generalize to handle arbitrarily many steps. This naturally arising pattern is particularly evident in robotics and control tasks, where physical constraints often impose regular structure. By formalizing these inductive relationships, we can move beyond treating each task instance as independent and instead leverage their inherent structural connections to enable systematic generalization.

To this end, we present a *logic-guided approach to inductive generalization* in RL. We use logical specifications to encode a class of *inductive tasks* which comprises of several similar tasks that can be enumerated from each other using an inductive relationship. We require that these tasks have identical logical structure but differ only in the low-level details of predicate values and/or environment parameters. Next, we leverage their similar structures to design a generalizable RL algorithm.

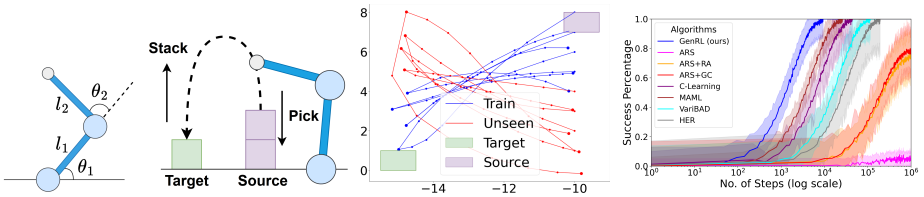
Formally, an inductive task is given by a tuple $\mathsf{R} = (\mathsf{R}_0, \mathsf{update\_pred}, \mathsf{update\_init})$ where R_0 is a *base task* given as a temporal logical specification over given predicates and $\mathsf{update\_pred}$ and $\mathsf{update\_init}$ define inductive updates to the predicates of the specification and environment parameters, respectively, to be applied to the base task repeatedly, so as to generate a family of tasks $\mathsf{R}_1, \mathsf{R}_2, \ldots$ and such. Intuitively, one can think of an inductive task to represent a complex task involving iterations as follows:

```
base_task = \task_0 // Encodes the base spec. and envt. conditions
current_task = base_task
repeat
    next_task = update_pred(current_task), update_init(current_task)
    current_task = next_task
```

For instance, in Figure 1 the robot is required to transport the source pile of boxes to a target pile. This complex task can be decomposed into a series of inductively related tasks: for the i-th instance, pick the topmost box from a height of i in *Source* and place it at the top of the *Target* pile at height $(h-i)$ (where h is the total number of blocks). Here the 0-th task instance forms the base task, and the update functions change the location of the topmost block in the source and target piles. Observe that all task instances in an inductive task have identical logical structure. They only differ in the instantiations of the low-level predicates and environment variables.

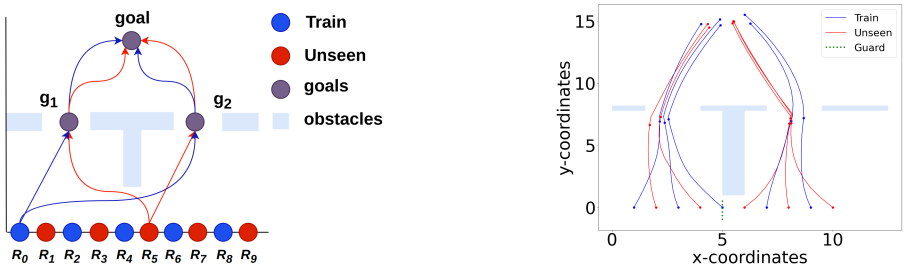
Our goal is to leverage this inductive relationship between tasks to design generalizable RL algorithms to learn policies for each task instance. Concretely, the question we ask is: if trained on a few task instances of an inductive task, can we obtain policies for the remaining tasks in a *zero-shot* manner?

Such generalization is difficult, in general. To solve this problem, we hypothesize that *inductively-related task instances may have inductively-related policies*. Based on this hypothesis, we attempt to learn an *inductive relationship* between the policies of simpler task instances to extract a *policy generator*: a higher-order function that returns an *adapted* policy for a given inductive task instance. Figure 1(c) shows the trajectories of the pickup head with $h = 8$ blocks: we trained a policy generator for the robot on picking and placing the first four blocks (shown



(a) Agent dynamics (b) Inductive Task (c) Trajectories (d) Sample Complexity

Fig. 1: Tower Destacking: The task is to pick boxes from *Source* and stack it on *Target*.



(a) Inductive Task (b) Trajectories

Fig. 2: Choice: visit either g_1 or g_2, then visit goal; task instances differ in initial state distribution.

in blue); The robot could complete the whole task, with adapted policies from the learnt policy generator for the unseen task instances, i.e. pick-n-place of the bottom four blocks are shown in red. We see that the policy generator lends *significant adaptability* to the robot to control its θ_1 and θ_2, as the trajectories of the task instances are quite different.

However, our hypothesis may not always hold. It is possible that despite the task being inductive, the policies are not immediately inductive. The motivating example from Figure 2 illustrates this complication. Figure 2 illustrates an *inductive task* in a 2D Cartesian plane: in a task instance, the agent is initially located in one of the blue or red regions marked R_k. The goal is to *visit the region* marked goal, *after visiting* one of the intermediate regions g_1 or g_2, while *always avoiding* the obstacles shown in light blue. The task is inductive on the initial position: the $(k+1)$-th task can be defined in terms of the k-th task, by shifting the initial location to the right by c units.

However, the policies are not inductive: there is a task R_k such that its policy needs to route through g_1 but the policy of task R_{k+1} must route through g_2 (eg. R_4 and R_5). Yet, we may be able to *classify* the task instances into multiple groups, such that all tasks in each group is *inductive* (eg. $\{\mathsf{R}_0, \ldots, \mathsf{R}_4\}$ and $\{\mathsf{R}_5,$

$\ldots, \mathsf{R}_9\}$). Our policy generator learns such *branches* such that the task instances on the same decision of the branch have inductive policies.

The benchmark environments we utilize in this work are particularly well-suited for evaluating inductive generalization capabilities. Our tasks span a range from simple reachability in 2D environments to complex robotic manipulation scenarios, all unified by their inherent inductive structure. These environments feature continuous state and action spaces, long-horizon planning requirements, and varying degrees of physical constraints, which are challenging even for RL without generalization.

We summarize our contributions: (a). We introduce a framework to learn inductively generalizable policies for long-horizon tasks. This comprises formalizing the notion of inductively-related tasks based on their logical specification and describing the generalization problem as learning a higher-order policy generator (Section 3). (b). We describe a procedure to learn a neural policy generator by leveraging the inductive relationship between task instances (Section 4-Section 5). (c). We perform an empirical evaluation of our inductive framework for generalization in learning unseen tasks in complex, long-horizon specifications in continuous environments, popular control environments, and robotic pick-n-place tasks. Our evaluation demonstrates the promise of our inductive approach (Section 6) as we are able to show that our approach outperforms mature policy-gradient state-of-the-art generalizable RL algorithms in their ability to generalize to unseen tasks and sample complexity.

2 Preliminaries

2.1 Markov Decision Process (MDP)

The environment in RL is given by a MDP $\mathcal{M} = (S, A, P, \eta)$ with continuous states $S \subseteq \mathbb{R}^n$, continuous actions $A \subseteq \mathbb{R}^m$, transitions $P(s, a, s') = p(s' \mid s, a) \in \mathbb{R}_{\geq 0}$ (i.e., probability density of transitioning from state s to state s' upon taking action a), and initial states $\eta : S \to \mathbb{R}_{\geq 0}$ (i.e., $\eta(s)$ is the probability of the initial state being s). A *trajectory* $\zeta \in \mathcal{Z}$ is either an infinite sequence $\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$ or a finite sequence $\zeta = s_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{t-1}} s_t$ where $s_i \in S$ and $a_i \in A$. A subtrajectory of ζ is a subsequence $\zeta_{\ell:k} = s_\ell \xrightarrow{a_\ell} \cdots \xrightarrow{a_{k-1}} s_k$. We let $\mathcal{Z}_f$ denote the set of finite trajectories. A (deterministic) *policy* $\pi : \mathcal{Z}_f \to A$ maps a finite trajectory to a fixed action.

Crucially, in RL we assume that the transition probabilities of the MDP is unknown. Hence, the MDP is accessed by sampling only. Concretely, given a policy π, we can sample a trajectory by sampling an initial state $s_0 \sim \eta(\cdot)$, and then iteratively taking the action $a_i = \pi(\zeta_{0:i})$ and sampling a next state $s_{i+1} \sim p(\cdot \mid s_i, a_i)$.

2.2 SPECTRL Specification Language and their Abstract Graphs

We express RL tasks using the logical specification language SPECTRL [18]. Every SPECTRL specification can be expressed as an *abstract graph* which can be used to design scalable compositional algorithms for RL from logical specifications [19].

A SPECTRL specification is defined over a set of *atomic predicates* $\mathcal{P}_0$ that ground environment states, where every $p \in \mathcal{P}_0$ is associated with a function $[\![p]\!] : S \to \mathbb{B} = \{\texttt{true}, \texttt{false}\}$; we say a state s *satisfies* p (denoted $s \models p$) if and only if $[\![p]\!](s) = \texttt{true}$. For $b \in \mathcal{P}$, the syntax of SPECTRL is: $\phi ::= \texttt{achieve}\ b \mid \phi_1\ \texttt{ensuring}\ b \mid \phi_1; \phi_2 \mid \phi_1\ \texttt{or}\ \phi_2$. Each specification ϕ corresponds to a function $[\![\phi]\!] : \mathcal{Z} \to \mathbb{B}$, and we say $\zeta \in \mathcal{Z}$ satisfies ϕ (denoted $\zeta \models \phi$) if and only if $[\![\phi]\!](\zeta) = \texttt{true}$. Intuitively, '`achieve` ' and ' `ensuring` ' are reachability and safety goals, respectively. ';' and '`or`' refer to sequencing and disjunction, respectively. Letting ζ be a finite trajectory of length t, this function is defined by

$$
\begin{aligned}
&\zeta \models \texttt{achieve}\ b && \text{if } \exists\ i \le t,\ s_i \models b \\
&\zeta \models \phi\ \texttt{ensuring}\ b && \text{if } \zeta \models \phi \text{ and } \forall\ i \le t,\ s_i \models b \\
&\zeta \models \phi_1; \phi_2 && \text{if } \exists\ i < t,\ \zeta_{0:i} \models \phi_1 \text{ and } \zeta_{i+1:t} \models \phi_2 \\
&\zeta \models \phi_1\ \texttt{or}\ \phi_2 && \text{if } \zeta \models \phi_1 \text{ or } \zeta \models \phi_2.
\end{aligned}
$$

Abstract Graph. An *abstract graph* of a SPECTRL specification is a DAG-like structure in which vertices represent sets of states (called subgoal regions) and edges represent sets of MDP trajectories that can be used to transition from the source to the target vertex without violating safety constraints.

Definition 1. An *abstract graph* $\mathcal{G} = (U, E, u_0, F, \beta, \mathcal{Z}_{\text{safe}})$ is a directed acyclic graph (DAG) with vertices U, (directed) edges $E \subseteq U \times U$, initial vertex $u_0 \in U$, final vertices $F \subseteq U$, subgoal region map $\beta : U \to 2^S$ such that for each $u \in U$, $\beta(u)$ is a subgoal region, and *safe trajectories* $\mathcal{Z}_{\text{safe}} = \bigcup_{e \in E} \mathcal{Z}^e_{\text{safe}} \cup \bigcup_{f \in F} \mathcal{Z}^f_{\text{safe}}$, where $\mathcal{Z}^e_{\text{safe}} \subseteq \mathcal{Z}$ denotes the safe trajectories for edge $e \in E$ and $\mathcal{Z}^f_{\text{safe}} \subseteq \mathcal{Z}$ denotes the safe trajectories for final vertex $f \in F$.

Intuitively, (U, E) is a DAG, and u_0 and F define a graph reachability problem for (U, E). Furthermore, β and $\mathcal{Z}_{\text{safe}}$ connect (U, E) back to the original MDP $\mathcal{M}$; in particular, for an edge $e = u \to u'$, $\mathcal{Z}^e_{\text{safe}}$ is the set of safe trajectories in $\mathcal{M}$ that can be used to transition from $\beta(u)$ to $\beta(u')$.

A trajectory $\zeta = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{t-1}} s_t$ in $\mathcal{M}$ satisfies the abstract graph $\mathcal{G}$ (denoted $\zeta \models \mathcal{G}$) if there is a sequence of indices $0 = k_0 \le k_1 < \cdots < k_\ell \le t$ and a path $\rho = u_0 \to u_1 \to \cdots \to u_\ell$ in $\mathcal{G}$ such that (a). $u_\ell \in F$, (b). for all $z \in \{0, \ldots, \ell\}$, we have $s_{k_z} \in \beta(u_z)$, (c). for all $z < \ell$, letting $e_z = u_z \to u_{z+1}$, we have $\zeta_{k_z:k_{z+1}} \in \mathcal{Z}^{e_z}_{\text{safe}}$, and (d). $\zeta_{k_\ell:t} \in \mathcal{Z}^{u_\ell}_{\text{safe}}$. The first two conditions state that the trajectory should visit a sequence of subgoal regions corresponding to a path from the initial vertex to some final vertex, and the last two conditions state that the trajectory is composed of subtrajectories that are safe according to $\mathcal{Z}_{\text{safe}}$.

The *edge policy* π_e for an edge $e = u \to u'$ is one that safely transitions from a state in $\beta(u)$ to a state in $\beta(u')$. Given edge policies Π along with a path

$\rho = u_0 \to u_1 \to \cdots \to u_k = u$ in $\mathcal{G}$, the *path policy* π_ρ navigates from $\beta(u_0)$ to $\beta(u)$. In particular, π_ρ executes $\pi_{u_j \to u_{j+1}}$ (starting from $j = 0$) until reaching $\beta(u_{j+1})$, after which it increments $j \leftarrow j + 1$ (unless $j = k$). Learning an optimal policy for SPECTRL is reduced to learning an optimal path policy from the initial to final vertex. This gives rise to a natural compositional learning approach that first learns edge policies and then returns the path policy with the maximum probability of reaching a final vertex [19].

3 Inductive Tasks

We begin by describing *inductive tasks*. These appear naturally in several scenarios, as shown in Figures 1-2.

Notation. An *RL task* be given by the tuple (ϕ, η) where ϕ is a SPECTRL specification and η is the initial state distribution in the MDP. We say a trajectory $\zeta = s_0 \dots s_t$ satisfies an RL task (ϕ, η), denoted $\zeta \models (\phi, \eta)$, if $s_0 \sim \eta$ and $\zeta \models \phi$, I.e., ζ begins in a state sampled from η and ζ satisfies ϕ.

An *inductive task* is a family of *RL tasks* that demonstrate the same overarching structure but differ inductively in the low-level details. I.e., an *inductive task* is given by a set of enumerable RL tasks such that the $(i+1)$-th task builds on the i-th task by updating the predicates in the specification and/or the MDP initial distribution. Formally,

Definition 2. *Let $\mathcal{P}$ and $D(S)$ denote the sets of predicates and state distributions in an MDP, respectively. Let $\phi(P)$ denote a SPECTRL specification defined over predicates $P \subseteq \mathcal{P}$. Then, an* inductive task *is given by* $\mathsf{R} = (\mathsf{R}_0, \mathsf{update\_pred}, \mathsf{update\_init})$ *where RL task* $\mathsf{R}_0 = (\phi(\mathcal{P}_0), \eta_0)$ *is the* base task*,* $\mathsf{update\_pred} : \mathcal{P} \mapsto \mathcal{P}$ *is the* predicate update function*, and* $\mathsf{update\_init} : \mathcal{D}(S) \mapsto \mathcal{D}(S)$ *is the* initial distribution update function*. The enumerable* task instances *in* R *are given by* $\mathsf{R}_0 = (\phi(\mathcal{P}_0), \eta_0)$ *and* $\mathsf{R}_{i+1} = (\phi(\mathcal{P}_{i+1}), \eta_{i+1})$ *for* $i > 0$ *where* $\mathcal{P}_{i+1} = \{\mathsf{update\_pred}(p) \mid p \in \mathcal{P}_i\}$ *and* $\eta_{i+1}(s) = \eta_i(\mathsf{update\_init}(s))$.

We denote the i-th task instance R_i by (ϕ_i, η_i) and refer to task instances R_i and R_{i+1} as *adjacent*.

Motivating Example #1. For Figure 1, the inductive task is formalized as: For $j \in \{0, \dots, h\}$, let the predicates $\mathsf{source\_j}$ and $\mathsf{target\_j}$ denote the location of the block at height j in the source and target tower, respectively; let $\eta\_\mathsf{source\_j}$ be a distribution around the block at height j in the source tower. The base task R_0 is given by

$$((\texttt{achieve}\ (\mathsf{target\_0})); (\texttt{achieve}\ (\mathsf{source\_(h-1)}))), \eta\_\mathsf{source\_h}).$$

The predicate update function updates predicates $\mathsf{source\_j}$ and $\mathsf{target\_j}$ to $\mathsf{source\_(j-1)}$ and $\mathsf{target\_(j+1)}$, resp. The initial distribution update function updates $\eta\_\mathsf{source\_j}$ to $\eta\_\mathsf{source\_(j-1)}$. Then, the j-th task instance

$$\mathsf{R}_j = ((\texttt{achieve}\ (\mathsf{target\_j})); (\texttt{achieve}\ (\mathsf{source\_(h-j-1)}))), \eta\_\mathsf{source\_h-j}).$$

Motivating Example #2. Choice tasks from Figure 2 is an inductive task that updates the initial state distribution, by shifting to the right by constant units for adjacent task instances. They are stack of l levels, where each task R_k requires reaching a goal $goal_i$ while avoiding the obstacle obs, either through the (sub)goal g_{i1} or g_{i2},

$$\begin{aligned}&(\texttt{achieve}\ (\texttt{reach}\ (g_{i1})\ \texttt{or}\ \texttt{reach}\ (g_{i2}));\\ &\quad \texttt{achieve}\ (\texttt{reach}\ (goal_i)))^{l}\\ &\texttt{ensuring}\ (\texttt{avoid}\ (obs))\end{aligned}$$

where, $1 \leq i \leq l$. We use the superscript l to indicate that the enclosed specification is repeated l times. Figure 2a illustrates a task with $l = 1$ and Figure 9c illustrates a task with $l = 2$. The update functions for the initial distribution is defined as $\mathsf{update\_init}(\eta(s)) = \eta(s + (c_1, 0))$, where $c_1 = 1$ unit.

Lemma 1. *For an inductive task* R*, let* $\mathcal{G}_i$ *be the abstract graph of the specification of the i-th task instance* R_i*. Then, all the* $\mathcal{G}_i$*s share a common DAG structure with the same initial and final vertices.*

Proof. The proof follows from the construction of abstract graphs in [19].

This lemma asserts that all task instances within an inductive task have identical logical structures. They differ only in the low-level details of the abstract graph.

4 Generalizable RL for Inductive Tasks.

We define the problem of learning generalizable policies for an inductive task by learning a *policy generator*. The *policy generator* for an inductive task R is a function $\mathbb{G} : \mathsf{R} \to \Pi$, where Π is the set of all policies in the MDP. E.g., the policy generator for tower-destacking from Figure 1 maps the j-th task instance to the policy that displaces the source's $(\mathsf{h} - \mathsf{j})$-th block to the target's j-th block, then returns to the source's $(\mathsf{h} - \mathsf{j} - 1)$-th block by manipulating the motor controls θ_1, θ_2. Note these policies are different for each task instance R_j.

Definition 3 (Learning a Policy Generator). *Given an MDP with unknown transitions, an* inductive task R *and a set of a* training task instances Train *s.t. the base task* $\mathsf{R}_0 \in \mathsf{Train}$*, the problem of* generalizable RL *is to learn a* policy generator $\mathbb{G}^* : \mathsf{R} \to \Pi$ *such that*

$$\mathbb{G}^* \in \arg\max_{\mathbb{G}} \frac{1}{|\mathsf{Train}|} \cdot \sum_{\mathsf{R}_j \in \mathsf{Train}} \Pr_{s_0 \sim \eta_j \zeta \sim \mathcal{D}_{\pi_j, s_0}} [\zeta \models \phi_j, \eta_j] \text{ where the policy } \pi_j = \mathbb{G}^*(\mathsf{R}_j)$$

Then, $\pi_j = \mathbb{G}^*(\mathsf{R}_j)$ *for all* $j \in \mathbb{N}$.

I.e., the policy generator optimizes the policies for all training task instances simultaneously, in an attempt to *generalize*, so as to also derive policies for all task instances not present in Train.

4.1 Learning Policy Generator

We present an overview of our approach to learning a policy generator. Learning a higher-order function such as the policy generator is difficult. To make learning a policy generator feasible, we (a) assume *inductive relations between policies* of task instances that are inductively related, (b) leverage similarity between the structure of inductive tasks (Lemma 1), and (c) leverage compositionality of SPECTRL specifications [19].

We leverage the inductive nature of the inductive tasks to learn the policy generator. We base our work on the following hypothesis: *As two adjacent task instances are related by an inductive relation, there may also exist an inductive relation over the corresponding policies of these tasks.* However, this may not hold for certain tasks (eg. Figure 2). We attempt to overcome this with *compositonality*: instead of learning an inductive policy for the whole task, we divide the task into *subtasks* via the abstract graph, where each edge in the abstract graph corresponds to a subtask.

[19] ensures that a policy for a task instance R_i is given by a path policy in its abstract graph $\mathcal{G}_i$. Lemma 1 informs that the DAG structure of all graphs $\mathcal{G}_i$ are identical, say $\mathcal{G}$. Hence, the policy generator can be viewed as a map from task instances to path policies from initial to final vertex in the same graph $\mathcal{G}$ (with appropriate instantiation for edge policies in each task instance). Hence, we learn an inductive relation between the corresponding edge policies of the abstract graphs. We formulate the problem to learn such inductive relations in Section 4.1.

Last but not least, the edge policies obtained from the inductive relation will result in multiple path policies for each task instance. We are interested in the policy generator to choose the optimal path policy for each task instance. We ensure this by incorporating *guards* along vertices in the common DAG $\mathcal{G}$ that route each task instance along the optimal path in the DAG (Section 4.1).

Learning an Inductive Relation on Edges. This section defines an inductive relation between corresponding edges of the abstract graphs of an inductive task and formulates our approach to learn neural inductive relations.

Let $e = u \to v$ be an edge in the common DAG $\mathcal{G}$ of an inductive task. Let π_i denote the edge policy for the i-th task on the edge e in $\mathcal{G}_i$.

Then, an *inductive relation* between these policies is a function $\Omega : \Pi \to \Pi$ s.t. $\pi_{i+1} = \Omega(\pi_i)$. Thus, given the edge policy π_0 in the base task, the inductive relation Ω can be inductively "unrolled" to construct the edge policy for any instance R_i of an inductive task R. That is,

$$\pi_i = \Omega(\pi_{i-1}) = \Omega(\Omega(\pi_{i-2})) = \cdots = \Omega^i(\pi_0)$$

where Ω^i composes Ω with itself i times.

As learning the inductive relation Ω is difficult, we resort to *polynomial approximation*: we approximate the inductive relation Ω over the policies as an m-degree polynomial. Polynomial approximations are interesting as any function

can be approximated as a polynomial up to an arbitrary precision using the Taylor expansion. This reduces learning Ω on edges to inferring the κ-coefficients $(\kappa_m, \cdots, \kappa_0)$ of an m-degree κ-polynomial. Details below:

Neural Policies. If the policy for the i-th task instance $\pi_i \in \Pi$ is implemented by a neural network with parameter vector $[\pi_i]$, then the m-degree polynomial inductive relation is given by

$$[\pi_{i+1}] = \kappa_m \odot [\pi_i]^m + \kappa_{m-1} \odot [\pi_i]^{m-1} + \cdots + \kappa_0 \tag{1}$$

where, the polynomial coefficients, κ_i, are vectors with the same dimension as $[\pi_i]$; the $\odot$ operator is the Hadamard product (element-wise multiplication) of the coefficient vectors κ_i with the parameter vector (weights and biases) of the policy network π_i, and '+' is element-wise vector addition. Then as described earlier, with π_0 as the base policy with parameters $[\pi_0]$, the inductive relation Ω can be inductively "unrolled" to construct the policy network for π_i. Hence, in this case, we attempt to learn an inductive relation between the parameter vectors of the policy (neural) network of task instances.

Learning the Policy Generator. Next, we describe a policy generator on the common DAG $\mathcal{G}$ between all task instances in R. Given the inductive relation and base policy for every edge in $\mathcal{G}$, our goal is to describe a mapping for task instances in R to a path policy in $\mathcal{G}$, as per the edge policies inferred by Equation 1.

Every path from the initial to the final vertex corresponds to a path policy for the i-th task. Elaborating further, for degree m, let $\kappa^e = (\kappa^e_m, \cdots, \kappa^e_0)$ denote the κ-coefficients on the edge $e \in \mathcal{G}$. Let $\rho = e_1 \cdots e_k$ be a path from the initial to a final vertex. Then, a policy for a task R_i is given by the path policy $\pi_i^{e_1} \cdots \pi_i^{e_k}$ where $[\pi_i^{e_j}] = \Omega^i[\pi_0^{e_j}]$. This requires *selection* of a path policy for each R_i.

We assign *guards* at vertices with multiple outgoing edges in $\mathcal{G}$ such that each vertex routes task instances to a unique outgoing edge, ensuring a unique path for every task instance from initial to a final vertex. Formally, a guard in a vertex maps task instances to the outgoing edges from the vertex.

Then, the policy generator for an abstract graph executes as follows: Given a task instance R_i, it uses the guard conditions to determine its unique path from the initial to a final vertex. It returns the path policy along this path as described above. For example, Figure 2a has two possible paths: via g_1 or g_2. We learn a guard, $(i \leq 4)$, that resolves this branching decision at the init node: a task like R_2 would select pass via g_1 while R_6 will via g_2.

Hence, learning a policy generator for a DAG entails learning the $(m+1)$ κ-coefficients of the m-degree κ-polynomial and a base policy for every edge, along with guard conditions for all vertices with multiple outgoing edges.

5 Algorithm

Algorithm 1 (GenRL) takes as input an inductive task R, the degree m of the κ-polynomial, and a finite set of training task instances Train (we assume $0 \in$ Train)

and outputs a *policy generator* for R. As described above, this entails learning a base policy and the $(m+1)$ κ-coefficients for edges and guard conditions for vertices in the common DAG $\mathcal{G}$ (with initial vertex u_0) of the inductive task.

GenRL operates in two phases: (1) learn κ-coefficients and base policy for all edges, and (2) learn guard conditions at vertices with multiple outgoing edges.

In the first phase, GenRL traverses the vertices in $\mathcal{G}$ in topological order. While processing a vertex u, we record the *success probability* of the best probability path from the initial vertex u_0 to u for the i-th task instance in $P(u,i)$. We also record $\mathsf{bestIn}(u,i)$ to be the set of incoming vertices to u that are along a best probability path from u_0 to u for the task instance $i \in \mathsf{Train}$. Then,

$$P(u,i) = \begin{cases} 1 & \text{if } u = u_0 \\ \max\left\{P(w,i) \cdot p_i^{w \to u} \mid w \to u \in \mathsf{InEdges}(u)\right\} & \text{if } u \neq u_0 \end{cases}$$

$$\mathsf{bestIn}(u,i) = \begin{cases} \emptyset & \text{if } u = u_0 \\ \arg\max_w \left\{P(w,i) \cdot p_i^{w \to u} \mid w \to u \in \mathsf{InEdges}(u)\right\} & \text{if } u \neq u_0 \end{cases}$$

where $p_i^{w \to u}$ is the estimated success probability of edge policy of i-th task instance on edge $w \to u$.

Next, we induce a state distribution η_i^u on vertex u for all task instances $i \in \mathsf{Train}$. $\eta_i^{u_0}$ is given by the initial distribution of the task instance R_i. For all other vertices $u \neq u_0$, the state distributions are induced along the *best probability path* from u_0 to u. To this end, η_i^u is induced from states in $\mathsf{bestIn}(u,i)$ using the leaned edge policies along these incoming edges.

Finally, for all outgoing edges $e = u \to v$, we learn the base policy and $(m+1)$ κ-coefficients. The base policy π_0^e is learned as a neural-network policy using standard RL such that π_0^e maximizes the satisfaction of the edge e for the 0-th task instance. I.e., the rewards are designed to encourage π_0^e to safely transition from an MDP state in u to an MDP state in v for the 0-th task instance.

The κ-coefficients are learned using an adaptation of the ARS (Augmented Random Search) (see Appendix B.1). The κ-coefficients capture an inductive relation between the parameters of the policy networks of adjacent task instances; $[\pi_i^e]$ is the parameter vector for the policy network corresponding to the i-th task instance. Let π_i^e be obtained by unrolling the κ-polynomial on the base policy parameters for all $i \in \mathsf{Train}$. Taking rewards of π_i^e to be based on satisfaction of the edge for the i-th task instance (as done for the base policy), κ-polynomials are trained to optimize the softmin of the rewards over all training task instances.

The second phase learns the guard conditions (see Appendix B): In addition to ensuring the uniqueness of the path, we require that the guards choose an edge such that the resulting path policy has a high probability of satisfaction. To this end, for each edge e, we create a set S^e of task instances such that the e appears on a best probability path to a final vertex for those task instances, using backward DAG traversal and bestIn. Finally, the guard on a vertex u is learned as a multi-task classifier with (a) outgoing edges as the class labels, (b) task instance indices as features, and (c) the dataset consists of data points (i,e) s.t. $i \in S^e$ for all outgoing edges e from u.

6 Empirical Evaluation

We evaluate GenRL[3] across diverse environments, demonstrating superior generalization and sample efficiency compared to state-of-the-art approaches. Through experiments spanning navigation, long-horizon tasks, complex decision-making, and control benchmarks, we show that explicitly modeling inductive relationships between tasks provides a fundamentally more effective approach to generalization.

6.1 Experimental Setup

Evaluation Methodology. To rigorously evaluate generalization capabilities, each experiment involves training on a fixed set of task instances (Train) and testing on unseen task instances (Unseen) from the same inductive task family. We estimate the probability of success for each task based on 1000 rollouts. A policy π_i is considered successful on task instance $\mathsf{R}_i : (\phi_i, \eta_i)$ if the rollouts ζ satisfy the specification with probability above δ, specifically $\Pr_{\zeta \sim \mathcal{D}_{\pi_i}}[\zeta \models \phi_i, \eta_i] > \delta$, where we set $\delta = 0.9$. All reported results represent the median of five independent runs with different random seeds. For fair comparison, all methods are evaluated on identical Train and Unseen sets.

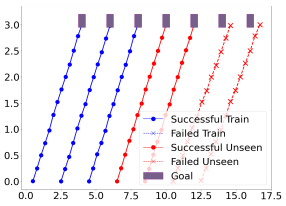
In all experiments, we train using a 1-degree κ-polynomial approximation of the inductive relationship (see Appendix G for analysis of polynomial degree choice). All experiments run on a cluster with Intel Xeon Gold 6226 CPUs (2.7 GHz, 24 cores per node) and 192GB RAM per node.

6.2 Baselines and Comparisons

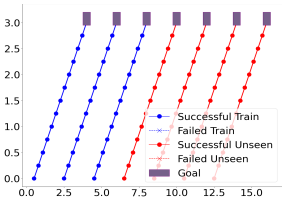
Baselines are state-of-the-art generalizable RL approaches and GenRL ablations:

- Generalizable RL algorithms across three categories:
  - *Inductive Generalization algorithms.* PSMP [16] leverages inductive structures between tasks but operates as a planning approach with known transition probabilities;
  - *Meta-Learning algorithms.* MAML-Reinforce [11] and VariBAD-A2C [42], which enable rapid adaptation through task-specific representations;
  - *Goal-Conditioned algorithms.* C-Learning [29] and HER-DDPG [4] enable generalization by incorporating goal-states directly into the policy input.
- Ablations of GenRL to analyze component contributions:
  - *Augmented Random Search (ARS) [28]:* This ablation uses the standard ARS algorithm, learning a single policy for all tasks in Train. Unlike GenRL, which trains an inductive policy generator that adapts to different tasks, ARS relies on a single reward signal rather than aggregating multiple task-specific rewards.

[3] Complete codebase of GenRL and the experimental setup is available at https://anonymous.4open.science/r/GenRL_Zenith-7EEB/; details in Appendix A

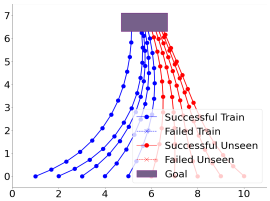


(a) GenRL

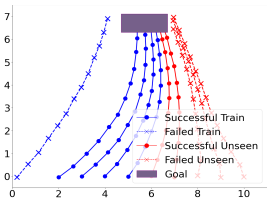


(b) PSMP

Fig. 3: Moving initial and goal distributions

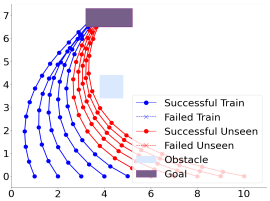


(a) GenRL

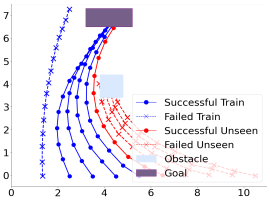


(b) ARS+RA

Fig. 4: Moving initial distribution, stationary goal



(a) GenRL



(b) ARS+RA

Fig. 5: Moving initial distribution, stationary goal, with obstacle

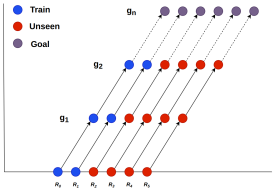
- *ARS + Reward Aggregation (ARS+RA):* This ablation incorporates our reward aggregation mechanism (softmin function across tasks) but still learns a single policy instead of generating task-specific policies.
- *ARS + Goal Conditioning (ARS+GC):* This ablation extends ARS+RA by providing the task index as additional input to the policy, enabling differentiation between tasks while maintaining a single policy.

6.3 Results and Analysis

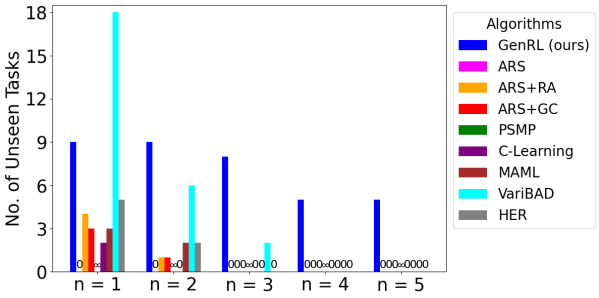
Simple Reachability Tasks We begin with simple reachability tasks in a 2D Cartesian plane. The inductive tasks (Figures 3–5) require navigating from initial positions (blue and red dots) to target positions (grey boxes) while avoiding obstacles when present. Task instances are generated by shifting either only the initial position (Figures 4, 5) or both initial and goal positions (Figure 3).

The figures show representative trajectories for training tasks (blue) and unseen tasks (red). Despite the apparent simplicity, most baselines struggle with generalization. While PSMP performs adequately when task variations follow simple shifts (Figure 3), it fails with more complex adaptations (Figures 4, 5). ARS+RA shows stronger performance in controlled variations but lacks consistent generalization across broader conditions.

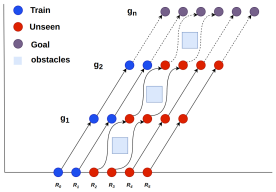
In contrast, GenRL demonstrates substantially superior generalization by producing custom trajectories adapted to each specific task instance rather than applying nearly identical paths across all variations. For the simplest case (Figure 3), GenRL successfully completes up to 99 unseen tasks after 400



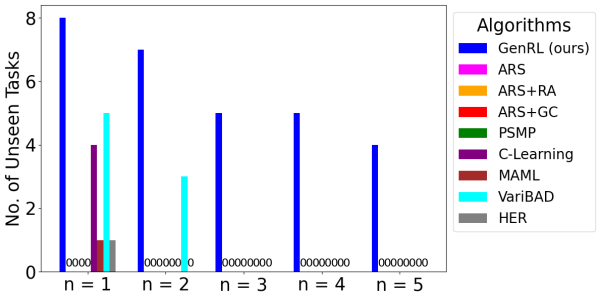
(a) N-Reachability without obstacles (Illustration)



(b) N-Reachability without obstacles (results)



(c) N-Reachability with obstacles (Illustration)



(d) N-Reachability with obstacles (results)

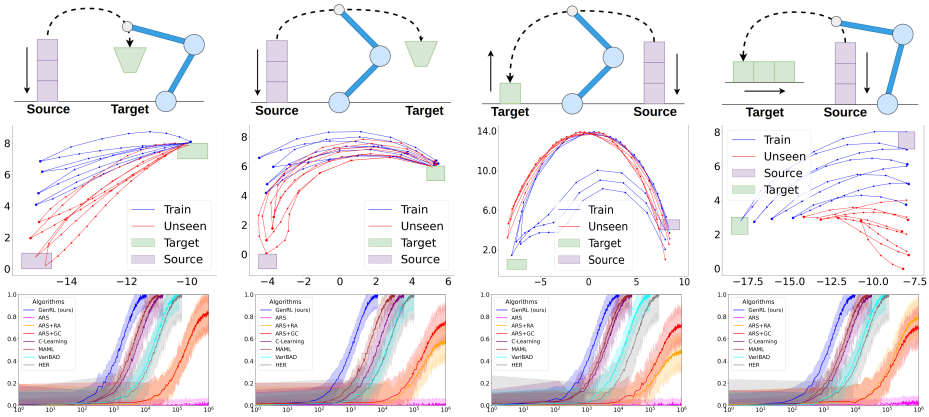
Fig. 6: N-Reachability Tasks: (a) and (c) illustrate the task environments without and with obstacles, respectively. (b) and (d) compare the number of successful unseen tasks for these environments.

iterations—a remarkable 1650% generalization rate relative to the training set. Even with obstacles (Figure 5), GenRL maintains impressive generalization with 22 successful unseen tasks (367% generalization rate). Successful trajectories (solid lines with •) vastly outnumber failures (dotted lines with ×) for GenRL even on unseen tasks, confirming its robust adaptation capabilities. More detailed results can be found in Appendix D.2.

Long-Horizon Tasks To evaluate scalability to longer horizons, we designed two classes of long-horizon tasks:

N-Reachability Tasks. These tasks (Figures 6a and 6c) extend simple reachability by requiring the agent to visit N intermediate points sequentially. Both the initial distribution and goal positions shift inductively across task instances.

Figures 6b and 6d show generalization performance as N increases. GenRL maintains consistently high performance across all horizon lengths, successfully generalizing to 5 unseen tasks even in the most complex scenarios ($N = 5$)—an 83%



(a) (b) (c) (d)

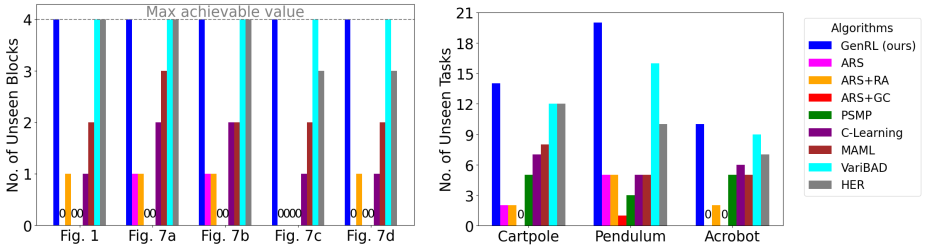
Fig. 7: Tower-destacking benchmarks on Reacher Environment: Task illustrations, trajectories, and learning curves: (a) Pick and Drop: Same side, (b) Pick and Drop: Opposite side, (c) Pick and Vertical Stack: Opposite side, (d) Pick and Horizontal Stack: Same side. Learning Curve: x-axis denotes the number of samples (steps) and y-axis denotes the average of the estimated probability of success of all tasks in Train. Results are averaged over 5 runs with the cloud indicating the minimum and maximum.

generalization rate. While VariBAD achieves competitive results for $N = 1$, its performance deteriorates dramatically as horizon length increases, highlighting its difficulty with longer-horizon tasks. Other algorithms show poor generalization across the board, with performance declining rapidly as N increases.

Tower-Destacking. We evaluated several variations of a tower-destacking task using a robotic arm in the Reacher environment (Figures 1 and 7). The source tower contains eight blocks, with algorithms trained on the top four blocks and tested on the remaining four.

Figure 8a shows that GenRL achieves generalization performance comparable to sophisticated methods like VariBAD and HER. However, the learning curves in Figure 7 reveal a critical advantage: GenRL reaches optimal performance with an order of magnitude fewer samples than these alternatives. While GenRL converges after approximately 10^4 environment steps, VariBAD requires 10^5 steps, and HER-DDPG needs over 10^6 steps to achieve similar performance. This remarkable sample efficiency demonstrates the power of explicitly modeling inductive relationships rather than relying on implicit learning through neural networks. Detailed results can be found in Appendices D.3, D.4, and E.

Complex Decision-Making Tasks Some of the most challenging scenarios involve optimal decision-making where the agent must choose between alternative paths based on task parameters. We evaluated GenRL on three "choice" tasks



(a) Reacher Benchmarks (b) Classical control benchmarks

Fig. 8: Comparison of the number of successful Unseen task instances. (a) Performance on Reacher benchmarks. (b) Performance on classical Control benchmarks.

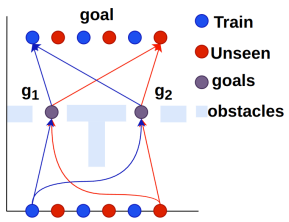
Benchmark	Successful Train	Successful Unseen	Learned Guard Predicate
Figure 2	**All**	7	$(i \leq 4)$
Figure 9a	**All**	5	$(i \leq 4)$
Figure 9c	**All**	5	$(i \leq 4)$, $(i \leq 4)$

Table 1: Choice benchmarks: No. of successful unseen tasks for the Car2D Choice benchmarks. **Successful Train** indicates if all training tasks were completed successfully ($|\mathsf{Train}| = 6$), while **Successful Unseen** reports the number of successful instances on unseen tasks. **Learned Guard Predicate** represents the decision index where the agent must choose the optimal goal. For example, when $i \leq 4$, for task R_i where $i \leq 4$, goal g_1 is chosen; otherwise, goal g_2 is chosen. In more complex benchmarks (e.g., Figure 9c with 2 levels), multiple predicates may emerge.

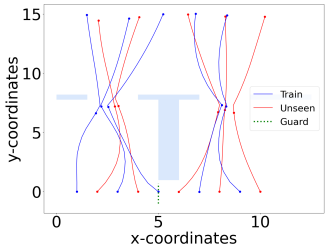
of increasing complexity: (a). Moving initial point with fixed goal (Figure 2) (b). Moving initial point and moving goal (Figure 9a) (c). Two-level choice with moving initial point and moving goal (Figure 9c). These tasks require the agent to select between alternative subgoals (g_1 or g_2) based on the specific task instance. GenRL's unique ability to learn guard conditions enables optimal branching decisions. For example, in Figure 2, GenRL learns the guard predicate $(i \leq 4)$, directing tasks with index $i \leq 4$ through the first subgoal and tasks with $i > 4$ through the second subgoal.

Table 1 shows that GenRL successfully generalizes to 5-7 unseen tasks across these choice benchmarks, achieving up to 117% generalization on the most basic choice task (Figure 2) and 84% generalization on more complex choice variants. Remarkably, no other algorithm demonstrates any meaningful generalization on these tasks, highlighting GenRL's distinctive capability to model complex decision boundaries. Detailed descriptions can be found in Appendix D.5.

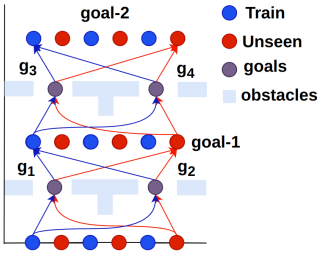
Classical Control Benchmarks To verify GenRL's applicability beyond navigation tasks, we evaluated it on classic OpenAI Gym control benchmarks: Pendulum,



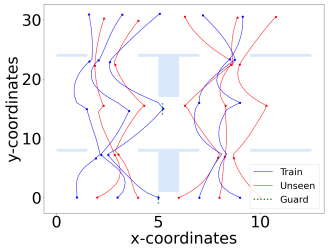
(a) Choice with moving goal - Illustration.



(b) Choice with moving goal - Trajectory.



(c) Two levels of choice with moving goal - Illustration.



(d) Two levels of choice with moving goal - Trajectory.

Fig. 9: Choice benchmarks task illustration and corresponding trajectories: (a) and (b) illustrate the benchmark, while (c) and (d) show the agent's trajectory in the respective benchmarks.

Acrobot, and Cartpole. These environments feature inductive variations in physical parameters (mass, length) rather than task specifications.

Figure 8b shows that GenRL achieves superior generalization across all three environments, with particularly impressive results in Pendulum (20 successful unseen tasks, 333% generalization) and Cartpole (14 successful unseen tasks, 233% generalization). Even in the challenging Acrobot environment, GenRL successfully generalizes to 10 unseen tasks (167% generalization) with sufficient training iterations, while competing methods plateau at much lower levels of generalization. This demonstrates that GenRL can effectively model inductive relationships in physical parameters and transfer knowledge between related control tasks. Detailed descriptions are available in Appendix F.

6.4 Key Findings and Analysis

Superior Performance and Stability Across Complexity Scales. GenRL consistently outperforms all baselines across diverse specifications, particularly as task complexity increases. While baseline methods exhibit diminishing returns—evident in N-Reachability tasks where VariBAD's performance drops dramatically from N=1

to N=5, and in choice tasks where baselines fail to generalize at all—GenRL maintains stable performance even for longer horizons and complex branching tasks. This stability, coupled with remarkable sample efficiency (requiring only 10^4 environment steps compared to 10^5-10^6 for policy gradient methods), demonstrates that explicitly modeling inductive relationships is fundamentally more effective than implicit meta-learning or goal-conditioning approaches. GenRL achieves this despite using the simpler Augmented Random Search algorithm without gradients, yet matches or exceeds the performance of sophisticated methods like VariBAD and HER.

This dramatic efficiency advantage demonstrates the power of leveraging logical specifications and inductive structure. For tasks with natural inductive relationships, explicitly modeling these relationships proves far more effective than attempting to learn them implicitly through meta-learning or policy gradients.

Policy Generator is key to GenRL*'s superior performance.* Our ablation studies with ARS, ARS+RA, and ARS+GC consistently show these variants underperforming compared to the full GenRL framework. The base ARS algorithm has no mechanism to generalize over tasks, resulting in poor performance. Although incorporating Reward Aggregation (ARS+RA) and Goal Conditioning (ARS+GC) provides marginal improvements, these variants still fall far short of GenRL's capabilities.

When ARS leverages GenRL's inductive policy generator framework, it not only overcomes its high sample complexity but significantly enhances performance. This integration enables GenRL to outperform even sophisticated policy gradient methods, demonstrating the substantial advantages of our approach. GenRL's ability to efficiently generate adapted policies for diverse task instances—rather than learning or fine-tuning individual policies—underlies its exceptional sample efficiency and performance.

7 Related Work

Specification-Guided RL. Recent years have seen an emergence of RL from logical specifications [1, 2, 5, 7, 9, 12–15, 17, 19, 20, 25, 26, 33, 38, 40]. Here, the task is expressed using high-level logical specifications rather than as low-level rewards. Logic specifications have received traction due to their ease in expressing complex long-horizon tasks and ability to efficiently scale learning. Prior works have focused primarily on scalability to long-horizon tasks and theoretical guarantees. Ours is the first work to leverage logical specifications specifically for generalization.

Zero-shot generalization. Zero-shot generalization relates to multi-task learning and skill transfer, distilling transferable skills from seen tasks to generalize to unseen ones [22, 30–32]. With logical specifications, existing approaches learn policies for sub-specifications and generalize to their combinations [23, 24, 27, 35, 37]. Our problem is orthogonal: in prior work, the predicate set remains constant while only specifications change. In our setting, both predicates and

environment distributions vary between training and unseen tasks. [21] considers changing distributions but with fixed predicates. Reward-based generalizable RL has been explored in [43],[34]. In meta-learning, algorithms like MAML[11] and VariBAD[42] enable rapid adaptation to new tasks by adjusting task-specific parameters and representations. Goal-conditioned algorithms like C-Learning[29] and HER[4] contribute by conditioning policies on desired outcomes, enhancing adaptation to new goals without additional training.

Inductive/Programmatic Approaches to Generalization. Closest to our work is PSMP [16], which learns inductive policies in the planning setting (known MDP) rather than RL (unknown MDP). Despite this advantage, PSMP cannot adapt to different task instances, as it learns a single policy for all instances. Our approach learns a higher-order policy generator that produces specialized policies for each task instance. Programmatic/logic-based policy representations generally demonstrate better generalizability than neural network policies [6, 8, 36, 41]. Non-programmatic policy sketches have also been explored [3]. Our work differs by exploiting the natural inductiveness in task specifications to extract inductive relations for policies and learn a higher-order policy generator.

8 Concluding remarks

While current advances in generalizable RL have focused primarily on making agents more adaptable through sophisticated architectures and training procedures, our work suggests an alternative path forward - one that leverages the inherent structure in specifications to enable systematic generalization. By making this structure explicit through formal specifications and inductive relationships, we not only achieve better generalization but also achieve better sample complexity by an order of magnitude. In doing so, we outperform several mature policy-gradient based state-of-the-art tools for generalization.

Currently, GenRL performs effectively in environments with lower-dimensional action and state spaces. However, its scalability to more complex environments with higher-dimensional spaces remains a challenge. Future work will focus on enhancing the algorithm's capability to handle these more complex scenarios. Additionally, while defining tasks via logical specifications is generally easier than specifying rewards, it still requires considerable effort to design these specifications accurately. To address these, future research will aim at developing more streamlined and user-friendly methods for task specification to make the specification process as lightweight as possible. This will help broaden the applicability of GenRL or any specification-guided learning to a wider range of tasks and environments. Even though the logic and formulation behind our research are principally motivated by a foundational hypothesis and empirically validated for its performance and results, we still not have investigated the possibilities of providing theoretical guarantees and this is also something our future work will focus on.

Bibliography

[1] Aksaray, D., Jones, A., Kong, Z., Schwager, M., Belta, C.: Q-learning for robust satisfaction of signal temporal logic specifications. In: Conference on Decision and Control (CDC). pp. 6565–6570. IEEE (2016)

[2] Alur, R., Bansal, S., Bastani, O., Jothimurugan, K.: A framework for transforming specifications in reinforcement learning. In: Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday, pp. 604–624. Springer (2022)

[3] Andreas, J., Klein, D., Levine, S.: Modular multitask reinforcement learning with policy sketches. In: International conference on machine learning. pp. 166–175. PMLR (2017)

[4] Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., Zaremba, W.: Hindsight experience replay. Advances in Neural Information Processing Systems **30** (2017)

[5] Bansal, S.: Specification-guided reinforcement learning. In: International Static Analysis Symposium. pp. 3–9. Springer (2022)

[6] Bastani, O., Pu, Y., Solar-Lezama, A.: Verifiable reinforcement learning via policy extraction. Advances in neural information processing systems **31** (2018)

[7] Brafman, R., De Giacomo, G., Patrizi, F.: LTLf/LDLf non-markovian rewards. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 32 (2018)

[8] Cao, Y., Li, Z., Yang, T., Zhang, H., Zheng, Y., Li, Y., Hao, J., Liu, Y.: GALOIS: boosting deep reinforcement learning via generalizable logic synthesis. Advances in Neural Information Processing Systems **35**, 19930–19943 (2022)

[9] De Giacomo, G., Iocchi, L., Favorito, M., Patrizi, F.: Foundations for restraining bolts: Reinforcement learning with LTLf/LDLf restraining specifications. In: Proceedings of the International Conference on Automated Planning and Scheduling. vol. 29, pp. 128–136 (2019)

[10] Dohmen, T., Perez, M., Somenzi, F., Trivedi, A.: Regular reinforcement learning. In: Gurfinkel, A., Ganesh, V. (eds.) Computer Aided Verification. pp. 184–208. Springer Nature Switzerland, Cham (2024)

[11] Finn, C., Abbeel, P., Levine, S.: Model-agnostic meta-learning for fast adaptation of deep networks. In: International Conference on Machine Learning. pp. 1126–1135. PMLR (2017)

[12] Hahn, E.M., Perez, M., Schewe, S., Somenzi, F., Trivedi, A., Wojtczak, D.: Omega-Regular Objectives in Model-Free Reinforcement Learning. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 395–412 (2019)

[13] Hasanbeig, M., Kantaros, Y., Abate, A., Kroening, D., Pappas, G.J., Lee, I.: Reinforcement learning for temporal logic control synthesis with probabilistic

satisfaction guarantees. In: Conference on Decision and Control (CDC). pp. 5338–5343 (2019)
[14] Hasanbeig, M., Abate, A., Kroening, D.: Logically-constrained reinforcement learning. arXiv preprint arXiv:1801.08099 (2018)
[15] Icarte, R.T., Klassen, T., Valenzano, R., McIlraith, S.: Using reward machines for high-level task specification and decomposition in reinforcement learning. In: International Conference on Machine Learning. pp. 2107–2116. PMLR (2018)
[16] Inala, J.P., Bastani, O., Tavares, Z., Solar-Lezama, A.: Synthesizing Programmatic Policies that Inductively Generalize. In: Internation Conference on Learning Representations (2020)
[17] Jiang, Y., Bharadwaj, S., Wu, B., Shah, R., Topcu, U., Stone, P.: Temporal-Logic-Based Reward Shaping for Continuing Reinforcement Learning Tasks. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 35, pp. 7995–8003 (2021)
[18] Jothimurugan, K., Alur, R., Bastani, O.: A composable specification language for reinforcement learning tasks. Advances in Neural Information Processing Systems **32** (2019)
[19] Jothimurugan, K., Bansal, S., Bastani, O., Alur, R.: Compositional reinforcement learning from logical specifications. Advances in Neural Information Processing Systems **34**, 10026–10039 (2021)
[20] Jothimurugan, K., Bansal, S., Bastani, O., Alur, R.: Specification-guided learning of Nash equilibria with high social welfare. In: International Conference on Computer Aided Verification. pp. 343–363. Springer (2022)
[21] Jothimurugan, K., Hsu, S., Bastani, O., Alur, R.: Robust Subtask Learning for Compositional Generalization. In: International Conference on Machine Learning (2023)
[22] Kirk, R., Zhang, A., Grefenstette, E., Rocktäschel, T.: A survey of zero-shot generalisation in deep reinforcement learning. Journal of Artificial Intelligence Research **76**, 201–264 (2023)
[23] Kuo, Y.L., Katz, B., Barbu, A.: Encoding formulas as deep networks: Reinforcement learning for zero-shot execution of LTL formulas. In: 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 5604–5610. IEEE (2020)
[24] León, B.G., Shanahan, M., Belardinelli, F.: Systematic generalisation through task temporal logic and deep reinforcement learning. arXiv preprint arXiv:2006.08767 (2020)
[25] Li, X., Vasile, C.I., Belta, C.: Reinforcement learning with temporal logic rewards. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 3834–3839. IEEE (2017)
[26] Littman, M.L., Topcu, U., Fu, J., Isbell, C., Wen, M., MacGlashan, J.: Environment-Independent Task Specifications via GLTL. arXiv preprint arXiv:1704.04341 (2017)
[27] Liu, J., Shah, A., Rosen, E., Jia, M., Konidaris, G., Tellex, S.: Skill Transfer for Temporal Task Specification. In: CoRL 2023 Workshop on Learning Effective Abstractions for Planning (LEAP) (2023)

[28] Mania, H., Guy, A., Recht, B.: Simple random search of static linear policies is competitive for reinforcement learning. In: Advances in Neural Information Processing Systems. pp. 1805–1814 (2018)
[29] Naderian, P., Loaiza-Ganem, G., Braviner, H.J., Caterini, A.L., Cresswell, J.C., Li, T., Garg, A.: C-learning: Horizon-aware cumulative accessibility estimation. International Conference on Learning Representations (2021)
[30] Oh, J., Singh, S., Lee, H., Kohli, P.: Zero-shot task generalization with multi-task deep reinforcement learning. In: International Conference on Machine Learning. pp. 2661–2670. PMLR (2017)
[31] Sodhani, S., Zhang, A., Pineau, J.: Multi-task reinforcement learning with context-based representations. In: International Conference on Machine Learning. pp. 9767–9779. PMLR (2021)
[32] Sohn, S., Oh, J., Lee, H.: Hierarchical reinforcement learning for zero-shot generalization with subtask dependencies. Advances in neural information processing systems **31** (2018)
[33] Svoboda, J., Bansal, S., Chatterjee, K.: Reinforcement learning from reachability specifications: Pac guarantees with expected conditional distance. In: Forty-first International Conference on Machine Learning (2024)
[34] Taiga, A.A., Agarwal, R., Farebrother, J., Courville, A., Bellemare, M.G.: Investigating Multi-task Pretraining and Generalization in Reinforcement Learning. In: The Eleventh International Conference on Learning Representations (2023)
[35] Vaezipoor, P., Li, A.C., Icarte, R.A.T., Mcilraith, S.A.: LTL2ACTION: Generalizing LTL Instructions for Multi-task RL. In: International Conference on Machine Learning. pp. 10497–10508. PMLR (2021)
[36] Verma, A., Murali, V., Singh, R., Kohli, P., Chaudhuri, S.: Programmatically interpretable reinforcement learning. In: International Conference on Machine Learning. pp. 5045–5054. PMLR (2018)
[37] Xu, D., Fekri, F.: Generalizing LTL Instructions via Future Dependent Options. arXiv preprint arXiv:2212.04576 (2022)
[38] Xu, Z., Topcu, U.: Transfer of temporal logic formulas in reinforcement learning. In: International Joint Conference on Artificial Intelligence. pp. 4010–4018 (7 2019)
[39] Yang, C., Littman, M., Carbin, M.: On the (in) tractability of reinforcement learning for ltl objectives. arXiv preprint arXiv:2111.12679 (2021)
[40] Yuan, L.Z., Hasanbeig, M., Abate, A., Kroening, D.: Modular deep reinforcement learning with temporal logic specifications. arXiv preprint arXiv:1909.11591 (2019)
[41] Zhu, H., Xiong, Z., Magill, S., Jagannathan, S.: An inductive synthesis framework for verifiable reinforcement learning. In: Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation. pp. 686–701 (2019)
[42] Zintgraf, L., Schulze, S., Lu, C., Feng, L., Igl, M., Shiarlis, K., Gal, Y., Hofmann, K., Whiteson, S.: Varibad: Variational bayes-adaptive deep rl via meta-learning. Journal of Machine Learning Research **22**(289), 1–39 (2021)
[43] Zisselman, E., Lavie, I., Soudry, D., Tamar, A.: Explore to Generalize in Zero-Shot RL. Advances in Neural Information Processing Systems **36** (2024)

Appendix

A Supplemental Material

The complete source code of GenRL tool along with our experimental setup has been made available at https://anonymous.4open.science/r/GenRL_Zenith-7EEB/. We provide comprehensive training and testing code for our experiments. In addition to the training and testing scripts, we include pre-trained models that allow users to generate rollouts and visualize the resulting trajectories. We have provided a `requirements.txt` within the artifact with detailed instructions.

To get started, please install the required dependencies:
`pip install -r requirements.txt`.

Training new models. To train the models, use the following command with the appropriate parameters:
`python -u -m spectrl.examples.car2d_dirl -n 0 -d car2d_k2/{name} -s {s} -h {h} -b 'j2_'`.

- For Towerstack with robotic arm experiments, set `s` from 0 to 5 and `h` to 8.
- For choice experiments, set `s` to 6 or 7 and `h` to 10.
- For classical control experiments, set `s` from 8 to 10 and `h` to 10.

Running the training script also executes the testing script. The codebase is set to test for only `n` number of unseen tasks. To test for a different number of task instances, change the `test_rounds` variable in the `spectrl/hierarchy/reachability.py` file. Replace `{any_name_of_your_choice}`, `{s}`, and `{h}` with your specific values. For example, to run a tower-stack with a robotic arm experiment with `s` set to 2 and `h` set to 8, use:
`python -u -m spectrl.examples.car2d_dirl -n 0 -d car2d_k2/experiment1 -s 2 -h 8 -b 'j2_'`.

Only visualizing rollouts from pre-trained models. If you only want to visualize the rollouts from the pre-trained models and not train new ones, modify the parameters in the `spectrl/examples/car2d_dirl.py` file: set `training` to `False` (line 521) and set `prepare_rollouts` to `True` (line 522), and run the same training commands given above as per your requirements by choosing the right file name. For example, `python -u -m spectrl.examples.car2d_dirl -n 0 -d car2d_k2/choice_test -s 6 -h 10 -b 'j2_'`.

B Algorithm Details

Algorithm 1 GenRL($\mathsf{R}, m, \mathsf{Train}$)

$\mathcal{G} \leftarrow \mathsf{CommonDAG}(\mathsf{R})$
while vertex $u \in \mathcal{G}$ is chosen in topological order **do**
  Compute $P(u,i), \mathsf{bestIn}(u,i)$ **for** all $i \in \mathsf{Train}$
  $\eta_i^u \leftarrow \mathsf{InduceDistrbution}(u, \mathsf{bestIn}(u,i))$ **for** all $i \in \mathsf{Train}$
  for edge $e = (u,v) \in \mathsf{OutEdges}(u)$ **do**
    $\pi_0^e \leftarrow \mathsf{LearnBasePolicy}(e, \eta_0^u)$
    $\kappa^e \leftarrow \mathsf{LearnKappaCoefficients}(e, m+1, \pi_0^e, \eta^u, \mathsf{Train})$
$\mathsf{Guard} \leftarrow \mathsf{LearnGuardConditions}(\mathcal{G}, \mathsf{bestIn})$
return κ^e, π_0^e for all edge and Guard for all vertices

B.1 Kappa Learning Algorithm (Algorithm 2 LearnKappaCoefficients)

Algorithm 2 LearnKappaCoefficients learns the kappa-coefficients using an ARS style algorithm with a softmin aggregator.

In more detail, vector $\kappa = [\kappa_0, \kappa_1, \ldots, \kappa_{m-1}]$ is initialized as a normal distribution vector, where m represents the number of elements in κ. SampleDelta samples perturbation vectors δ, conforming to the same dimensionality as κ. PerturbKappa creates perturbed kappa vectors $\kappa_{plus} = \kappa + (\delta_{scale} \cdot \delta)$ and $\kappa_{minus} = \kappa - (\delta_{scale} \cdot \delta)$. For each perturbed kappa vector, the equation 1 (KappaPolicy) generates policies by polynomially combining its elements with the base policy π_e^0. These generated policies are evaluated for each task $\mathsf{R}_i \in \mathsf{Train}$, with rewards accumulated in r_{plus} and r_{minus} for policies derived from κ_{plus} and κ_{minus} for each task, respectively. Score aggregates these rewards into collective performance scores R_{plus} and R_{minus} by computing $R_{plus} \leftarrow \mathit{softmin}(r_{plus})$ and similarly for R_{minus}. It then forms tuples $\delta_{samples}$, pairing each perturbation δ with its corresponding aggregate scores. DeltaUpdate computes a weighted average perturbation δ_κ from these samples, guiding the update of the kappa vector as $\kappa_{updated} \leftarrow \kappa + \delta_\kappa$. This iterative process of sampling, evaluating, and updating is continued until convergence of the kappa vector is reached, optimizing the policies for the specified tasks in Train. The reward function Reward based on the Euclidean distance between the agent's position and the goal position can be expressed as $\mathsf{Reward} = -\|\mathbf{p}_{\text{agent}} - \mathbf{p}_{\text{goal}}\|$ where $\mathbf{p}_{\text{agent}}$ represents the position vector of the agent, $\mathbf{p}_{\text{goal}}$ represents the position vector of the goal, $\|\cdot\|$ denotes the Euclidean norm (or Euclidean distance).

Overview Post-DAG traversal, Algorithm 3 conducts a reverse traversal to ascertain optimal tasks R_i for each edge e, with $i \in \mathsf{Train}$, guiding the task from the initial state u_0 to any one of the final states in F. The identified tasks for each task are stored in $\mathcal{D}_e$ where edge $e \in \mathcal{G}$. Subsequently, a dataset D is generated, on which a decision tree classifier is trained to yield a Guard. Guard directs the choice of edges for R, such that the likelihood of reaching final state $f \in F$ with maximal success likelihood.

$\mathcal{D}_e(e)$: Given a set of task indices $i \in \mathsf{Train}$ and $i \in \mathcal{D}_e(e)$, the most optimal path from the initial state to the final state for the task R_i goes through the edge e.

Algorithm 2 $\mathsf{LearnKappaCoefficients}(e, m, \pi_e^0, \Gamma_u, \mathsf{Train})$
Kappa Training using a modified Augmented Random Search

Initialize $\kappa(m)$
  where $m \leftarrow$ number of kappa in the polynomial template
while κ not converged **do**
  $\delta_{samples} \leftarrow \emptyset$
  for $s = 0$ to $n\_samples$ **do**
    $r_{plus} \leftarrow \emptyset$, $r_{minus} \leftarrow \emptyset$
    $\delta \leftarrow \mathsf{SampleDelta}(\kappa)$
    $\kappa_{plus} \leftarrow \mathsf{PerturbKappa}(\kappa, \delta, \delta_{scale})$
    $\kappa_{minus} \leftarrow \mathsf{PerturbKappa}(\kappa, \delta, -\delta_{scale})$
    for $k = 0$ to $|\mathsf{Train}|$ and task R_i where $i \in \mathsf{Train}$ **do**
      $\pi_{plus} \leftarrow \mathsf{KappaPolicy}(\kappa_{plus}, \pi_e^0, k, m)$
      $r_{plus}[k] \leftarrow \mathsf{Reward}(\pi_{plus}, \mathsf{R}_i)$
      $\pi_{minus} \leftarrow \mathsf{KappaPolicy}(\kappa_{minus}, \pi_e^0, k, m)$
      $r_{minus}[k] \leftarrow \mathsf{Reward}(\pi_{minus}, \mathsf{R}_i)$
    $R_{plus} \leftarrow \mathsf{Score}(r_{plus})$
    $R_{minus} \leftarrow \mathsf{Score}(r_{minus})$
    $\delta_{samples}[s] \leftarrow (\delta, R_{plus}, R_{minus})$
  $\delta_\kappa \leftarrow \mathsf{DeltaUpdate}(\delta_{samples})$
  Update $\kappa \leftarrow \mathsf{PerturbKappa}(\kappa, \delta_\kappa, 1)$
return κ

Details The Algorithm 3 commences by initializing the outgoing edges $O(v)$ for every vertex $v \in \mathcal{G}$ and populating a queue Q with the final states F. In its reverse traversal phase, the algorithm, for each vertex u dequeued from Q, examines the incoming vertices. For every incoming vertex v, it removes u from v's outgoing edges and for every task $i \in \mathsf{Train}$, it checks whether the vertex v is in $\mathsf{bestIn}(u, i)$ and if this condition is true, then the task index i is appended to $De(e)$ where edge $e = (v \to u)$. Then, if $O(v)$ becomes empty, meaning v has no more outgoing vertices to process, v is enqueued in Q for further processing. This iterative process continues until all vertices in $\mathcal{G}$ are traversed, ultimately determining the $\mathcal{D}_e$) for each edge, which identifies the optimal tasks for which each edge forms a part of the best path to any of the final states F from the initial state u_0.

Now, using $\mathcal{D}_e$, we establish decision boundaries for vertices u in $\mathcal{G}$ where $\mathsf{OutVertices}(u) > 1$, identifying vertices necessitating decision-making (choose the optimal outgoing edge for a particular task). For each such vertex u, the algorithm examines every edge $e = (u \to v)$, with $v \in \mathsf{OutVertices}(u)$. In this process, the algorithm iterates over each task R in $\mathcal{D}_e$ to create a dataset to train our decision tree classifier:

(Let dataset $D = \{(x, y) \mid x \in X, y \in Y\}$ denote the dataset, where X is the set of input feature vectors and Y is the set of output target labels. Each pair (x, y) in D corresponds to a feature vector x from X and its associated label y from Y)

Algorithm 3 LearnGuardConditions($\mathcal{G}$, bestIn)
Training a Decision Tree Classifier at every edge where decision making is involved

```
\\ Creating Decision Sets
Initialize O(v) ← OutVertices(v) for all v ∈ 𝒢
Initialize Q ← F \\ Q is a queue
while Q is not empty do
    vertex u ← Q.dequeue
    for vertex v ∈ InVertices(u) do
        O(v).remove(u)
        for i ∈ Train do
            if v ∈ (bestIn(u, i)) then
                𝒟_e(e).append(i) where edge e = (u → v)
        if O(v) is empty then
            Q.enqueue(v)
\\ Learning Guard
dataset D = {(x, y) | x ∈ X, y ∈ Y} where X ← EnvInputValues(R), Y ← edge e
Guard ← TrainDecisionTree(D) where Guard = (f : X → Y)
return Guard
```

- If all tasks R are common across all outgoing edges from u to v, the feature set X for the vertex u is formed using the environmental input values EnvInputValues(R) (here, Cartesian coordinates of the task's initial distribution points), and the target label Y is set as the first outgoing edge for all tasks R for all $\mathcal{D}_e$.
- In cases where tasks R are not common to all edges, and a specific task appears in multiple but not all edges, the algorithm includes in the dataset for this task the environmental input values $X \leftarrow$ EnvInputValues(R) and target label $Y \leftarrow$ edge e for the first edge it appears in while ignoring its appearances in subsequent edges.
- For tasks unique to an edge, the dataset is constructed such that $X \leftarrow$ EnvInputValues(R) and the target label $Y \leftarrow$ edge e, where the task R appears.

Upon completing data collection for each vertex u, the algorithm proceeds with training a decision tree classifier TrainDecisionTree($D(u)$) for each vertex's dataset. The trained model, denoted as Guard(u), establishes a guard for that particular vertex u.

C Experimental Setup: Implementation Level Details

Model Configuration with Augmented Random Search We use Augmented Random Search (ARS) algorithm to train both the kappa vector and the base policy with the following specific hyperparameters:

- **Learning Rate:** The learning rate is conditionally step decayed, starting from 1 and decreasing to 0.1.

- **Number of Directions Sampled:** The model samples 30 directions per iteration. This sampling is part of the exploration strategy of ARS, allowing the model to investigate various policy adjustments.
- **Number of Top Samples used for Policy Update:** We use the top 8 samples for updating the policy. This means that out of all the directions sampled, the 8 with the highest rewards are used to guide the policy update.
- **Number of Steps in Training:** During training, the agent is allowed 15 steps in each iteration. This setting defines the length of each episode or trial used to evaluate the policy during training.
- **Number of Steps in Testing:** In testing, the agent is allowed a longer leash with 60 steps per iteration. This extended step count enables a more comprehensive evaluation of the trained policy.
- **Network Architecture:** The model's neural network consists of one input layer, two hidden layer, and one output layer.
- **Activation Function:** The ReLU (Rectified Linear Unit) activation function is used for the input and the hidden layer and the tanh activation function is used for the output layer.
- **No. of rollouts tested on:** During testing, we conduct 1000 rollouts for each task.

Pre-Processing Tasks. While training κ_e for an edge $e \in \mathcal{G}$, we start by filtering out all tasks $\mathsf{R}_i \in \mathsf{Train}$ for which no feasible policy exists. We do so by creating task-specific training sets, Train_e for edges $e \in \mathcal{G}$; if a policy cannot be learnt for a certain edge (using $\mathsf{LearnBasePolicy}$) for a task R_i, we remove the task from Train_e. Doing this improves the learning for the other tasks for which a feasible policy exists in the edge e and makes sure the infeasible task does not affect the training of the other tasks.

Testing Methodology for Successful Unseen Tasks. To test for successful unseen tasks, we iterate over each inductive task instance R_i that is not included in Train. For each task, we increment the index i and check the success probability of the task. If the success probability is above or equal to the success threshold (in our success threshold is 0.9), we consider the unseen task R_i successful. We continue this process until we encounter five consecutive indices where the success probability is below 0.9. At this point, we terminate our testing for successful unseen tasks.

D Cartesian Plane (Car2D) Benchmarks

D.1 Environment Description

We consider a continuous cartesian plane which consists of a car which is free to move in the plane. The base objective of the training agent is to teach the car to reach the final goal from its initial position which could be any point in the 2D plane. In this environment, both the state space and action space are continuous in nature.

We add complexity to this task by providing an intermediate point for the car to meet before reaching the end goal or providing rectangular obstacles in its path of traversal.

The coordinates of the points are given as (x, y) where x denotes its position along the x-axis and y denotes its position along the y-axis. The dimensions and coordinates of the obstacle are given by (x_1, y_1, x_2, y_2) where x_1, y_1 gives the coordinates of the bottom left point of the obstacle and x_2, y_2 gives the coordinates of the top right point of the obstacle.

We use certain pre-defined predicates to define our tasks in the environment. Predicate `reach` is interpreted as reaching the coordinates of the specified point. Predicate `avoid` is interpreted as avoiding the cartesian space of the defined obstacle.

Assume a rectangular obstacle *obs* with $a = (a_x, a_y)$ as the bottom-left corner and with $b = (b_x, b_y)$ as the top-right corner, then

- `reach` holds true when point s is near the point *goal* w.r.t euclidean norm $\|.\|_2$

$$\texttt{reach}\ (goal) = (\|s - goal\|_2 < \epsilon_1)$$

- `avoid` holds true when point s is outside the rectangular region defined by its bottom-left corner a and its top-right corner b

$$\texttt{avoid}\ (obs) = (s \notin [a_x, b_x] \times [a_y, b_y])$$

D.2 1-Reachability Tasks with Simple Specifications

In our 1-reachability experiments, we start from an initial distribution $\eta(s)$ and aim to reach a goal point g_1. We examine four different variations of 1-reachability tasks in our Car2D environment. These variations include inductive updates to the initial distribution and the goal state, both with and without obstacles which our car agent must avoid (Illustration in Figure 10a, 10b, 11a, 11b, 12a, 12b). These experiments help us assess the performance of our algorithms across a range of simple tasks.

RL Specifications.

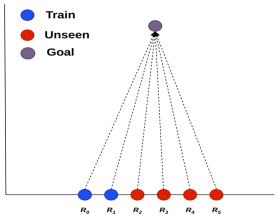
- 1-Reachability Task without obstacle: reach g_1 from $\eta(s)$ with updating initial distribution (Figure 10a),

$$\texttt{achieve}\ (\texttt{reach}\ (g_1)); \ldots; \texttt{achieve}\ (\texttt{reach}\ (g_n))$$

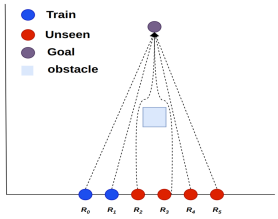
- 1-Reachability Task with obstacle: reach g_1 with updating initial distribution (Figure 10b),

$$\texttt{achieve}\ (\texttt{reach}\ (g_1)); \ldots; \texttt{achieve}\ (\texttt{reach}\ (g_n))\ \texttt{ensuring}\ (\texttt{avoid}\ (obs))$$

The initial distribution is inductively updated by increasing the x-coordinate in successive instances of an inductive task, i.e. $\mathsf{update\_init}(\eta(s)) = \eta(s + (c_1, 0))$ where $c_1 = 0.5$ units.

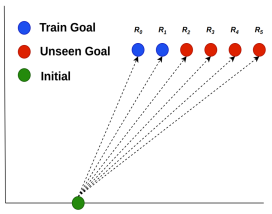


(a)

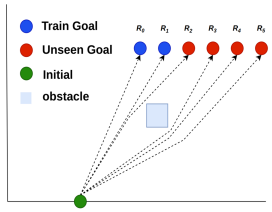


(b)

Fig. 10: 1-Reachability Inductive Task with moving initial distribution: a) without obstacle b) with obstacle

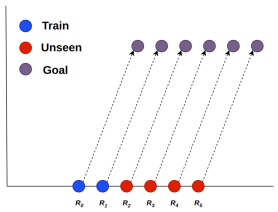


(a)

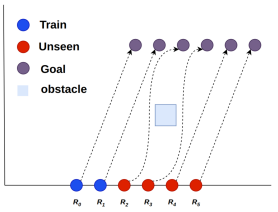


(b)

Fig. 11: 1-Reachability Inductive Task with moving goal point: a) without obstacle b) with obstacle



(a)



(b)

Fig. 12: 1-Reachability Inductive task with moving initial distribution and goal point: a) without obstacle b) with obstacle

– 1-Reachability Task without obstacle: reach g_1 from $\eta(s)$ with updating goal point (Figure 11a),

$$\texttt{achieve}\ (\texttt{reach}\ (g_1)); \ldots ; \texttt{achieve}\ (\texttt{reach}\ (g_n))$$

– 1-Reachability Task with obstacle: reach g_1 with updating goal point (Figure 11b),

$$\texttt{achieve}\ (\texttt{reach}\ (g_1)); \ldots ; \texttt{achieve}\ (\texttt{reach}\ (g_n))\ \texttt{ensuring}\ (\texttt{avoid}\ (obs))$$

The goal is also inductively updated by increasing the x-coordinate in successive instances of an inductive task, i.e. $\texttt{update\_pred}(\texttt{reach}\ (g_1)) = \texttt{reach}\ (g_1 + (c_2, 0))$ where $c_2 = 0.5$ units.

– 1-Reachability Task without obstacle: reach g_1 from $\eta(s)$ with updating initial distribution and goal point (Figure 12a),

$$\texttt{achieve}\ (\texttt{reach}\ (g_1)); \ldots ; \texttt{achieve}\ (\texttt{reach}\ (g_n))$$

– 1-Reachability Task with obstacle: reach g_1 with updating initial distribution and goal point (Figure 12b),

$$\texttt{achieve}\ (\texttt{reach}\ (g_1)); \ldots ; \texttt{achieve}\ (\texttt{reach}\ (g_n))\ \texttt{ensuring}\ (\texttt{avoid}\ (obs))$$

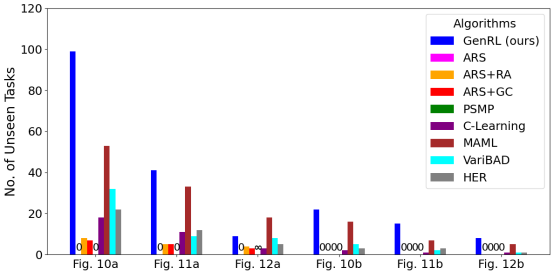


Fig. 14: No. of successful Unseen Tasks for Simpler 1-Reachability experiments.

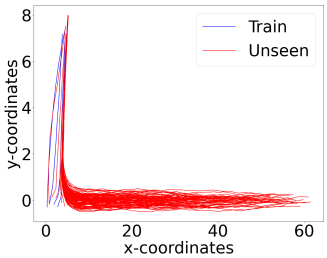


Fig. 15: 1-Reachability Task (updating initial distribution)

Benchmark	**Iterations**					**% gen.**
$\lvert\mathsf{Train}\rvert = 6$	200	400	600	800	1000	**(Best Iter)**
1-Reachability Task without Obstacle						
Figure 10a	48	**99**	**99**	**99**	**99**	1650
Figure 11a	28	33	**41**	**41**	**41**	683
Figure 12a	5	8	8	**9**	**9**	150
1-Reachability Task with Obstacle						
Figure 10b	18	**22**	**22**	**22**	**22**	367
Figure 11b	**15**	**15**	**15**	**15**	**15**	250
Figure 12b	**8**	**8**	**8**	**8**	**8**	133.33

Table 2: No. of successful Unseen task instances for Simpler 1-Reachability experiments across multiple iterations. The number(s) in **boldface** under "Iterations" are the largest (i.e. best) generalization counts for that specification.

The initial distribution and the goal are inductively updated by increasing their x-coordinates by $c_1 = 0.5$ units and $c_2 = 0.5$ units respectively in successive instances of an inductive task, i.e., $\mathsf{update\_init}(\eta(s)) = \eta(s + (c_1, 0))$ and $\mathsf{update\_pred}(\mathtt{reach}\ (g_1)) = \mathtt{reach}\ (g_1 + (c_2, 0))$.

Observations. From Table 2, it is evident that our proposed GenRL method demonstrates significantly superior generalizability across various iterations. We can also see that the GenRL method consistently exhibits higher performance metrics than the baseline in every evaluated scenario from Figure 14. Figure 15 shows the trajectory of the car agent on a 1-reachability task with updating initial distribution which shows our model's capability to generate policies that successfully satisfy unseen tasks for simpler environments and specifications.

Figures 6b and 6d generally perform poorly, with the exception of PSMP in Figure 6b. Notably, PSMP demonstrates absolute success in Figure 6b but fails when obstacles are introduced or when trajectories are no longer simple

Benchmark	**Iterations**					**% gen.**
$\|\mathsf{Train}\| = 6$	**200**	**400**	**600**	**800**	**1000**	**(Best Iter)**
NReach(1)	11	11	**12**	**12**	**12**	200
NReach(2)	8	8	**9**	**9**	**9**	150
NReach(3)	**8**	**8**	**8**	**8**	**8**	133
NReach(4)	**5**	**5**	**5**	**5**	**5**	83
NReach(5)	**5**	**5**	**5**	**5**	**5**	83

Table 3: No. of successful Unseen task instances for N-Reachability experiments without obstacles across multiple iterations. The number in **boldface** under Iterations represents the best generalization for the benchmark.

proportional functions, as seen in Figure 6d and 8a. PSMP, being a planning algorithm rather than a learning algorithm, employs a proportional controller that maps states to actions. This makes PSMP effective in tasks with constant and proportional trajectories but unsuitable for most real-world tasks that involve significant variance in trajectories, as demonstrated by the results.

D.3 N-Reachability Tasks without obstacles

In N-Reachability Tasks, we start from an initial distribution $\eta(s)$ and aim to reach the goal point g_n while navigating via $g_1, \ldots, g_{n_1}$ intermediate goal points. This experiment is designed to test our model's generalization capabilities on long-horizon tasks. Here, the initial distribution $\eta(s)$ and all the goal points $g_1, \ldots, g_n$ inductively update.

RL Specifications. N-Reachability Task without obstacle (NReach(n): reach a set of n goals states $g_1, g_2, \ldots, g_n$ in sequence (Figure 6a),

$$\texttt{achieve}\ (\texttt{reach}\ (g_1)); \ldots; \texttt{achieve}\ (\texttt{reach}\ (g_n))$$

The initial distribution is inductively updated by increasing the x-coordinate in successive instances of an inductive task, i.e. $\mathsf{update\_init}(\eta(s)) = \eta(s + (c_1, 0))$ where $c_1 = 0.5$ units. The goal is also inductively updated by increasing the x-coordinate in successive instances of an inductive task, i.e. $\mathsf{update\_pred}(\texttt{reach}\ (g_1, \ldots, g_n)) = \texttt{reach}\ (g_1, \ldots, g_n + (c_2, 0))$ where $c_2 = 0.5$ units.

Observations. Table 3 clearly demonstrates that GenRL exhibits high generalizability across multiple iterations in N-reachability tasks. Figure 6b highlights that GenRL consistently outperforms the baselines in terms of satisfying unseen tasks across all N-reachability benchmarks. Figure 16 illustrates the trajectory of the car agent in a NReach(3) task where we can see how unseen tasks (marked in red) traverses through the intermediate goal points to reach g_3.

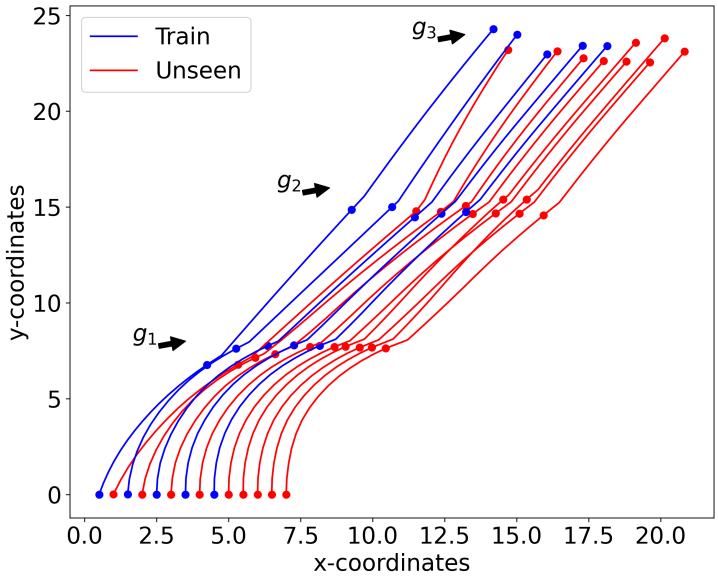


Fig. 16: 3-Reachability Task without obstacle (updating initial distribution and goal) - Trajectory

D.4 N-Reachability Tasks with Obstacles

In N-Reachability Tasks, we start from an initial distribution $\eta(s)$ and aim to reach the goal point g_n while navigating via $g_1, \ldots, g_{n_1}$ intermediate goal points while avoiding the obstacles *obs*. This experiment is designed to test our model's generalization capabilities on long-horizon tasks. Here, the initial distribution $\eta(s)$ and all the goal points $g_1, \ldots, g_n$ inductively update.

RL Specifications. N-Reachability Task with obstacle (NReachObs(n)): reach a set of n goals $g_1, g_2, \ldots, g_n$ in sequence *while avoiding the obstacles* in *obs* (Figure 6c),

$$\texttt{achieve}\ (\texttt{reach}\ (g_1)); \ldots; \texttt{achieve}\ (\texttt{reach}\ (g_n))$$
$$\texttt{ensuring}\ (\texttt{avoid}\ (obs))$$

The initial distribution is inductively updated by increasing the x-coordinate in successive instances of an inductive task, i.e. $\texttt{update\_init}(\eta(s)) = \eta(s + (c_1, 0))$ where $c_1 = 0.5$ units. The goal is also inductively updated by increasing the x-coordinate in successive instances of an inductive task, i.e. $\texttt{update\_pred}(\texttt{reach}\ (g_1, \ldots, g_n)) = \texttt{reach}\ (g_1, \ldots, g_n + (c_2, 0))$ where $c_2 = 0.5$ units.

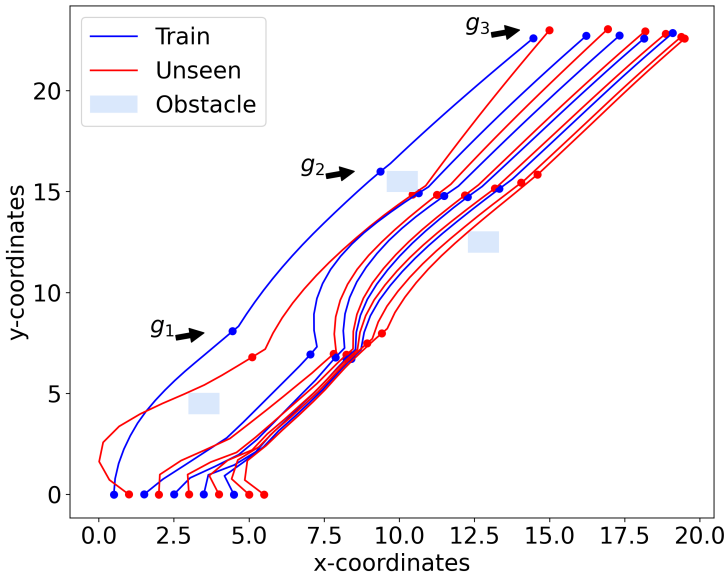


Fig. 17: 3-Reachability Task with obstacle (updating initial distribution and goal) - Trajectory

Observations. Table 4 clearly shows that GenRL has very high generalizability across multiple iterations in tasks with obstacles. Figure 6d shows that GenRL consistently achieves better generalizability than the baselines in all scenarios. However, it can be noted that the generalizability of GenRL shows a marginal decline when compared to tasks without obstacles. This observation can be attributed to the increased complexity and difficulty presented by the obstacle specifications. Figure 17 shows the trajectory of the car agent on a NReachObs(3) task.

D.5 Choice Tasks

In choice tasks, we start from an initial distribution $\eta(s)$ and choose between navigating to intermediate goal g_1 or g_2 based on reachability and then reach *goal* while avoiding obstacle *obs*. This experiment allows us to test the optimality of our guards (branching predicate to choose between the goals) and test long-horizon reachability on complex decision-involving specifications (Illustration in Figure 2a, 9a, 9c).

RL Specifications. Choice Task - Choice(l) (Figure 2a, 9a, 9c): A stack of l sub-tasks (or *levels*), where each sub-task i requires reaching a goal $goal_i$ while

Benchmark	**Iterations**					**% gen.**
$\|\mathsf{Train}\| = 6$	**200**	**400**	**600**	**800**	**1000**	**(Best Iter)**
NReachObs(1)	**8**	**8**	**8**	**8**	**8**	133.33
NReachObs(2)	**7**	**7**	**7**	**7**	**7**	116.33
NReachObs(3)	**5**	**5**	**5**	**5**	**5**	83.33
NReachObs(4)	4	**5**	**5**	**5**	**5**	83.33
NReachObs(5)	**4**	**4**	**4**	**4**	**4**	83.33

Table 4: No. of successful Unseen task instances for N-Reachability experiments with obstacles across multiple iterations. The number in **boldface** under Iterations represents the best generalization for the benchmark.

Benchmark	**Iterations**					**% gen.**	**Guard**
$\|\mathsf{Train}\| = 6$	**200**	**400**	**600**	**800**	**1000**	**(Best Iter)**	**predicate**
Figure 2	6	6	**7**	**7**	**7**	117	$(i \leq 5)$
Figure 9a	FAIL	**5**	**5**	**5**	**5**	84	$(i \leq 5)$
Figure 9c	FAIL	4	**5**	**5**	**5**	84	$(i \leq 5), (i \leq 5)$

Table 5: No. of successful Unseen task instances for Choice experiments. The number in **boldface** under Iterations represents the best generalization for the benchmark.

avoiding the obstacle obs_i, either through the (sub)goal g_{i1} or g_{i2},

$$\begin{aligned}&(\texttt{achieve}\ (\texttt{reach}\ (g_{i1})\ \texttt{or}\ \texttt{reach}\ (g_{i2}));\\ &\quad \texttt{achieve}\ (\texttt{reach}\ (goal_i)))^{l}\\ &\texttt{ensuring}\ (\texttt{avoid}\ (obs))\end{aligned}$$

where, $1 \leq i \leq l$.

We use the superscript l to indicate that the enclosed specification is repeated l times. The inductive tasks on the above tasks inductively modify the initial distribution $\eta(s)$, and the goal positions $goal_i$. The initial distribution is inductively updated by increasing the x-coordinate in successive instances of an inductive task, i.e. $\mathsf{update\_init}(\eta(s)) = \eta(s + (c_1, 0))$ where $c_1 = 1$ units. The goal is also inductively updated by increasing the x-coordinate in successive instances of an inductive task, i.e. $\mathsf{update\_pred}(\texttt{reach}\ (goal_i)) = \texttt{reach}\ (goal_i + (c_2, 0))$ where $c_2 = 1$ units.

Observations. From Table 5, it is evident that our proposed GenRL method demonstrates significantly superior generalizability across various iterations. Figures 2b, 9b, and 9d illustrate how our learned guard optimally indicates which edge to traverse based on the task index, enabling collision-free traversal.

E Tower-Destacking Benchmarks

Environment Description. The 2-link planar arm environment is similar to the Reacher-v2 environment with a modified state and action space. The dynamics of the arm can be described through the joint angles θ_1 and θ_2, which dictate the Cartesian coordinates x and y of the end effector. The relationship between these variables is captured by the forward kinematics equations:

$$x = l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2)$$

$$y = l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2)$$

where l_1 and l_2 represent the lengths of the first and second links, respectively. The state space includes the x and y cartesian coordinates, while the action space consists of θ_1 and θ_2.

The predicate `reach` holds true when point s is inside the rectangular region (block) defined by its bottom-left corner a and its top-right corner b

$$\texttt{reach}\ (s) = (s \in [a_x, b_x] \times [a_y, b_y])$$

RL Specifications. We create five distinct benchmarks to evaluate our model's ability to generalize to long-horizon, complex tasks in this environment.

- **Pick and Drop - Same Side (Figure 7 a)**
  The task involves performing a pick-and-drop operation. Starting from a designated target dropbox, the arm grabs a box from the source tower and moves it to the dropbox where both the target dropbox and the source tower are on the same side of the arm.
  Length of arm: $l_1 = 10$ units and $l_2 = 10$ units.

```
achieve (reach(top_block_source_tower));
achieve (reach(target_box)
```

- **Pick and Vertical Stack - Same Side (Figure 1)**
  The task involves performing a pick-and-drop operation. Starting from a designated source stack of boxes, the arm grabs a box from the source stack and stacks it vertically onto a target stack where both the source and the target stacks are on the same side of the arm.
  Length of arm: $l_1 = 10$ units and $l_2 = 10$ units.

```
achieve (reach(top_block_target_tower));
achieve (reach(top_block_source_tower)
```

– **Pick and Drop - Opposite Side (Figure 7 b)**
The task involves performing a pick-and-drop operation. Starting from a designated target dropbox, the arm grabs a box from the source tower and moves it to the dropbox where both the target dropbox and the source tower are on the opposite side of the arm.
Length of arm: $l_1 = 5$ units and $l_2 = 5$ units.

```
achieve (reach(top_block_source_tower));
achieve (reach(target_box)
```

– **Pick and Vertical Stack - Opposite Side (Figure 7 c)**
The task involves performing a pick-and-drop operation. Starting from a designated source stack of boxes, the arm grabs a box from the source stack and stacks it vertically onto a target stack where both the source and the target stacks are on the opposite side of the arm.
Length of arm: $l_1 = 10$ units and $l_2 = 10$ units.

```
achieve (reach(top_block_target_tower));
achieve (reach(top_block_source_tower)
```

– **Pick and Horizontal Stack - Same Side (Figure 7 d)**
The task involves performing a pick-and-drop operation. Starting from a designated source stack of boxes, the arm grabs a box from the vertical stack and stacks it horizontally along the x-axis onto a target stack where both the source and the target stacks are on the same side of the arm.
Length of arm: $l_1 = 10$ units and $l_2 = 10$ units.

```
achieve (reach(leftmost_block_target_tower));
achieve (reach(top_block_source_tower)
```

F OpenAI Gym Classic Control Benchmarks

F.1 Cartpole

Environment Description. The objective in the Cartpole environment is to balance a pole on a moving cart by applying force to the cart's base. The action space is discrete, with two possible actions: moving the cart left or right. The observation space includes the cart position, cart velocity, pole angle, and pole angular velocity.

Predicate `holdpole` can be defined as:

$$\texttt{holdpole}(\mathit{goal}) = (|\theta - \text{goal}| < \epsilon) \text{ for time } t$$

where θ is the angle of the pole.

RL Specifications. The goal is to balance a pole on a moving cart by applying force to the cart's base i.e. reach a certain theta *goal* and hold it for time t.

$$\texttt{achieve}\ (\texttt{holdpole}(\mathit{goal}))$$

The induction is on the length l of the cartpole which involves increasing the length by $l + 0.4$ units. The training range for l is from 0.4 to 2 units.

F.2 Pendulum

Environment Description. The goal in the Pendulum environment is to keep a free pendulum standing up by applying torque at the pivot point. The action space consists of a single continuous value representing the torque applied to the pendulum's free end. The observation space provides the x-y coordinates of the pendulum's free end and its angular velocity, represented by cosine and sine of the angle and the angular velocity.

Predicate `reachtheta` can be defined as:

$$\texttt{reachtheta}(\mathit{goal}) = (|\theta - \text{goal}| < \epsilon)$$

where θ is the angle of the pendulum.

RL Specifications. The goal is to keep a free pendulum standing up by applying torque at the pivot point i.e. reach a certain theta *goal*.

$$\texttt{achieve}\ (\texttt{reachtheta}(\mathit{goal}))$$

The induction is on the mass m_p of the pendulum which involves increasing the mass by $m_p + 0.1$ units. The training range for m_p is from 1 to 1.4 units.

F.3 Acrobot

Environment Description. The Acrobot's challenge is to swing up the end of a two-link robot arm above a certain threshold. The action space is discrete and deterministic, representing the torque applied at the joint between the two links.

Predicate `reachtip` can be defined as:

$$\texttt{reachtip}(\mathit{goal}) = (|-\cos(\theta_1) - \cos(\theta_2 + \theta_1)| > \epsilon)$$

where θ_1 is the angle of the first joint, and θ_2 is the angle of the second joint w.r.t the first joint.

RL Specifications. The goal is to swing up the end of a two-link robot arm above a certain threshold i.e the tip must reach a certain point *goal*

$$\texttt{achieve}\ (\texttt{reachtip}(\mathit{goal}))$$

The induction is on the mass m_a of the acrobot which involves increasing the mass by $m_a + 0.1$ units. The training range for m_a is from 0.2 to 0.6 units.

Benchmark	**Iterations**					**% Gen.**
$\lvert\mathsf{Train}\rvert = 6$	**100**	**200**	**300**	**400**	**500**	**(Best Iter)**
Cartpole	9	11	11	**14**	12	233
Pendulum	9	8	10	9	**20**	333
Acrobot	0	6	7	9	**10**	167

Table 6: No. of successful Unseen task instances for OpenAI Gym Classic Control experiments. The number in **boldface** under Iterations represents the best generalization for the benchmark.

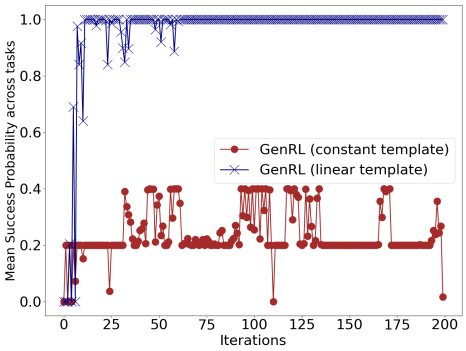


Fig. 18: Template Comparison: Mean Succ. Prob. on Train in 1-Reachability Inductive Task with moving initial distribution

F.4 Observations

Table 6 clearly shows that GenRL satisfies a great number unseen tasks across multiple iterations in the control benchmarks. This shows GenRLs capability to even inductively generate policies where the update is on environment parameters rather than on specification parameters.

G Template Complexity Analysis

The complexity of the policy generator template offers a tradeoff between generalizability and the difficulty of learning (Figure 18): the x-axis represents the number of training iterations and the y-axis represents the success probabilities across all tasks, $\mathsf{R}_i \in \mathsf{Train}$ (each task shown in a different color). For successful learning, the success probability should increase with more training iterations and saturate at a high probability. We consider two templates: (a). **constant update template** $[\pi_{i+1}] = [\pi_i] + \kappa_0$ that only requires learning κ_0 and (b) **linear template** $[\pi_{i+1}] = \kappa_1 \cdot [\pi_i] + \kappa_0$, with two coefficient vectors κ_0 and κ_1 to be learned.

Figure 18 shows the mean success probability over the tasks $\mathsf{R}_i \in \mathsf{Train}$: *constant update template* oscillates around a low success probability of around 0.2,

demonstrating that this template is unable to learn a reasonable policy generator. On the other hand, *linear template* seems to be effective, converging to a success high probability close to 1.0. At the same time, the linear templates has double the number of trainable parameters as compared to constant update template, making it a more involved training exercise. This study outlines the importance of selecting appropriate templates to learn the policy generator successfully.