DO PROGRAMMATIC POLICIES REALLY GENERALIZE BETTER? A RE-EVALUATION STUDY

Anonymous authors
Paper under double-blind review

ABSTRACT

Programmatic policies are often reported to generalize better than neural policies in reinforcement learning (RL) benchmarks. We revisit some of these claims and show that much of the observed gap arises from uncontrolled experimental factors rather than intrinsic representational reasons. Re-evaluating three core benchmarks used in influential papers—TORCS, Karel, and Parking—we find that neural policies, when trained with a few modifications, such as sparse observations and cautious reward functions, can match or exceed the out-of-distribution (OOD) generalization of programmatic policies. We argue that a representation enables OOD generalization if (i) the policy space it induces includes a generalizing policy and (ii) the search algorithm can find it. The neural and programmatic policies in prior work are comparable in OOD generalization because the domain-specific languages used induce policy spaces similar to those of neural networks, and our modifications help the gradient search find generalizing solutions. However, resolving these confounds does not address the deeper question of when programmatic representations provide an inherent OOD generalization advantage over neural ones. We provide an answer to this question by focusing on problems whose solutions require working memory that grows with input size. Commonly used neural architectures cannot encode a solution to this type of problem due to their fixed-capacity design. We show that navigation tasks, such as pathfinding, and domains with nested subproblems fall into this category. As a proof of concept, we construct a modified Karel task for which simple constant-memory heuristics such as wall-following strategies cannot solve, and demonstrate that FUNSEARCH can synthesize an implementation of breadth-first search that provably generalizes OOD. By disentangling representational factors from experimental confounds, we advance our understanding of what makes a representation succeed or fail at OOD generalization. We hope our results will help guide the design of experiments and representations for advancing OOD generalization.

1 INTRODUCTION

Reinforcement learning (RL) has led to remarkable successes in domains ranging from games to robotics, largely by representing policies as highly parametrized neural networks and optimizing them end-to-end (Mnih et al., 2015; Schulman et al., 2017; Lillicrap et al., 2019). However, neural policies often struggle to generalize outside the distribution of their training environments, exhibiting brittle behavior when confronted with out-of-distribution (OOD) scenarios. In contrast, a growing literature on programmatic policies, where decision-making rules are expressed in a domain-specific language, presents empirical evidence of superior OOD generalization compared to neural representations (Verma et al., 2018; 2019; Trivedi et al., 2021; Inala et al., 2020). Despite ample empirical evidence, the literature does not explain why programmatic representations generalize better than neural models. Motivated to find an explanation, we re-evaluated the experiments of these previous works.

Our re-evaluation shows that the reported gap between neural and programmatic representations in generalization stems from uncontrolled experimental factors rather than representational differences. With a few adjustments to the training pipeline, neural policies generalized as well as programmatic ones. For example, in the TORCS experiments of Verma et al. (2018), the neural models failed to generalize to unseen race tracks because they excelled in optimizing for the car's speed on the training track. Programmatic policies are less effective at optimizing speed and thus generalize better to tracks

with sharper turns. Once we replaced the original reward function with a safer one that de-emphasized speed, neural policies matched programmatic ones in generalization. Our re-evaluation still leaves open the question of whether programmatic representations can generalize better than the neural models used in previous work, and, if so, what factors would explain such an advantage.

We posit that a type of representation is successful in OOD generalization if the following two conditions are satisfied: (i) it encodes in its space a solution that generalizes (expressivity), and (ii) the search process used can find and return a solution that generalizes (discoverability). The languages used in the studies we re-evaluate define a space of solutions that is similar, if not identical, to the space of solutions a neural network induces. This means that, if the programmatic space had solutions that generalized, then the neural space also encoded such solutions. The changes that we made to the training pipeline in our re-evaluation allowed the gradient search to find the solutions that generalize.

The comparison of different representations for solving RL problems can be examined considering expressivity and discoverability, either in conjunction or separately. We conjecture that previous work inadvertently evaluated programmatic and neural representations that satisfied expressivity, and discoverability was controlled for the search in the programmatic space, but not in the neural space. As demonstrated by our experiments, controlling for discoverability when both representations evaluated satisfy expressivity can be difficult. This is because, given that expressivity is satisfied, the ability to retrieve solutions that generalize depends on search heuristics that vary with the domain and representation used. Instead, to address which classes of problems permit OOD generalization via programmatic representations but not via commonly used neural ones, we focus on cases where expressivity cannot be satisfied by neural representations but can be satisfied by programmatic ones.

Our investigation focuses on settings where solving the task requires working memory that grows with the input, which is an ability that the commonly used feedforward and recurrent policies lack because their capacity is fixed at training time. Even simple navigation domains contain such cases: general pathfinding requires memory proportional to the size of the problem, and benchmarks with nested subproblems, e.g., NetHack (Küttler et al., 2020), require stack-like context management. In both cases, constant-capacity neural policies cannot represent the algorithmic structures needed to generalize OOD. By contrast, programmatic representations can express such solutions, and we demonstrate this through a proof-of-concept experiment where FUNSEARCH (Romera-Paredes et al., 2024) synthesizes a Python implementation of breadth-first search that provably generalizes OOD.

This paper makes the following contributions. We re-evaluate OOD generalization claims from the literature and show that many of the reported advantages of programmatic representations arose from experimental confounds rather than representational differences. Building on this analysis, we identify classes of problems for which commonly used neural architectures fail to satisfy expressivity. In particular, tasks whose solutions require working memory that grows with the input, such as general pathfinding and domains with nested subproblems. Finally, we provide a proof-of-concept experiment demonstrating that programmatic representations can express solutions with instance-scaling memory that provably generalizes OOD. Together, these results highlight the properties that differentiate representations in their ability to generalize OOD, and we hope they help guide not only the design of future evaluations, but also of novel representations that enable OOD generalization in RL.

2 PROBLEM DEFINITION

We consider sequential decision-making problems modeled as partially observable Markov decision processes (POMDPs) $\mathcal{M} = (S, A, O, p, \Omega, r, \mu, \gamma)$. Here, S is a set of states, A a set of actions, and O a set of observations. The transition $p : S \times A \to \Delta(S)$ and observation functions $\Omega : S \times A \to \Delta(O)$ specify the environment dynamics and the observation process. The agent receives a reward value $R_{t+1} = r(o_t, a_t)$ after taking action a_t at o_t; such a value is given by the reward function $r : O \times A \to \mathbb{R}$. The distribution of initial states is given by μ in $\Delta(S)$, and γ in $[0, 1]$ is the discount factor. A policy $\pi : O \times A \to [0, 1]$ returns the probability of taking action a at observation o.

A class of problems (X, F) defines a set of POMDPs $\{F(x) : x \in X\}$ generated by a parameter space X and a mapping $F : X \to \mathcal{M}$, where each $x \in X$ encodes a problem's input and $F(x)$ is a POMDP defining x. Given a policy class Π, the goal is to find a π that maximizes the return in $F(x)$

$$\arg\max_{\pi \in \Pi} \mathbb{E}_{\pi,p,\mu}[\sum_{k=0}^{\infty} \gamma^k R_{k+1}] \,. \tag{1}$$

The function F is designed such that a solution to $F(x)$ is a solution to the problem that x defines.

Example 1. *To illustrate, consider pathfinding problems over a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of vertices and $\mathcal{E}$ is a set of edges. For an initial and goal vertices v_0 and v_g in $\mathcal{V}$, a solution to the problem is a path $P = \{(v_0, v_1), (v_1, v_2), \cdots, (v_k, v_g)\}$, such that each (v_i, v_j) in P is also in $\mathcal{E}$. In this case, x defines the graph, and the initial and goal vertices. The function F defines the pathfinding problem as a POMDP. In this example, the states S are the set of sequences of edges starting from s_0. The actions for a state where its sequence of edges ends in v_k are all edges (v_k, v_i) in $\mathcal{E}$ that can be added to the sequence. The transition function p is deterministic, since it returns 1.0 if the edge added is in $\mathcal{E}$ and 0.0, otherwise. Since the problem is fully observable, we have that $\Omega = p$. The reward function r returns 0 once the agent finds the last edge connecting s_0 to s_g and -1, otherwise. Finally, μ assigns the value of 1.0 to v_0 and 0.0 to all other vertices, and $\gamma = 1.0$. With this formulation, a policy that solves Equation 1 can also retrieve a path from v_0 to v_g in $\mathcal{G}$.*

Definition 1 (OOD Generalization)**.** *Given a class of problems (X, F), a policy π generalizes out of distribution if a learner searches in a policy space Π for a policy π that solves $F(x)$ for all x in $X_{train} \subset X$, and the resulting π also solves $F(x')$ for any x' in X.*

In practice, $|X_{\text{train}}|$ can be as small as 1, as in the experiments considered in this paper. Moreover, we often cannot prove that the learned π solves all x' in X. Instead, we sample x' from a set $X_{\text{test}} \subset X$ with $X_{\text{test}} \cap X_{\text{train}} = \emptyset$ to evaluate a policy's capability of generalizing OOD.

Example 2. *From Example 1, if the π a learner finds to solve $F(x)$ encodes breadth-first search, such a π is guaranteed to solve any x' in X.*

The class Π determines the biases of the policies we consider. For example, Π could be an architecture of a neural network, and the policies π within this class are the different weights we can assign to the connections of the neural network. We consider classes Π determined by a domain-specific language, so programs written in the language form Π. A language is defined with a context-free grammar $(\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathcal{I})$, where $\mathcal{N}$, $\mathcal{T}$, $\mathcal{R}$, $\mathcal{I}$ are the sets of non-terminals, terminals, the production rules, and the grammar's initial symbol, respectively. Figure 1 (a) shows an example of a context-free grammar encoding a language for TORCS policies. The grammar's initial symbol $\mathcal{I}$ is E. It accepts strings such as the one shown in Figure 1 (b), which is obtained through a sequence of production rules applied to the initial symbol: $E \to \textbf{if } B \textbf{ then } E \textbf{ else } E \to \textbf{if } B \textbf{ and } B \textbf{ then } E \textbf{ else } E \to \cdots$.

We compare policy classes given by neural networks and domain-specific languages, which we refer to as programmatic policies, in terms of OOD generalization. We consider TORCS (Verma et al., 2018; 2019), KAREL (Trivedi et al., 2021), and PARKING (Inala et al., 2020) in our experiments.

3 BACKGROUND: SEARCHING FOR PROGRAMMATIC POLICIES

This section describes the algorithms used to synthesize programmatic policies for solving TORCS (Section 3.1), KAREL (Section 3.2), and PARKING (Section 3.3). We aim to provide enough information so the reader understands our results in Section 4. We do not intend to detail the original algorithms. For full method descriptions, see the cited papers in each subsection.

3.1 NEURALLY DIRECTED PROGRAM SEARCH (NDPS)

Verma et al. (2018) introduced Neurally Directed Program Search (NDPS), a method that uses imitation learning through the DAGGER algorithm (Ross et al., 2011) to learn programmatic policies. Figure 1 (a) shows the domain-specific language Verma et al. (2018) considered in their experiments on the TORCS benchmark. The **peek** function reads the value of a sensor. For example, $\textbf{peek}(h_{\texttt{RPM}}, -1)$ reads the latest value (denoted by the parameter -1) of the rotation-per-minute sensor ($h_{\texttt{RPM}}$); $\textbf{peek}(h_{\texttt{RPM}}, -2)$ would read the second latest value of the sensor. The $\textbf{fold}(+, \epsilon - h_i)$ operation adds the difference $\epsilon - h_i$ for a fixed number of steps of the past readings of sensor h_i.

The non-terminal symbols P, I, and D in Figure 1 (a) form the operations needed to learn PID controllers, with programs that switch between different PID controllers, as shown in Figure 1 (b).

NDPS uses a neural policy as an oracle to guide its synthesis. Given a set of state-action pairs H, where the actions are given by the neural oracle, NDPS evaluates a program ρ by computing the action agreement of ρ with the actions in H. NDPS runs a brute force search algorithm (Albarghouthi

(a) Domain-Specific Language

$$
\begin{array}{rcl}
P & ::= & \textbf{peek}((\epsilon - h_i), -1) \\
I & ::= & \textbf{fold}(+, \epsilon - h_i) \\
D & ::= & \textbf{peek}(h_i, -2) - \textbf{peek}(h_i, -1) \\
C & ::= & c_1 * P + c_2 * I + c_3 * D \\
B & ::= & c_0 + c_1 * \textbf{peek}(h_1, -1) + \ldots \\
 & & \quad \cdots + c_k * \textbf{peek}(h_m, -1) > 0 \mid \\
 & & \quad B \textbf{ or } B \mid B \textbf{ and } B \\
E & ::= & C \mid \textbf{if } B \textbf{ then } E \textbf{ else } E.
\end{array}
$$

(b) Example Policy

$$
\begin{array}{l}
\textbf{if } (0.001 - \textbf{peek}(h_{\texttt{TrackPOS}}, -1) > 0) \\
\quad \textbf{and } (0.001 + \textbf{peek}(h_{\texttt{TrackPOS}}, -1) > 0) \\
\quad\quad \textbf{then } 3.97 * \textbf{peek}((0.44 - h_{\texttt{RPM}}), -1) \\
\quad\quad\quad\quad +0.01 * \textbf{fold}(+, (0.44 - h_{\texttt{RPM}})) \\
\quad\quad\quad\quad +48.79 * (\textbf{peek}(h_{\texttt{RPM}}, -2) - \textbf{peek}(h_{\texttt{RPM}}, -1)) \\
\quad\quad \textbf{else } \ 3.97 * \textbf{peek}((0.40 - h_{\texttt{RPM}}), -1) \\
\quad\quad\quad\quad +0.01 * \textbf{fold}(+, (0.40 - h_{\texttt{RPM}})) \\
\quad\quad\quad\quad +48.79 * (\textbf{peek}(h_{\texttt{RPM}}, -2) - \textbf{peek}(h_{\texttt{RPM}}, -1))
\end{array}
$$

Figure 1: (a) Context-free grammar specifying a domain-specific language for TORCS, a racing car domain (Verma et al., 2018). The initial symbol of the language is E, ϵ is a pre-defined constant, and $\{h_i\}_{i=1}^{m}$ is a set of m sensors from which the agent can read. The grammar allows programs that switch between different PID controllers. (b) Example of a policy written in the language.

et al., 2013; Udupa et al., 2013), to generate a set of candidate programs C. Then, it learns the parameters of the programs (c_1, c_2, and c_3 in Figure 1) with Bayesian optimization (Snoek et al., 2012) such that the programs mimic H. Once NDPS determines the parameters of programs C, it selects the candidate c in C that maximizes the agent's return; c is the starting point of a local search that optimizes a mixture of the action agreement function and the agent's return. Later, Verma et al. (2019) introduced Imitation-Projected Programmatic Reinforcement Learning (PROPEL), an algorithm that also synthesizes programmatic policies, but it controls how different the oracle can be from the programmatic learner, to ease the immitation learning process. The programmatic policies of both NDPS and PROPEL are called for every state the agent encounters.

3.2 LEARNING EMBEDDINGS FOR LATENT PROGRAM SYNTHESIS (LEAPS)

Trivedi et al. (2021) introduced Learning Embeddings for Latent Program Synthesis (LEAPS), a system that learns a latent representation of the space of programs a language induces. When given an MDP $\mathcal{M}$, LEAPS searches in the learned latent space for a vector decoded into a program encoding a policy that maximizes the agent's return at $\mathcal{M}$. LEAPS's premise is that searching in the learned latent space is easier than searching in the space of programs, as NDPS and PROPEL do.

Figure 2 (a) shows the context-free grammar specifying the language used to encode policies for KAREL. The language accepts programs with conditionals and loops. It also includes a set of perception functions, such as `frontIsClear`, which verifies whether the cell in front of the agent is clear. Further included are action instructions such as `move` and `turnLeft`. The set of perception functions is important because it defines what the agent can observe. As we show in Section 4.2, having access to less information allows the agent to generalize to OOD problems. Figure 2 (b) shows an example of a KAREL program. Here, the agent will perform two actions, `pickMarker` and `move`, if a marker is present in its current location; otherwise it will not perform any action.

To learn its latent space, LEAPS generates a data set of programs P by sampling a probabilistic version of the context-free grammar defining the domain-specific language. That is, each production of a non-terminal can be selected with a given probability. A program can be sampled from this probabilistic grammar by starting at the initial symbol and randomly applying production rules until we obtain a program with only terminal symbols. This set of programs is used to train a Variational Auto-Encoder (VAE) (Kingma & Welling, 2014), with its usual reconstruction loss. However, in addition to learn spaces that are more friendly to search algorithms, LEAPS uses two additional losses that attempt to capture the semantics of the programs. These two losses incentivize latent vectors that decode into programs with similar agent behavior to be near each other in the latent space. The intuition is that this behavior locality can render optimization landscapes easier to search.

Once the latent space is trained, it is used to solve MDPs. Given an MDP, LEAPS uses the Cross-Entropy Method (CEM) (Mannor et al., 2003) to search for a vector that decodes into a program that maximizes the return. The rollouts of the decoded policies are used to inform the CEM search.

(a) Domain-Specific Language

```
ρ := def run m(s m)
s := while c(b c) w( s w) | if c(b c) i(s i) |
     ifelse c(b c) i(s i) else e(s e) |
     repeat R=n r(s r) | s;s | a
b := h | not (h)
n := 0, 1, ··· , 19
h := frontIsClear | leftIsClear | rightIsClear |
     markersPresent | noMarkersPresent
a := move | turnLeft | turnRight |
     putMarker | pickMarker
```

(b) Example Policy

```
def run m(
  if c(markersPresent c) i(
    pickMarker move
  i)
m)
```

Figure 2: (a) Context-free grammar specifying a domain-specific language for KAREL. The programs written in this language accept conditional statements and loops. There is a set of perception functions (h) and functions that return actions (a). (b) Example of a policy for a KAREL task.

TRACKS	NDPS LAP TIME	DRL ($\beta = 1.0$) LAP TIME	DRL ($\beta = 0.5$) LAP TIME
G-TRACK-1	1:01	54	1:17
G-TRACK-2 (OOD)	1:40	CR 1608M	1:48 (0.76)
E-ROAD (OOD)	1:51	CR 1902M	1:54 (0.69)
AALBORG	2:38	1:49	2:24
ALPINE-2 (OOD)	3:16	CR 1688M	3:13 (1.00)
RUUDSKOGEN (OOD)	3:19	CR 3232M	2:46 (1.00)

Table 1: For DRL ($\beta = 0.5$), we trained 30 models (seeds) for G-TRACK-1 and 15 for AALBORG. Each cell shows the average lap time (mm:ss) over three laps per model, then averaged across models; 13 models learned to complete G-TRACK-1 and four models learned to complete AALBORG. Values in parentheses for DRL ($\beta = 0.5$) show the fraction of seeds that successfully generalized to the test track (out of 13 and 4 for G-TRACK-1 and AALBORG, respectively). For NDPS and DRL ($\beta = 1.0$), we used the data from (Verma et al., 2018), which is over three models. "CR" indicates that all three models crashed, and the number reported is the average distance at which the agent crashed the car.

3.3 PROGRAMMATIC STATE MACHINE POLICIES (PSM)

Inala et al. (2020) introduced Programmatic State Machine Policies, which we refer to as PSM, a system that learns a policy as a finite-state machine. A finite state machine policy for an MDP $\mathcal{M}$ is a tuple $(M, S, A, \delta, m_0, F, \alpha)$ where M is a finite set of modes. The sets S and A are the sets of states and actions from $\mathcal{M}$. The function $\delta : M \times S \to M$ is the transition function, m_0 in M is the initial mode, and $F \subseteq S$ is the set of modes in which the policy terminates. The transition function δ defines the next mode given the current mode and input state s in S. Finally, $\alpha : M \times S \to A$ determines the policy's action when in mode m and the agent observes state s.

In the PARKING environment, Inala et al. (2020) considered a domain-specific language for the transition function δ and constant values for α. The grammar defining the language δ is the following.

$$B \quad ::= \quad \{s[i] \geq v\}_{i=1}^{n} \mid \{s[i] \leq v\}_{i=1}^{n} \mid B \wedge B \mid B \vee B$$

Here, the values v are constants that need to be learned, $s[i]$ is the i-th entry of the state s the agent observes at a given time step, and n is the dimensionality of the observation.

4 EXPERIMENTS

In this section, we revisit the experiments of Verma et al. (2018) and Verma et al. (2019) on TORCS (Section 4.1 and Appendix B), of Trivedi et al. (2021) on KAREL (Section 4.2 and Appendix C), and of Inala et al. (2020) on PARKING (Section 4.3 and Appendix D).

4.1 TORCS

Verma et al. (2018) and Verma et al. (2019) showed that programmatic policies written in the language from Figure 1 generalize better to OOD problems than neural policies in race tracks of the Open

Racing Car Simulator (TORCS) (Wymann et al., 2000). The results of Verma et al. (2018) also showed that neural policies better optimize the agent's return than programmatic policies, as the former complete laps more quickly than the latter on the tracks on which they are trained. We hypothesized that the programmatic policies generalize better not because of their representation, but because the car moves more slowly, thus making it easier to generalize to tracks with sharper turns.

We test our hypothesis by training models with two different reward functions: the original function used in previous experiments ($\beta = 1.0$ in Equation 2), which we refer to as "original", and a function that makes the agent more cautious about speeding ($\beta = 0.5$), which we refer to as "cautious".

$$\beta \times V_x \cos(\theta) - |V_x \sin(\theta)| - V_x |d_l| \ . \tag{2}$$

Here, V_x is the speed of the car along the longitudinal axis of the car, θ is the angle between the direction of the car and the direction of the track axis, and d_l is the car's lateral distance from the center of the track. The first term of the reward measures the velocity along the central line of the track, while the second is the velocity moving away from the central line. Maximizing the first term minus the second allows the agent to move fast without deviating from the central line. The last term also contributes to having the agent follow the center of the track. Once we set $\beta = 0.5$, the agent will learn policies where the car moves more slowly, which allows us to test our hypothesis. Note that Equation 2 defines an intrinsic reward, since the evaluation, after the agent is trained, is performed on other metrics: lap time and whether the agent has crashed or not. Therefore, by changing β from 1.0 to 0.5 we are not changing the problem, but only how the agent learns to complete a given track.

Following Verma et al. (2018), we use the Deep Deterministic Policy Gradient (DDPG) algorithm (Lillicrap et al., 2019) and TORCS's practice mode, which includes 29 sensors as observation space and the actions of accelerating and steering. We considered two tracks for training the agent: G-TRACK-1 and AALBORG. The first is considered easier than the second based on the track's number of turns, length, and width. The models trained on G-TRACK-1 were tested on G-TRACK-2 and E-ROAD, while the models trained on AALBORG were tested on ALPINE-2 and RUUDSKOGEN.

Table 1 presents the results. NDPS can generalize to the test problems in all three seeds evaluated. DRL with $\beta = 1.0$ does not generalize to the test tracks, with the numbers in the table showing the average distance at which the agent crashes the car in all three seeds. For DRL ($\beta = 0.5$) we trained 30 models (seeds) for G-TRACK-1 and 15 for AALBORG. Then, we verified that 13 of the 30 models learned how to complete laps of the G-TRACK-1 track, and 4 of the 15 models learned to complete laps of the AALBORG track; these models were evaluated on the OOD tracks.

The results support our hypothesis that by changing the reward function, we would allow the agent to generalize. On the training tracks, the lap time increases as we reduce β. Most models trained with $\beta = 0.5$ generalize from the G-TRACK-1 to G-TRACK-2 (76% of the models) and E-ROAD (69%) tracks; all models that learned to complete a lap on AALBORG generalized to the other two tracks.

4.2 KAREL

Trivedi et al. (2021) showed that programs LEAPS synthesized in the language shown in Figure 2 (a) generalized better than deep reinforcement learning baselines to problem sizes much larger than those the agent encountered during training. In our experiments, we consider the fully observable version of KAREL, where the agent has access to the entire grid, and the partially observable version, where the agent can only perceive the cells around it, as shown by the non-terminal h in Figure 2 (a).

In the partially observable case, the problem cannot, in principle, be solved with fully connected neural networks. Consider the two states shown in Figure 3. In one, the agent is going downstairs; in the other, it is going upstairs. Yet, the observation is the same for both states. Trivedi et al. (2021) used LSTMs (Hochreiter & Schmidhuber, 1997) to deal with the partial observability problem. Instead of using LSTMs, which tend to be more complex to train than fully connected networks, we add the last action the

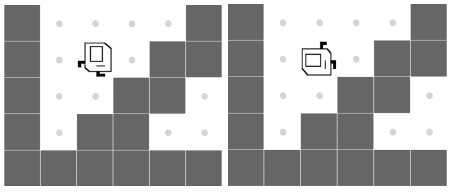

Figure 3: Different states but same observation.

		StairClimber	Maze	TopOff	FourCorner	Harvester
Leaps†	Small	1.00 (0.00)	1.00 (0.00)	0.81 (0.07)	0.45 (0.40)	0.45 (0.28)
	100×100	1.00 (0.00)	1.00 (0.00)	0.21 (0.03)	0.45 (0.37)	0.00 (0.00)
PPO with ConvNet†	Small	1.00 (0.00)	1.00 (0.00)	0.32 (0.07)	0.29 (0.05)	0.90 (0.10)
	100×100	0.00 (0.00)	0.00 (0.00)	0.01 (0.01)	0.00 (0.00)	0.00 (0.00)
PPO with LSTM†	Small	0.13 (0.29)	1.00 (0.00)	0.63 (0.23)	0.36 (0.44)	0.32 (0.18)
	100×100	0.00 (0.00)	0.04 (0.05)	0.15 (0.12)	0.37 (0.44)	0.02 (0.01)
PPO with a_{t-1}	Small	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)	0.59 (0.05)
	100×100	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)	1.00 (0.00)	0.04 (0.00)

Table 2: Generalization results on Karel, where cells show the average return and standard deviation in brackets. "PPO with ConvNet" observes the entire state and employs a convolutional network to learn its representation. "PPO with LSTM" uses an LSTM layer for both actor and critic, while "PPO with a_{t-1}" uses a fully connected network with the observation space augmented with the agent's last action. "Small" refers to the problems in which the models were trained, which were of size either 8×8 or 12×12. Rows marked with a † are from Trivedi et al. (2021). The results for PPO with a_{t-1} are over 30 seeds, and each seed is evaluated on 10 different initial states; the results for Leaps and PPO with a ConvNet and with an LSTM are over five seeds and 10 different initial states.

	Psm		DQN	
	Successful-on-100	Success Rate	Successful-on-100	Success Rate
Training	0.06 (0.09)	0.26 (0.13)	0.40 (0.22)	0.86 (0.14)
Test	0.06 (0.09)	0.16 (0.12)	0.00 (0.10)	0.18 (0.08)

Table 3: Evaluation of 30 seeds of Psm and 15 seeds of DQN on the Parking domain. Each model trained was evaluated on 100 different initial states of both training and testing settings. The columns "Successful-on-100" report the fraction of models trained that successfully solved all 100 initial states. The columns "Success Rate" report the average number of initial states solved across different seeds. We also present the 95% confidence intervals in brackets.

agent has taken as part of the observation. For the fully observable case, we report the results of Trivedi et al. (2021), which used a convolutional network on the input.

We trained policies for the following problems, which were chosen to match the design of Trivedi et al. (2021): StairClimber, Maze, TopOff, FourCorner, and Harvester. The grid size of these problems was either 8×8 or 12×12. After learning to solve these small problems, we evaluated them on grids of size 100×100, also following Trivedi et al. (2021). In the Maze problem, the agent learns to escape a small maze and is evaluated on a larger one. Table 2 shows the results.

Our results show that partial observability combined with a simpler model can generalize to larger grids. Namely, "PPO with a_{t-1}", which uses a fully connected network with the observation augmented with the agent's last action, generalizes to larger problems. This contrasts with "PPO with ConvNet", which operates in the fully observable setting, and "PPO with LSTM", which operates in the partially observable setting but uses a more complex neural model. To illustrate, in Maze, if the agent can only see the cells around itself, it can learn strategies such as "follow the right wall", which is challenging to learn in the fully observable setting. The LSTM agent fails not only to generalize to larger problems, but it often also fails to learn how to solve even the smaller problems.

4.3 Parking

In Parking, the agent must exit a parking spot. During training, the gap between cars is sampled uniformly from $[12.0, 13.5]$, while the test range is $[11.0, 12.0]$, requiring generalization to tighter scenarios. We evaluate both programmatic policies, as described by Inala et al. (2020), and neural policies trained using DQN (Mnih et al., 2015). Preliminary experiments showed that DQN outperformed the PPO and DDPG algorithms considered in our previous experiments. For each policy type, we trained 30 independently seeded models and evaluated each one on 100 test episodes, where the test gap was sampled uniformly from the range $[11.0, 12.0]$.

Table 3 shows the results. We trained 30 independent models of PSM and 15 of DQN. Each model was evaluated on 100 different initial states. The columns "Successful-on-100" refer to the ratio of models that could solve all 100 initial states. For example, 0.06 for PSM means that two of the 30 models solved all initial states on training and test. The "Successful Rate" column shows the ratio of times across all models and initial states that the learned policy could solve the problem. For example, 0.86 for DQN in training means that DQN models solved 86% of the $15 \times 100 = 1500$ initial states.

Our results suggest that the PSM policies generalize better than the DQN policies, as two out of 30 models could solve all 100 test initial states. Looking at the difference between the "Success Rate" of PSM and DQN in training and test also suggests that PSM's policies generalize better, as the gap between the two scenarios is small for PSM: $0.26 - 0.16 = 0.10$ versus $0.86 - 0.18 = 0.68$ for DQN. However, looking at the test "Success Rate" alone suggests that DQN is the winner, as DQN policies can solve more test initial states on average than PSM can. Independent of the metric considered, our results show that PARKING is a challenging domain for both types of representation.

4.4 DISCUSSION

Our experiments showed that neural models can also generalize to OOD problems commonly used in the literature. One key aspect of programmatic solutions is the policy's sparsity. For example, the mode transitions in Figure 5 use a single variable in the Boolean expression. By contrast, neural networks typically use all variables available while defining such transitions, often by encountering spurious correlations between input features and the agent's action. That is why providing fewer input features, combined with a simpler neural model, helped with generalization in KAREL—we remove features that could generate spurious correlations with the model's actions. These results on reducing input features to enhance generalization align with other studies involving the removal of visual distractions that could hamper generalization (Bertoin et al., 2022; Grooten et al., 2024).

In the case of TORCS, OOD generalization was possible due to a safer reward function. If the agent learns on a track that allows it to move fast and never slow down, then it is unlikely to generalize to race tracks with sharp turns that require the agent to slow down. In this case, generalization or lack thereof is not caused by the representation, but by how well the agent can optimize its return while using that representation. We conjecture that NDPS and PROPEL would not generalize to OOD problems if they could find better optimized policies for the agent's return in the training tracks.

PARKING was the most challenging benchmark we considered in our experiments, and we believe it points in the direction of benchmarks that could distinguish the generalization power of programmatic and neural representations. Recurrent neural networks can, in principle, represent the solution shown in Figure 5. In fact, due to the loop of the agent interacting with the environment, the solution to PARKING does not even require loops. If we augment the agent's observation with its last action, a decision tree could encode the repetitive behavior needed to solve the problem. Yet, we could not find either a neural policy or a programmatic one that reliably generalizes to OOD problems in PARKING.

5 WHAT DOES IT TAKE TO GENERALIZE OUT OF DISTRIBUTION?

We define two properties that a given representation needs to satisfy to enable OOD generalization.

Definition 2 (Expressivity)**.** *A policy class Π is expressive for a problem class (X, F) if there exists a policy π in Π that generalizes OOD (Definition 1) for (X, F).*

Definition 3 (Discoverability)**.** *Given a policy class Π that is expressive for a problem class (X, F), we say that Π is discoverable for (X, F) if there exists an algorithm that receives Π and (X, F) and returns a policy π in Π that generalizes OOD for (X, F) within a bounded time limit.*

The domain-specific languages used in our three domains induce spaces similar to those of neural networks. For example, TORCS's language (Figure 1) allows programs with if-then-else chains and trainable parameters $c_0, \ldots, c_k$, a space resembling that of ReLU networks (Orfanos & Lelis, 2023). The ReLU space can be made a superset of the TORCS language by providing the **peek** and **fold** functions, as shown in Figure 1, as network inputs and varying the number of neurons so the network programs can also grow in length—larger networks represent longer programs. In this case, both programmatic and neural spaces are expressive, as they contain solutions that generalize. In our experiments, adjusting the intrinsic reward allowed gradient search to find such solutions, proving

discoverability of the neural space. We also observed a representation that is expressive but not discoverable: although LSTMs can, in practice, approximate finite-state machines (Weiss et al., 2018), which is sufficient for KAREL problems, we could not adjust learning to yield generalizable solutions.

For which type of problem do commonly used neural architectures not satisfy either the expressivity or discoverability property? As suggested by our experiments and those in the literature, controlling for the discoverability property can be challenging because it depends on search heuristics that may not be initially obvious. Thus, we answer this question by controlling for the expressivity property. In KAREL's Maze, shown in Figure 4, the agent needs to find a path from its initial location to the marker (light-gray object), while sensing only the cells adjacent to the agent. Since all corridors of the maze are one-cell wide, wall-following strategies can solve this problem. The neural models considered in our experiments can encode a wall-following algorithm because such algorithms have a constant-memory requirement. Given the direction the agent is moving, which can be given by the agent's last action, and the wall sensors, the next action is computable in $O(1)$ memory per step. Since the width of feedforward models and the hidden state of recurrent models are fixed and independent of the input, they cannot encode algorithms whose working memory grows with the input size.

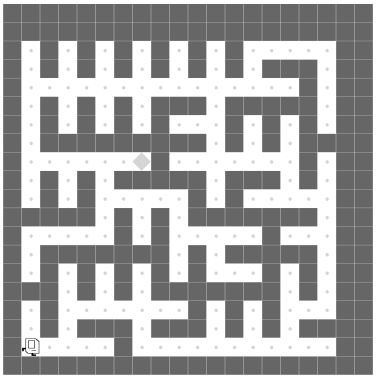

Figure 4: KAREL's Maze.

To illustrate, consider pathfinding over graphs $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ (Example 1), of which KAREL's Maze is a special case. The memory requirement of exact algorithms grows with input size: breadth-first search maintains a frontier and visited set of size $\Theta(|\mathcal{V}|)$; iterative-deepening depth-first search has a memory requirement of $\Theta(d)$, where d is the solution depth, which can grow with the input size. Independent of any specific algorithm, simply indexing a vertex among $|\mathcal{V}|$ candidates requires $\Omega(\log |\mathcal{V}|)$ bits, so a constant-memory representation is insufficient. The neural policies we evaluated have fixed-sized hidden states, which are independent of $|\mathcal{V}|$. Consequently, they are not expressive to represent instance-growing structures and thus cannot generalize OOD in pathfinding problems.

In addition to pathfinding, constant-memory models cannot guarantee OOD generalization in benchmarks that exhibit nested subproblems, such as NetHack (Hambro et al., 2022). In these problems, the agent is interrupted to solve an inner task and must later resume the outer one. Correct handling of arbitrarily deep nesting requires maintaining a stack of pending contexts, used to push a subproblem on interruption and pop it on completion. When the nesting depth can grow with the instance, fixed-capacity models cannot guarantee correctness or OOD generalization, as they cannot represent a stack that grows depending on the input problem. Independent of the inability of these neural models to grow their memory capacity to match the needs of the input, they have been shown to fail to learn stack-like structures (Joulin & Mikolov, 2015). This suggests that these models would fail to solve benchmarks with nested subproblems, even on problems that require only constant memory.

In contrast to the neural models considered, whose memory capacity is constant and determined at training time, programmatic representations can produce policies whose memory capacity grows according to the input size. An algorithm whose working memory usage is a function of the input size can generalize to larger instances, whereas a fixed-capacity model cannot. Although recurrent models are, in theory, computationally universal (Siegelmann & Sontag, 1994; 1995), recent work has shown that they are more limited, theoretically (Nowak et al., 2023) and empirically (Delétang et al., 2023). For example, even when the model has the memory capacity required to solve the problem, increasing the memory needed can lead to imprecise computation, as Weiss et al. (2018) showed in their counting experiment: while LSTMs can learn to count so they can recognize languages such as $a^n b^n$, as the value of n grows during test time, the model starts to present what they called a "slightly-imprecise counting" behavior and fail to generalize to large n. By contrast, a programmatic representation could implement a pushdown automaton that provably generalizes for any value of n.

As a proof-of-concept, we use FUNSEARCH (Romera-Paredes et al., 2024) with Qwen 3-Coder (30B) (Bai et al., 2023) to synthesize a Python program encoding a policy to solve a wall-sparse version of KAREL's Maze (Figure 7), so that wall-following strategies cannot be implemented. The return of a

policy rollout serves as the evaluation function in FUNSEARCH. Three runs of FUNSEARCH returned a correct implementation of breadth-first search, which generalizes to any pathfinding problem.

These findings suggest that, similarly to how software engineers consider the computational requirements of a problem class when implementing a solution, the choice of which representation to use in RL should consider similar factors, so that the learned policy can attain OOD generalization.

Large language models, and memory-augmented models, such as stack-RNNs (Joulin & Mikolov, 2015) and neural Turing machines (Graves et al., 2014), can in principle approximate the structures needed to solve SparseMaze (Yang et al., 2025). However, they do so imperfectly and lack formal correctness and input-scale generalization guarantees. By contrast, programmatic representations can provide such guarantees. Nevertheless, these neural models point to a promising research direction that combines the strengths of neural and programmatic representations, which is worth investigating.

6 RELATION TO OTHER WORKS

Our findings may have implications for other studies comparing the generalization of programmatic and neural policies. For example, Cui et al. (2024) evaluated their method on Karel tasks using a recurrent neural policy baseline; our experiments show that a simpler feedforward network augmented with the agent's last action can improve discoverability in this setting. Guo et al. (2023) represent policies as symbolic equations. Since standard neural networks can approximate these functions arbitrarily well, the two representations are equivalent in expressivity, and any observed generalization gap could be the result of differences in discoverability. Qiu & Zhu (2022) also reported generalization advantages of programs represented by a chain of if-then-else structures over a recurrent model. The programs make calls to predefined functions that encode agent behaviors, such as a move-left function. Feedforward networks could represent the same space of programs if their outputs defined a mixture of the values returned by the functions, as options are often used in RL (Sutton et al., 1999; Bacon et al., 2017). Although a careful investigation is needed, Qiu & Zhu's reported advantage of programmatic representations may also be attributed to confounding factors related to discoverability.

Other works adopt hybrid representations that combine the perceptual strengths of neural networks with the algorithmic structure of programmatic components (Qiu et al., 2023), which have the potential to overcome the memory limitations of standard neural architectures while still benefiting from neural models' ability to handle perception tasks, as in the Houdini system (Valkov et al., 2018).

Although we focused on generalization, programmatic representations offer additional benefits, such as interpretability (Kohler et al., 2024) and improved sample efficiency (Qiu et al., 2023). Some of the benefits related to sample efficiency stem from the modular nature of programs, which enables the reuse of subprograms (Liu et al., 2023). In this case, the subprograms need to generalize to enable reuse. Although the reuse of subprograms is more common with programmatic representations, previous work has also investigated the reuse of subprograms in neural representations through the decomposition of networks (Alikhasi & Lelis, 2024) and policy composition (Qureshi et al., 2020).

7 CONCLUSION

In this paper, we revisited prior claims that programmatic representations generalize better than neural networks in RL tasks. Our re-evaluation showed that much of the reported advantage of programmatic policies arose from experimental confounds rather than fundamental representational differences. When the training pipeline is carefully controlled, neural policies can generalize as well as programmatic ones in the domains considered by previous work. We argued that programmatic policies can offer an advantage by meeting instance-scaling memory requirements needed to generalize OOD. The neural networks considered in this paper are limited by their fixed memory capacity. In contrast, programmatic solutions can find solutions whose memory usage grows with input size. We hope that our results will help guide not only the design of experiments assessing generalization capabilities in RL, but also the design of representations that enable OOD generalization.

REPRODUCIBILITY STATEMENT

The code used to run all experiments in this paper will be made available after the review process.

REFERENCES

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *International Conference Computer Aided Verification*, pp. 934–950, 2013.

Mahdi Alikhasi and Levi Lelis. Unveiling options with neural network decomposition. In *International Conference on Learning Representations*, 2024.

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *AAAI conference on artificial intelligence*, 2017.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report, 2023.

David Bertoin, Adil Zouitine, Mehdi Zouitine, and Emmanuel Rachelson. Look where you look! saliency-guided q-networks for generalization in visual reinforcement learning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Neural Information Processing Systems*, 2022.

Tales Henrique Carvalho, Kenneth Tjhia, and Levi H. S. Lelis. Reclaiming the source of programmatic policies: Programmatic versus latent spaces. In *International Conference on Learning Representations*, 2024.

Guofeng Cui, Yuning Wang, Wenjie Qiu, and He Zhu. Reward-guided synthesis of intelligent agents with control structures. In *The ACM on Programming Languages*, 2024.

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A. Ortega. Neural networks and the chomsky hierarchy. In *Conference on Learning Representations*, 2023.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Bram Grooten, Tristan Tomilin, Gautham Vasan, Matthew E. Taylor, A. Rupam Mahmood, Meng Fang, Mykola Pechenizkiy, and Decebal Constantin Mocanu. Madi: Learning to mask distractions for generalization in visual deep reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems*, pp. 733–742, 2024.

Jiaming Guo, Rui Zhang, Shaohui Peng, Qi Yi, Xing Hu, Ruizhi Chen, Zidong Du, Xishan Zhang, Ling Li, Qi Guo, and Yunji Chen. Efficient symbolic policy learning with differentiable symbolic expression. In *Neural Information Processing Systems*, 2023.

Eric Hambro, Sharada Mohanty, Dmitrii Babaev, Minwoo Byeon, Dipam Chakraborty, Edward Grefenstette, Minqi Jiang, Jo Daejin, Anssi Kanervisto, Jongmin Kim, Sungwoong Kim, Robert Kirk, Vitaly Kurin, Heinrich Küttler, Taehwon Kwon, Donghoon Lee, Vegard Mella, Nantas Nardelli, Ivan Nazarov, Nikita Ovsov, Jack Holder, Roberta Raileanu, Karolis Ramanauskas, Tim Rocktäschel, Danielle Rothermel, Mikayel Samvelyan, Dmitry Sorokin, Maciej Sypetkowski, and Michał Sypetkowski. Insights from the neurips 2021 nethack challenge. In *NeurIPS 2021 Competitions and Demonstrations Track*, volume 176, pp. 41–52, 2022.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8): 1735–1780, 1997.

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

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Neural Information Processing Systems (NeurIPS)*, volume 28, 2015.

Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In *International Conference on Learning Representations*, 2014.

Hector Kohler, Quentin Delfosse, Riad Akrour, Kristian Kersting, and Philippe Preux. Interpretable and editable programmatic tree policies for reinforcement learning, 2024.

Heinrich Küttler, Nantas Nardelli, Alexander H. Miller, Roberta Raileanu, Matteo Selvatici, Edward Grefenstette, and Tim Rocktäschel. The nethack learning environment. In *Conference on Neural Information Processing Systems*, 2020.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

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

Shie Mannor, Reuven Y. Rubinstein, and Yohai Gat. The cross entropy method for fast policy search. In *International Conference on Machine Learning*, pp. 512–519, 2003.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Franz Nowak, Anej Svete, Li Du, and Ryan Cotterell. On the representational capacity of recurrent neural language models. In *Conference on Empirical Methods in Natural Language Processing*, 2023.

Spyros Orfanos and Levi H. S. Lelis. Synthesizing programmatic policies with actor-critic algorithms and relu networks, 2023.

Wenjie Qiu and He Zhu. Programmatic reinforcement learning without oracles. In *Proceedings of the International Conference on Learning Representations*, 2022.

Wenjie Qiu, Wensen Mao, and He Zhu. Instructing goal-conditioned reinforcement learning agents with temporal logic objectives. In *Neural Information Processing Systems*, 2023.

Ahmed H. Qureshi, Jacob J. Johnson, Yuzhe Qin, Taylor Henderson, Byron Boots, and Michael C. Yip. Composing task-agnostic policies with deep reinforcement learning. In *International Conference on Learning Representations*, 2020.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *International Conference on Artificial Intelligence and Statistics*, pp. 627–635, 2011.

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

Hava T. Siegelmann and Eduardo D. Sontag. Analog computation via neural networks. *Theoretical Computer Science*, 131(2):331–360, 1994. doi: 10.1016/0304-3975(94)90172-4.

Hava T. Siegelmann and Eduardo D. Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150, 1995. doi: 10.1006/jcss.1995.1013.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Neural Information Processing Systems*, volume 25, pp. 2951–2959, 2012.

Richard S Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J. Lim. Learning to synthesize programs as interpretable and generalizable policies. In *Neural Information Processing Systems*, pp. 25146–25163, 2021.

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. Transit: Specifying protocols with concolic snippets. In *Conference on Programming Language Design and Implementation*, pp. 287–296, 2013.

Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. Houdini: Lifelong learning as program synthesis. In *Neural Information Processing Systems*, 2018.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, volume 80, pp. 5045–5054, 2018.

Abhinav Verma, Hoang Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected programmatic reinforcement learning. In *Neural Information Processing Systems*, volume 32, 2019.

Gail Weiss, Yoav Goldberg, and Eran Yahav. On the practical computational power of finite precision rnns for language recognition. In *Annual Meeting of the Association for Computational Linguistics*, pp. 740–745, 2018.

Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. Torcs, the open racing car simulator. *Software available at http://torcs. sourceforge. net*, 4(6):2, 2000.

Siwei Yang, Bingchen Zhao, and Cihang Xie. AQA-bench: An interactive benchmark for evaluating LLMs' sequential reasoning ability in algorithmic environments. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856.

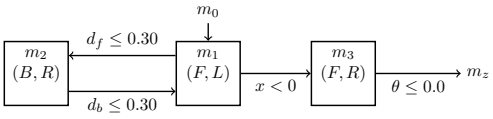

Figure 5: Example of a state machine policy, where m_0 is the initial mode and m_z is an accepting mode. The tuples inside each mode specify the agent's action when in that mode (e.g., (F, L) means "move forward and steer to the left". The transitions from one mode to another are triggered by a Boolean expression shown in the arrows. For example, if the car is too close to the car in front of it ($d_f \leq 0.30$), then the policy moves from m_1 to m_2. The agent remains in the current mode if no outgoing Boolean expression is triggered. This policy is based on an example by Inala et al. (2020).

A EXAMPLE PSM

Figure 5 shows an example of the type of policy PSM learns. In this example, the policy is for PARKING, a domain where the agent must learn how to exit a parking spot with a car in front of the agent's car (car_f) and another at the rear (car_b). The policy uses the following state features: the distance between the agent's car and car_f (d_f) and car_b (d_b), the x coordinate of the car, and the angle θ of the car. A solution involves the agent moving forward to the left (mode m_1) and then back to the right (mode m_2), until the agent has cleared car_f (transitioning to mode m_3). The agent solves the problem if it straightens the car after clearing car_f, thus transitioning from m_3 to m_f. PSM's policies are called only once for the initial state; the policy returns only at the end of the episode.

B TORCS DETAILS

We use the hyperparameters in Table 4 with DDPG (Lillicrap et al., 2019).

Hyperparameter	**Selected Value**
Actor's learning rate	0.0003
Critic's learning rate	0.001
Batch size	64
Buffer size	100000
τ	0.005
L1 regularization	0.00001
Max steps	20000
Training episodes	600

Table 4: Hyperparameter Configuration Used for TORCS

C KAREL DETAILS

We used Proximal Policy Optimization (PPO) (Schulman et al., 2017) with the agent's previous action appended to the observation vector. A comprehensive hyperparameter sweep was conducted over the values from Table 5.

The max steps used for training and testing different Karel tasks are shown in Table 6.

Table 7 shows the best-performing configuration across all five tasks for final evaluation. The selection was based on the agent's average return across the three seeds after two million time steps of training.

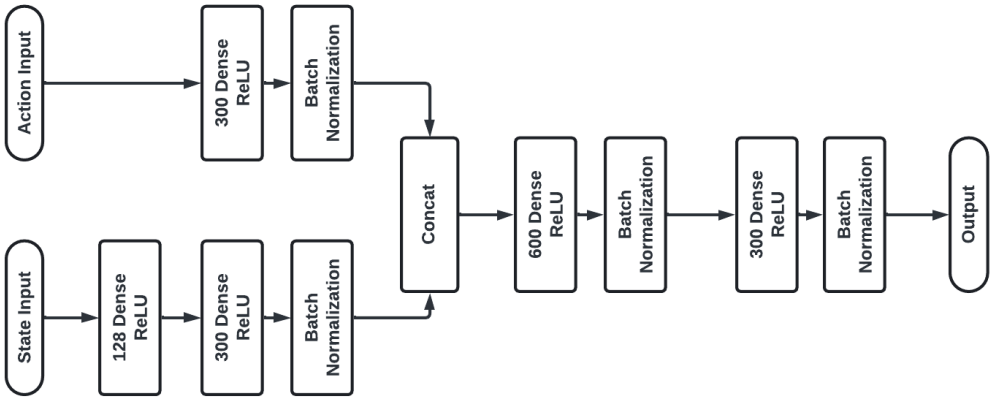

Figure 6: Architecture of the critic network used in DDPG for the TORCS environment.

Hyperparameter	Values Tested
Learning rate	{0.001, 0.0001, 0.00001}
Clipping coefficient	{0.01, 0.1, 0.2}
Entropy coefficient	{0.001, 0.01, 0.1}
L1 regularization	{0.0, 0.0001, 0.0005, 0.001}
Actor's hidden layer size	{32, 64}
Training time steps	{2 million}
Seeds per config	{3}

Table 5: PPO + Last Action: Hyperparameter Sweep Configuration for Karel

Hyperparameter	Selected Value
Learning rate	0.001
Clipping coefficient	0.1
Entropy coefficient	0.1
L1 regularization	0.0
Actor's hidden layer size	32

Table 7: Best Hyperparameter Configuration Used for Final Training of Karel

The best configuration was then trained with 30 random seeds, and evaluation results were averaged over 10 distinct initial configurations per seed. Additionally, four other hyperparameter configurations achieved 100% generalization on four out of five tasks: StairClimber, Maze, and TopOff.

Training used diverse initial state configurations. Whenever feasible, we enumerated all combinations of agent and goal placements. Specifically:

- For StairClimber, TopOff, and FourCorner, all possible agent-goal placements were used.
- For Maze, where full enumeration was computationally infeasible, we sampled 5 random mazes and placed the goal at every position on the grid.

The training grid sizes for each task were:

- 12×12 for StairClimber, TopOff, and FourCorner
- 8×8 for Maze and Harvester

	StairClimber	Maze	TopOff	FourCorner	Harvester
Training	50	100	100	100	200
Test	1000	100000	1000	1000	10000

Table 6: Max steps of episodes for each Karel task during training and test.

D Parking Details

We used a single-hidden-layer DQN architecture with 64 units for the neural baseline. The agent operated over a discretized action space, where continuous actions were mapped onto n equally spaced values using a fixed action resolution. We performed a grid search over the hyperparameter values listed in Table 8. The selected hyper-parameters are shown in Table 9.

Hyperparameter	**Values Tested**
Learning rate	{0.01, 0.001, 0.0001}
Batch size	{64, 128, 256}
Target update frequency	{100, 500, 1000}
ϵ	{0.1, 0.01}
Replay buffer size	{1 million, 2 million}
Action resolution	{3, 5, 7}
Seeds per config	{10}

Table 8: DQN: Hyperparameter Sweep Configuration for Parking Domain

Hyperparameter	**Selected Value**
Learning rate	0.0001
Batch size	64
Target update frequency	1000
ϵ	0.01
Action resolution	3

Table 9: Best Hyperparameter Configuration Used for Final Training

The original Parking benchmark introduced by Inala et al. (2020) was not designed with reinforcement learning in mind—it provides "safety check" to invalidate policies that crash the car or get out of boundaries. We define both a shaped reward function and a termination condition to adapt it for RL. If the agent successfully reaches the parking exit, the episode ends with a large positive reward (2 × max episode length); if it takes an unsafe action, it terminates immediately with a large negative penalty (–2 × max episode length). Otherwise, at each timestep the agent receives $r_t = -\big(2|x_{\text{agent}} - x_{\text{goal}}| + |y_{\text{agent}} - y_{\text{goal}}|\big) - 1$, i.e., the (weighted) negative Manhattan distance minus an extra step penalty of 1, encouraging the car to move closer to the exit.

E Sparser Version of Karel's Maze

Figure 7 shows the sparser version of Karel's Maze we used in our FunSearch experiments.

Listing 1 shows the policy that Funsearch (Romera-Paredes et al., 2024) gave after 21 iterations and Listing 2 is the given prompt. We used Qwen 3-Coder, 30B variant (Bai et al., 2023) as the LLM for this part.

Listing 1: Funsearch Policy

```
def get_action(env: KarelGymEnv) -> Union[list[int], str]:
```

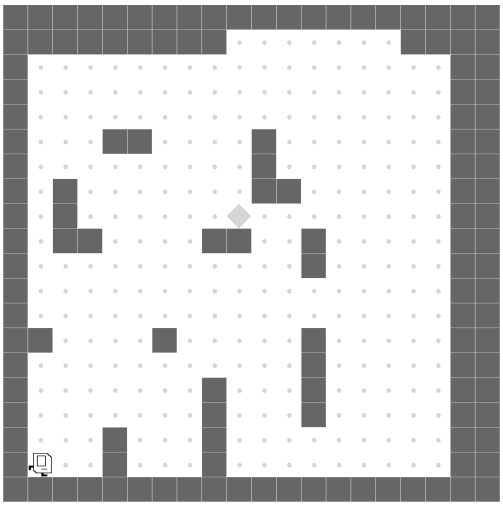

Figure 7: Sparser version of KAREL's Maze, which cannot be solved with wall-following strategies.

```
"""Creates a policy that returns a list of actions for the Karel agent
    to take in the environment."""

def find_path_bfs(start_r, start_c, goal_r, goal_c, walls):
  from collections import deque

  queue = deque([(start_r, start_c, [])])
  visited = set((start_r, start_c))
  directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

  while queue:
    current_r, current_c, path = queue.popleft()

    if (current_r, current_c) == (goal_r, goal_c):
      return path

    for dr, dc in directions:
      new_r, new_c = current_r + dr, current_c + dc
      if 0 <= new_r < env.env_height and 0 <= new_c < env.env_width and
          not walls[new_r, new_c] and (new_r, new_c) not in visited:
        visited.add((new_r, new_c))
        queue.append((new_r, new_c, path + [(dr, dc)]))

  return []

def convert_path_to_actions(start_r, start_c, start_d, path):
  actions = []
  current_r, current_c, d = start_r, start_c, start_d
  directions_map = {(0, 1): 1, (1, 0): 2, (0, -1): 3, (-1, 0): 0}

  for dr, dc in path:
    target_d = directions_map[(dr, dc)]

    while d != target_d:
      if (d + 1) % 4 == target_d: # Turn right
        actions.append(2)
      elif (d - 1) % 4 == target_d: # Turn left
        actions.append(1)
      else: # Turn around
        actions.extend([1, 1])

      d = target_d
```

```
    actions.append(0) # Move forward

  return actions

state_arr = env.task.get_state()
walls = state_arr[4].astype(bool)
r, c, d = env.task.get_hero_pos()
goal_r, goal_c = env.task_specific.marker_position

path = find_path_bfs(r, c, goal_r, goal_c, walls)

if not path:
  return [random.randint(0, 4) for _ in range(50)]

actions = convert_path_to_actions(r, c, d, path)

# Extend actions to ensure the policy has enough steps
while len(actions) < 50:
    actions.append(random.choice([1, 2])) # Randomly turn left or right

return actions
```

Listing 2: Funsearch Prompt

```
"""
Specification for the Karel SparseMaze environment.

We are searching for a function `get_action(env)` that returns a list of
    actions list[int] for the Karel environment.
get_action(env) should return a policy that can solve the maze all the
    time, regardless of the initial configuration. Then by calling this
    policy, it can get the actions for that specific initial
    configuration.

Input is a KarelGymEnv object.
- You can access the height and width of the env like this: env.env_width,
     env.env_height
- You can access walls like this:
  # static walls from feature index 4 of the Karel state
  state_arr = env.task.get_state() # shape: (features, H, W)
  self.walls = state_arr[4].astype(bool) # True where wall
- You can the row, column, and direction of the agenr like this:
  r, c, d = env.task.get_hero_pos()
- And the directions are like this:
  0: 'Karel facing North',
  1: 'Karel facing East',
  2: 'Karel facing South',
  3: 'Karel facing West',
- Access the goal marker position like this:
  goal_r, goal_c = env.task_specific.marker_position
- You can access the observation like this:
  obs = env._get_observation_dsl() # shape: (4,), [frontIsClear,
      leftIsClear, rightIsClear, markersPresent]

The actions are:
  0: move
  1: turnLeft
  2: turnRight
  3: pickMarker (not used in maze)
  4: putMarker (not used in maze)

In the maze task, the agent starts at a fixed position and must find its
    path to a goal marker. The environment uses a sparse reward: 1 when
    reaching the goal, 0 otherwise.
```

```
The environment is a sparse maze (corridors are 2 cells wide) and has
    multiple initial configurations (both mazes and goal positions).

This specification describes the key classes, variables, and functions
    used to define a Gym compatible "Karel SparseMaze" task, where an
    agent navigates a
carved maze to reach a goal marker under sparse rewards.

Package Layout:
  project_root/
  |-- funsearch/
     |-- implementation/
     | |-- utils.py
     | |-- temp1.py # top-level script that calls evaluate and get_action
     |-- karel_wide_maze/
     | |-- __init__.py
     | |-- karel_wide_maze.py
     | |-- karel_wide_maze_prompt_spec.py
     | |-- gym_envs/
     | | |-- __init__.py
     | | |-- karel_gym.py # Defines KarelGymEnv
     | |-- karel_tasks/
     | | |-- __init__.py
     | | |-- maze.py # Defines Maze, MazeSparse, MazeWide, etc.
     | |-- karel/
     | | |-- __init__.py
     | | |-- environment.py # Defines KarelEnvironment and features
     | |-- base/
     | |-- __init__.py
     | |-- task.py # Defines BaseTask
     |...

Usage Summary:
  1. The FunSearch framework "evolves" a Python function `get_action(env)`
       to maximize `evaluate(n)`.
  2. `evaluate(n)` runs n episodes of the Karel SparseMaze environment,
      each seeded differently, with different locations for walls and goal.

  3. Each episode calls `run_episode()`, which repeatedly:
     - Queries `get_action(env)` to obtain actions: {0..4}.
     - Steps the Gym environment and accumulates sparse/dense rewards.
     - Terminates when Karel reaches the goal or max_steps is reached.
  4. Maze-classes in karel_tasks/maze.py carve out a random maze layout (
      via DFS), set a goal marker,
     and compute rewards either sparsely (1 upon reach) or densely (
        normalized distance progress).
  5. KarelGymEnv wraps these Tasks into a standard Gym API: it exposes `
      step()`, `reset()`, `render()`,
     `action_space`, `observation_space`, and handles "multiple initial
        configurations" if requested.
"""
```

F LEARNING CURVES

Figure 8 shows the learning curves of the neural policies for the five KAREL tasks and the PARKING environment. Each plot reports the mean return across seeds: 30 for each Karel task and 15 for the PARKING environment, with 95% confidence intervals. Figure 9 shows the learning curves for the programmatic approach of LEAPS for 32 seeds. Figure 10 shows the learning curves for the neural policies for the TORCS environments, with 95% confidence intervals. As explained in the main text, the agent learned to complete AALBORG in only four out of 30 seeds, and 13 out of 15 seeds learned

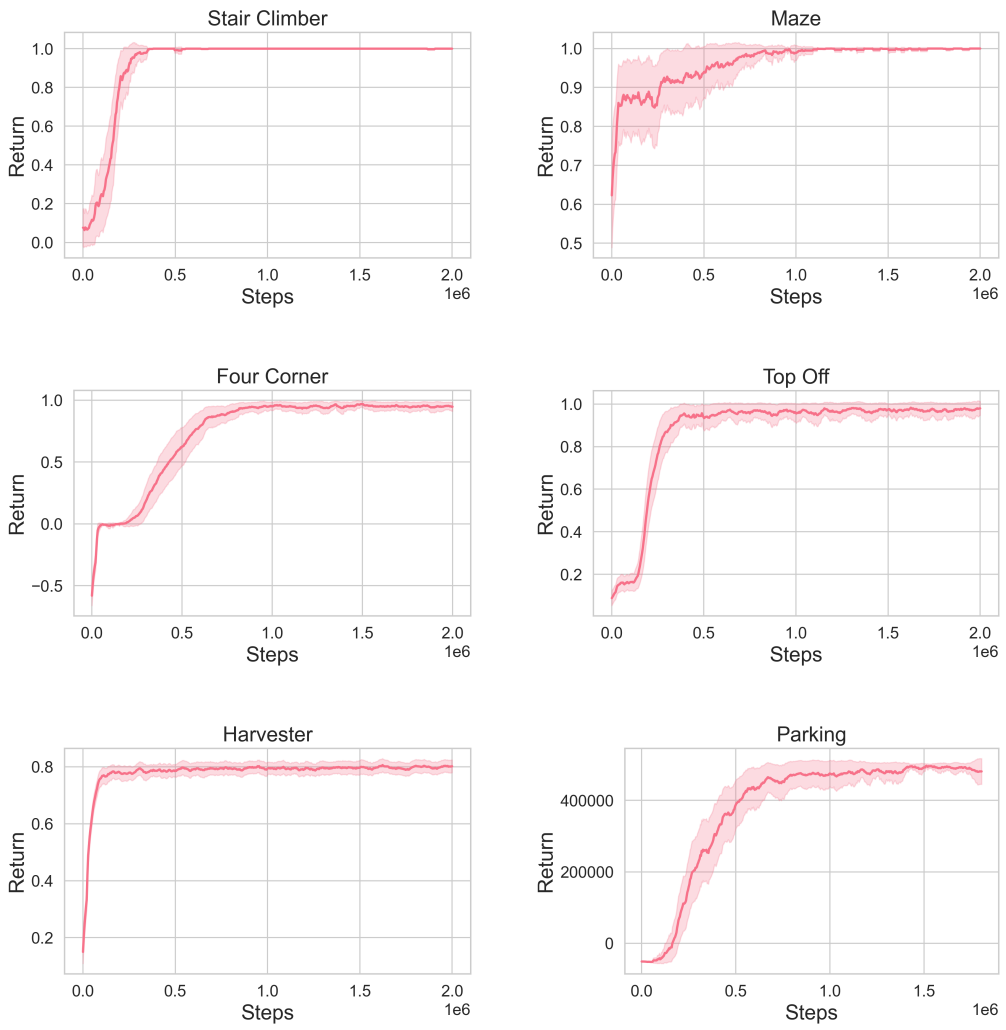

Figure 8: Learning curves of neural policies on Karel and Parking tasks. Reported mean and 95% confidence interval over 30 and 15 seeds for Karel and Parking environments, respectively.

to complete G-TRACK-1; we plot the seeds in which the agent was successful, as these are the ones we evaluated in our OOD experiments. Naturally, a small number of seeds leads to a large uncertainty. Despite the small number of successful seeds, the experiment still fulfills our goal of showing that there exist neural policies that can generalize OOD in TORCS. Following Carvalho et al. (2024), the leaps plots follow a watermark approach: once a solution with a return of x is encountered, we continue plotting x until a better solution is discovered. We follow a similar approach for TORCS. For all domains, we return the solution corresponding to the rightmost value in the plots.

G LARGE LANGUAGE MODEL USAGE

We used LLMs to edit the text lightly, to trim words to shorten paragraphs, and to catch typos and ungrammatical sentences.

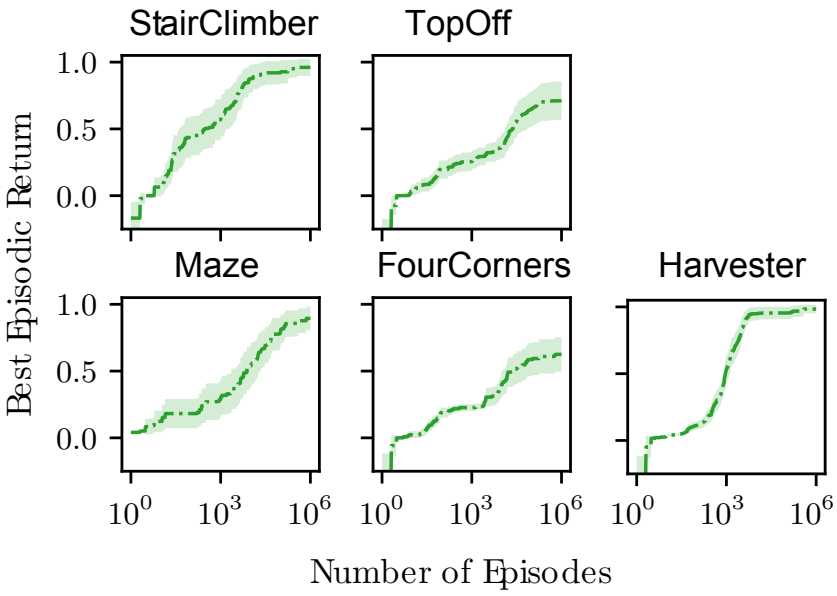

Figure 9: Episodic return performance of LEAPS on Karel tasks with the implementation from Carvalho et al. (2024). Reported mean and 95% confidence interval over 32 seeds. The x-axis is represented on a log scale.

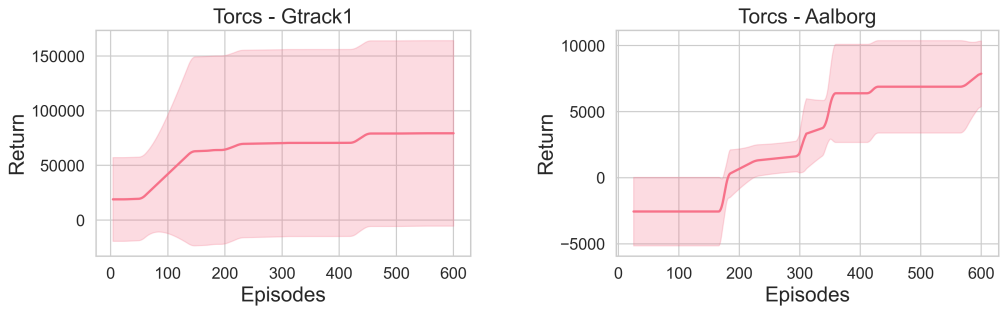

Figure 10: Episodic return performance of neural policies on the TORCS training tracks. Reported mean and standard error over 13 and 4 seeds for G-TRACK-1 and AALBORG tracks, respectively.