Few-Shot Bayesian Imitation Learning with Logic over Programs

Tom Silver, Kelsey R. Allen, Alex K. Lew, Leslie Pack Kaelbling, Josh Tenenbaum
MIT
Cambridge, MA
{tslvr, krallen, alexlew, jbt}@mit.edu, lpk@csail.mit.edu

Abstract

We describe an expressive class of policies that can be efficiently learned from a few demonstrations. Policies are represented as logical combinations of programs drawn from a small domain-specific language (DSL). We define a prior over policies with a probabilistic grammar and derive an approximate Bayesian inference algorithm to learn policies from demonstrations. In experiments, we study five strategy games played on a 2D grid with one shared DSL. After a few demonstrations of each game, the inferred policies generalize to new game instances that differ substantially from the demonstrations. We argue that the proposed method is an apt choice for policy learning tasks that have scarce training data and feature significant, structured variation between task instances.

1 Introduction

In spite of the recent advances in policy learning, catalyzed in large part by the integration of deep learning methods with classical reinforcement learning techniques, we still know relatively little about policy learning in the regime of very little data and very high variation between task instances. We study the problem of learning a policy from a few (at most ten) demonstrations and generalizing to new task instances that differ substantially from those seen during training.

For example, consider the "Reach for the Star" task illustrated in Figure 1. A robot must navigate a grid to reach a star in the presence of gravity. Clicking an empty cell creates a new block and clicking an arrow (bottom right) moves the agent. Accomplishing this task requires building "stairs," block by block, from the floor to the star and moving the robot to climb them. The size of the grid, the positions of the robot and star, and the size of the required staircase vary between instances. Given only the three demonstrations in Figure 1, we wish to learn a policy that generalizes to new (arbitrarily large) instances.

To proceed with policy learning, we must first define a hypothesis class Π of candidate policies. In deep reinforcement learning, Π is typically the set of all neural networks with a particular architecture. This policy class is highly (universally) expressive; one can be confident that a nearly optimal policy exists within the class. Moreover, policies in this class can be learned via gradient-based optimization. The well known challenge of this approach is that policy learning can require an exorbitant amount of training data. As we see in experiments, deep policies are prone to severe overfitting in the low-data, high-variation regime that we consider here, and fail to generalize as a result.

We may alternatively define policies with programs drawn from a domain-specific language (DSL). Program-based representations are invoked in cognitive science to explain how humans can learn concepts from only a few examples [Goodman et al., 2008, 2014, Piantadosi et al., 2016]. A prior over programs can be defined with a probabilistic grammar so that, for example, shorter programs are preferred over longer ones (c.f. Occam's razor). Policies implemented as programs are thus promising

Preprint. Work in progress.

from the perspective of generalization in our low-data regime. However, it is difficult to achieve both expressivity and learnability at the same time. For a small DSL, learning is easy, but many functions cannot be represented; for a large DSL, more functions may be expressible, but learning is often intractable. It is perhaps because of this dilemma that program-based policies have received scarce attention in the literature (but see Wingate et al. [2013], Xu et al. [2018], Lázaro-Gredilla et al. [2019]).

In this work, we describe a new program-based policy class — PLP (Policies represented with Logic over Programs) — that leads to strong generalization from little data without compromising on expressivity or learnability. Our first main idea is to use programs not as full policies but as low-level feature detectors, which are then composed into full policies. The feature detectors output binary values and are combined with Boolean logical operators (and's, or's, and not's). In designing a programming language over feature detectors, we are engaging in a sort of "meta-feature engineering" where high-level inductive biases are automatically compiled into rich low-level representations. For example, if we suspect that our domain exhibits local structure and translational invariance (as in convolutional neural networks), we can include in our programming language a family of feature detectors that examine local patches of the input (with varying receptive field size). Similarly, if we expect that spatial relations such as "somewhere above" or "somewhere diagonally left-down" are useful to consider, we can include a family of feature detectors that "scan" in some direction until some condition holds, thereby achieving an invariance over distance. These invariances are central to our ability to represent a single policy for "Reach for the Star" that works across all grid sizes and staircase heights.

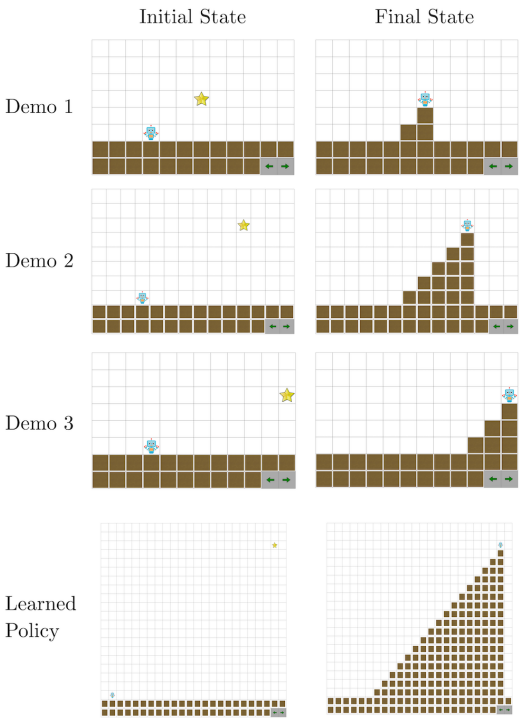

Figure 1: "Reach for the Star" demonstrations and learned policy.

Our second main idea is to exploit the structure of PLP to derive an efficient imitation learning algorithm, overcoming the intractability of general program synthesis. To learn a policy in PLP, we gradually enumerate feature detectors (programs), apply them to the training data, and invoke an off-the-shelf Boolean learning method (decision tree ensembles) to learn combinations of the feature detectors found so far. What would be an intractable search over full policies is effectively reduced to a manageable search over feature detectors. We frame this learning algorithm as approximate Bayesian inference and introduce a prior over policies $\mathsf{p}_\pi(\cdot)$ via a probabilistic grammar over the constituent programs. Given training data $\mathcal{D}$, the output of our learning algorithm is then an approximate posterior over policies $\mathsf{q}(\pi) \approx \mathsf{p}_{\pi|\mathsf{D}}(\pi \mid \mathcal{D}) \propto \mathsf{p}_{\mathsf{D}|\pi}(\mathcal{D} \mid \pi)\mathsf{p}_\pi(\pi)$, which we further convert into a final policy $\pi_*(s) = \arg\max_{a \in \mathcal{A}} \mathbb{E}_{\mathsf{q}}[\pi(a \mid s)] = \arg\max_{a \in \mathcal{A}} \sum_{\pi \in \Pi} \pi(a \mid s)\mathsf{q}(\pi)$.

While PLP and our proposed learning method are agnostic to the particular choice of the DSL and application domain, we focus here on five strategy games which are played on 2D grids of varying sizes (see Figure 2). In these games, a state is a full grid layout and an action is a single grid cell (a "click" somewhere on the grid). The games are diverse in their transition rules and in the tactics required to win, but the common state and action spaces allow us to build our policies for all five games from one shared, small DSL. Our DSL is inspired by ways that humans focus and shift their attention between cells when analyzing a scene. In experiments, we learn policies from a few demonstrations in each game. We find that the learned policies can generalize perfectly in all five games after five or fewer demonstrations. In contrast, policies learned as convolutional neural networks fail to generalize, even when domain-specific locality structure is built into the architecture (as in fully convolutional networks [Long et al., 2015]). We also verify empirically that enumerating full program-based policies is intractable for all games. In ablation studies, we find that the feature

detectors alone do not suffice, nor does a simple sparsity prior; the probabilistic grammar prior is required to learn policies that generalize. Overall, our experiments suggest that PLP offers an efficient, flexible framework for learning rich, generalizable policies from very little data.

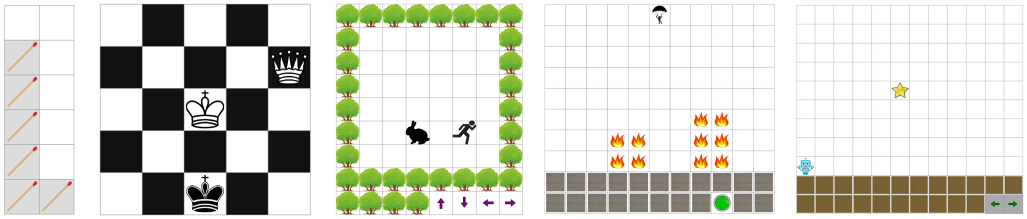

Figure 2: The five strategy games studied in this work. See the appendix for descriptions and additional illustrations.

2 Related Work

Imitation learning is the problem of learning and generalizing from expert demonstrations [Pomerleau, 1991, Schaal, 1997, Abbeel and Ng, 2004]. To cope with limited data, the demonstrations can be used in combination with additional reinforcement learning [Hester et al., 2018, Nair et al.]. Alternatively, a mapping from demonstrations to policies can be learned from a background set of tasks [Duan et al., 2017, Finn et al., 2017]. A third option is to introduce a prior over a structured class of policies [Andre and Russell, 2002, Doshi-Velez et al., 2010, Wingate et al., 2011], e.g., hierarchical or compositional policies [Niekum, 2013, Ranchod et al., 2015, Daniel et al., 2016, Krishnan et al., 2017]. Our work here fits into the third tradition: our main contribution is a new policy class with a structured prior that enables efficient imitation learning.

We define policies in terms of programs [Wingate et al., 2013, Xu et al., 2018] and a prior over policies using a probabilistic grammar [Goodman et al., 2008, 2014, Piantadosi et al., 2016]. Similar priors appear in Bayesian concept learning from cognitive science [Tenenbaum, 1999, Tenenbaum and Griffiths, 2001, Lake et al., 2015, Ellis et al., 2018a,b]. Incorporating insights from program induction for planning has seen renewed interest in the past year. Of particular note is the work by Lázaro-Gredilla et al. [2019], who learn object manipulation concepts from before/after image pairs that can be transferred between 2D simulation and a real robot. They define a DSL that involves visual perception with shifting attention, working memory, new object imagination, and object manipulation. We use a DSL that similarly involves shifting attention and object-based (grid cell-based) actions. However, in this work, we focus on the problem of *efficient inference* for learning programs that specify policies, and make use of full demonstrations of a policy (instead of start/end configurations alone). We compare our inference method against enumeration in experiments.

3 The PLP Policy Class

Our goal in this work is to learn policies π where $\pi(a|s)$ gives a distribution over discrete actions $a \in \mathcal{A}$ conditioned on a discrete state $s \in \mathcal{S}$. In many domains of interest, there are often multiple optimal (or at least good) actions for a given state. We therefore parameterize policies π with a classifier $h_\pi : \mathcal{S} \times \mathcal{A} \mapsto \{0, 1\}$ where $h_\pi(s, a) = 1$ if a is among the optimal (or good) actions to take in state s. In other words, $\pi(a|s) \propto h(s, a)$, i.e., $\pi(a|s)$ is a uniform distribution over the support $\{a : h_\pi(s, a) = 1\}$. Often there will be a unique a for which $h_\pi(s, a) = 1$; in this case, $\pi(a|s)$ is deterministic. If $h_\pi(s, a) = 0$ for all a, then $\pi(a|s)$ is uniform over all actions.

A classifier h_π for a policy π in PLP is a Boolean combination of features. Without loss of generality, we can express h_π in disjunctive normal form:

$$h_\pi(s, a) \triangleq (f_{1,1}(s, a) \wedge ... \wedge f_{1,n_1}(s, a)) \vee ... \vee (f_{m,1}(s, a) \wedge ... \wedge f_{m,n_m}(s, a))$$

where each function $f_{i,j}(s, a)$ is a (possibly negated) Boolean feature of the input state-action. The feature detectors $f_{i,j} : \mathcal{S} \times \mathcal{A} \to \{0, 1\}$ are implemented as programs written in a domain-specific

language. The classifier h_π then uses these features to make a judgment about whether the action a should be taken in state s.

3.1 Domain-Specific Language

Method	Type	Description
`cell_is_value`	$\text{V} \to \text{C}$	Check whether the attended cell has a given value
`shifted`	$\text{O} \times \text{C} \to \text{C}$	Shift attention by an offset, then check a condition
`scanning`	$\text{O} \times \text{C} \times \text{C} \to \text{C}$	Repeatedly shift attention by the given offset, and check which of two conditions is satisfied first
`at_action_cell`	$\text{C} \to \text{P}$	Attend to the candidate action cell and check a condition
`at_cell_with_value`	$\text{V} \times \text{C} \to \text{P}$	Attend to a cell with the given value and check a condition

Table 1: Methods of the domain-specific language (DSL) used in this work. A *program* (P) in the DSL implements a predicate on state-action pairs (i.e., $\text{P} = \mathcal{S} \times \mathcal{A} \to \{0, 1\}$), by attending to a certain cell, then running a *condition* (C). Conditions check that some property holds of the current state *relative* to an implicit attention pointer. V ranges over possible grid cell values and an "off-screen" token, and O over "offsets," which are pairs (x, y) of integers specifying horizontal and vertical displacements.

The specific DSL we use in this work (Table 1) is inspired by early work in visual routines [Ullman, 1987] and deictic references [Agre and Chapman, 1987, Ballard et al., 1997]. Each program $f_{i,j}$ takes a state and action as input and returns a Boolean value. The program implements a procedure for attending to some grid cell and checking that a local condition holds nearby. (This locality bias is similar to, but more flexible than, the locality bias built into a convolutional neural network.) Recall that in our tasks, states are full grid layouts and actions are single grid cells ("clicks"). Given input (s, a), a program can initialize its attention pointer either to the grid cell in s associated with action a (`at_action_cell`), or to an arbitrary grid cell containing a certain value (`at_cell_with_value`). For example, a program in "Reach for the Star" may start by attending to the potential click location a, or to the grid cell with the star. Next, the program will check a condition at or near the attended cell. The simplest condition is `cell_is_value`, which checks if the attended cell has a certain value. In "Reach for the Star," we could write a program that checks whether the candidate action cell is a blank square as follows: `at_action_cell(cell_is_value(□))`. More complex conditions, which look not just *at* but *near* the attended cell, can be built up using the `shifted` and `scanning` methods. The `shifted` method builds a condition that first shifts the attention pointer by some offset, then applies another condition. The `scanning` method is more sophisticated: starting at the currently attended cell, the program "scans" along some direction, repeatedly shifting the attention pointer by a specified offset and checking if either of two other conditions hold. If, while scanning, the first condition becomes satisfied before the second, the `scanning` condition returns 1. Otherwise, it returns 0. This method could be used, for example, to check whether all cells below the star contain blocks. Thus the overall DSL contains five methods, which are summarized in Table 1.

See Figure 3 for a complete example of a policy in `PLP` using the DSL described above. This DSL is well suited for the grid-based strategy games considered in this work. Attention shifts, scanning, indexing on object types, and a locality bias are also likely to apply in other domains with multiple interacting objects and spatial layouts. Other tasks that differ significantly in structure, e.g. card games or arithmetic problems, would require a different DSL. In designing a DSL, one should strive to make all necessary programs as short as possible while keeping the overall language as small as possible so that the time required to enumerate all necessary programs is minimized.

4 Policy Learning as Approximate Bayesian Inference

We now describe an approximate Bayesian inference algorithm for imitation learning of policies in `PLP` from demonstrations $\mathcal{D}$. We will define the prior $\mathsf{p}_\pi(\pi)$ and the likelihood $\mathsf{p}_{\mathsf{D}|\pi}(\mathcal{D} \mid \pi)$ and then describe an approximate inference algorithm for approximating the posterior $\mathsf{p}_{\pi|\mathsf{D}}(\pi \mid \mathcal{D}) \propto \mathsf{p}_{\mathsf{D}|\pi}(\mathcal{D} \mid \pi)\mathsf{p}_\pi(\pi)$. We can then use the approximate posterior over policies to derive a state-conditional posterior over actions, giving us a final stochastic policy π_*. See Algorithm 1 in the appendix for pseudocode.

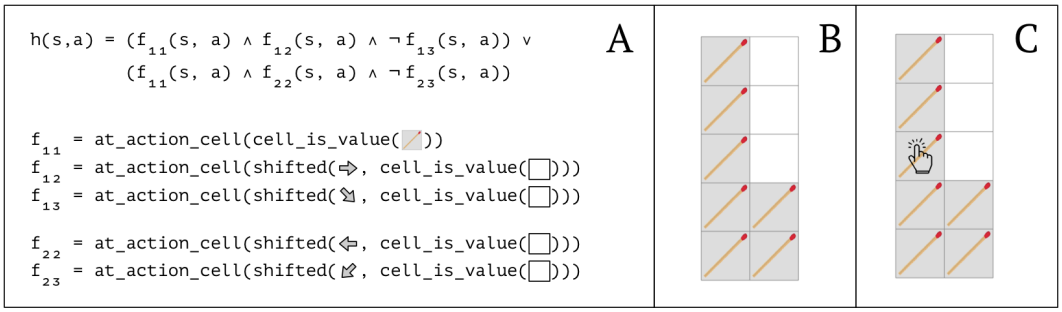

Figure 3: Example of a policy in PLP for the "Nim" game. (A) $h(s, a)$ is a logical combination of programs from a DSL. The induced policy is $\pi(a \mid s) \propto h(s, a)$. (B) Given state s, (C) there is one action selected by h. This policy encodes the "leveling" tactic, which wins the game.

4.1 Policy Prior $\mathsf{p}_\pi(\pi)$

Production rule	Probability
Programs	
P $\to$ `at_cell_with_value`(V, C)	0.5
P $\to$ `at_action_cell`(C)	0.5
Conditions	
C $\to$ `shifted`(O, B)	0.5
C $\to$ B	0.5
Base conditions	
B $\to$ `cell_is_value`(V)	0.5
B $\to$ `scanning`(O, C, C)	0.5
Offsets	
O $\to$ (N, 0)	0.25
O $\to$ (0, N)	0.25
O $\to$ (N, N)	0.5
Numbers	
N $\to \mathbb{N}$	0.5
N $\to -\mathbb{N}$	0.5
Natural numbers (for $i = 1, 2, \dots$)	
$\mathbb{N} \to i$	$(0.99)(0.01)^{i-1}$
Values (for each value v in this game)	
V $\to$ v	$1/\lvert\text{V}\rvert$

Table 2: The prior p_f over programs, specified as a probabilistic context-free grammar (PCFG).

Recall that a policy $\pi \in$ PLP is defined in terms of many Boolean programs $f_{i,j}$ written in a DSL. We factor the prior probability of a policy π into a product of priors over its constituent programs: $\mathsf{p}_\pi(\pi) \propto \prod_{i=1}^{M} \prod_{j=1}^{N_i} \mathsf{p}_\mathsf{f}(f_{i,j})$. (Note that without some maximum limit on $\sum_{i=1}^{M} N_i$, this is an improper prior, and for this technical reason, we introduce a uniform prior on $\sum_{i=1}^{M} N_i$, between 1 and a very high maximum value α; the resulting factor of $\frac{1}{\alpha}$ does not depend on π at all, and can be folded into the proportionality constant.)

If there were a finite number of programs and p_f were uniform, this policy prior would favor policies with the fewest programs [Rissanen, 1978]. But we are instead in the regime where there are an infinite number of possible programs and some are much simpler than others. We thus define a prior

p_f over programs $f_{i,j}$ using a probabilistic grammar [Manning et al., 1999]. Given a probabilistic grammar, it is straightforward to enumerate programs with monotonically decreasing probability via best-first search. We take advantage of this property during inference. The specific grammar we use in this work is shown in Table 2.

Overall, our prior encodes a preference for policies that use fewer and simpler programs (feature detectors). Moreover, these are the only preferences it encodes: two policies that use the exact same programs $f_{i,j}$ but in a different order, or with different logical connectives (and's, or's, and not's), are given equal prior probability.

4.2 Likelihood $\mathsf{p}_{\mathsf{D}|\pi}(\mathcal{D} \mid \pi)$

We now describe the likelihood $\mathsf{p}_{\mathsf{D}|\pi}(\mathcal{D} \mid \pi)$. Let $(s_0, a_0, ..., s_{T-1}, a_{T-1}, s_t)$ be a demonstration in $\mathcal{D}$. The probability of this demonstration given π is $\mathsf{p}(s_0) \prod_{t=0}^{T-1} \mathsf{p}(s_{t+1} \mid s_t, a_t)\pi(a_t \mid s_t)$ where $\mathsf{p}(s_{t+1} \mid s_t, a_t)$ is the (unknown) transition distribution. As a function of π, the likelihood of the demonstration is thus proportional to $\prod_{t=0}^{T-1} \pi(a_t \mid s_t)$. Given a set $\mathcal{D}$ of N demonstrations of the same policy π, the likelihood is then $\mathsf{p}_{\mathsf{D}|\pi}(\mathcal{D} \mid \pi) \propto \prod_{i=1}^{N} \prod_{j=1}^{T_i} \pi(a_{ij} \mid s_{ij})$.

4.3 Approximating the Posterior $\mathsf{p}_{\pi|\mathsf{D}}(\pi \mid \mathcal{D})$

We now have the prior $\mathsf{p}_\pi(\pi)$ and the likelihood $\mathsf{p}_{\mathsf{D}|\pi}(\mathcal{D} \mid \pi)$ and we wish to compute the posterior $\mathsf{p}_{\pi|\mathsf{D}}(\pi \mid \mathcal{D}) \propto \mathsf{p}_{\mathsf{D}|\pi}(\mathcal{D} \mid \pi)\mathsf{p}(\pi)$. Computing this posterior exactly is intractable, so we will approximate it with a distribution q. Formally, our scheme is a variational inference algorithm, which iteratively minimizes the KL divergence from q to the true posterior. We take q to be a finite mixture of K policies $\mu_1 \dots \mu_K$ (in our experiments, $K = 25$), and initialize it so that each μ_i is the uniform policy, $\mu_i(a \mid s) \propto 1$. Our core insight is a way to take advantage of the structure of PLP to efficiently search the space of policies and update q to better match the posterior.

Recall that a policy π is determined by a function h_π which takes a state-action pair (s, a) as input and produces a Boolean output indicating whether π can take action a in state s. We can therefore reduce the problem of policy learning to one of classification. Positive examples are supplied by the demonstrations themselves; negative examples can be derived by considering actions *not* taken from a state s in *any* of the demonstrations, that is, $\{(s', a') : \exists(s', \cdot) \in \mathcal{D} \text{ and } \forall(s, a) \in \mathcal{D}, s' = s \implies a \neq a'\}$. This reduction to classification is approximate; if multiple actions are permissible by the expert policy, this classification dataset will include actions mislabelled as negative (but they will generally constitute only a small fraction of the negative examples).

Recall further that the classifier h_π is represented as a logical combination of programs $f_{i,j}$ drawn from a DSL, each of which maps state-action pairs (s, a) to Booleans. As such, h_π depends on its input only through terms of the form $f_{i,j}(s, a)$; we can think of these as *binary features* used by the classifier. Given a finite set of programs $f_1, \dots, f_n$, we can create vectors $\mathbf{x}$ of n binary features for each training example (state-action pair) in our classification dataset, by running each of the n programs on each state-action pair. We then have a binary classification problem with n-dimensional Boolean inputs $\mathbf{x}$. The problem of learning a binary classifier as a logical combination of binary features is very well understood [Mitchell, 1978, Valiant, 1985, Quinlan, 1986, Dietterich and Michalski, 1986, Haussler, 1988, Dechter and Mateescu, 2007, Wang et al., 2017]. A fast approximate method that we use in this work is greedy decision-tree learning. This method tends to learn small classifiers, which are hence likely according to our prior. We can then view the decision-tree learner as performing a quick search for a policy that explains the observed demonstrations; indeed, it is straightforward to derive from the learned classifier a proposed Boolean combination of programs h_π and a corresponding policy π.

Although this learning algorithm finds policies with high posterior likelihood, there are two remaining problems to resolve: first, we must decide on a finite set of programs (features) to feed to the decision tree learner, out of the infinitely many programs our DSL permits; and second, the decision tree learner knows nothing about p_f, which encodes our preference for simpler component programs. To the classifier, every program is just a feature. We resolve these problems by using the classifier only as a heuristic, to help us update our posterior approximation q.

In particular, we iteratively enumerate programs f_i in best-first order from our PCFG $\mathsf{p_f}$. On each iteration, we take the currently enumerated set of programs $\{f_1, \ldots, f_i\}$, and use a stochastic greedy decision tree learning algorithm to find several (in our experiments, $P = 5$) candidate policies $\mu'_1, \ldots, \mu'_P$ which use only the already-enumerated programs. For each, we check whether $\mathsf{p}_{\pi,\mathsf{D}}(\mu', \mathcal{D}) = \mathsf{p}_\pi(\mu')\mathsf{p}_{\mathsf{D}|\pi}(\mathcal{D} \mid \mu')$ is higher than the score of any existing policy in our mixture q. If it is, we replace the lowest-scoring policy in q with our newly found policy μ'. We then update the weights of each policy μ in the support of q so that $\mathsf{q}(\mu) = \frac{\mathsf{p}_{\pi,\mathsf{D}}(\mu,\mathcal{D})}{\sum_{i=1}^{K} \mathsf{p}_{\pi,\mathsf{D}}(\mu_i,\mathcal{D})}$. The process of replacing one policy with a higher-scoring policy and reweighting the entire mixture is guaranteed to decrease the KL divergence $D(\mathsf{q} \mid\mid \mathsf{p}_{\pi|\mathsf{D}})$.

We can stop after a fixed number of iterations, or after the enumerated program prior probabilities fall below a threshold: if at step i, $\mathsf{p_f}(f_i)$ is smaller than the joint probability $\mathsf{p}_{\pi,\mathsf{D}}(\mu, \mathcal{D})$ of the lowest-weight policy $\mu \in \mathsf{q}$, there is no point in enumerating it, as any policy that uses it will necessarily have too low a joint probability to merit inclusion in q.

4.4 Deriving the Final Policy π_*

Once we have $\mathsf{q} \approx \mathsf{p}_{\pi|\mathsf{D}}(\pi \mid \mathcal{D})$, we use it to derive a final policy

$$\pi_*(s) = \arg\max_{a \in \mathcal{A}} \mathbb{E}_{\mathsf{q}}[\pi(a \mid s)] = \arg\max_{a \in \mathcal{A}} \sum_{\pi \in \Pi} \mathsf{q}(\pi)\pi(a \mid s) = \arg\max_{a \in \mathcal{A}} \sum_{\mu \in \mathsf{q}} \mathsf{q}(\mu)\mu(a|s).$$

We also note the possibility of using the full posterior distribution over actions to guide exploration, e.g., in combination with reinforcement learning Hester et al. [2018], Nair et al.. In this work, however, we focus on exploitation and therefore require only the MAP actions and a deterministic final policy.

5 Experiments and Results

We now study the extent to which policies in PLP can be learned from a few demonstrations. For each of five tasks, we evaluate PLP as a function of the number of demonstrations in the training set and the number of programs enumerated. We examine the performance of the learned policies empirically and qualitatively and analyze the complexity of the discovered programs. We further compare the performance of the learned policies against several baselines including two different types of convolutional networks, local linear policies, and program-based policies learned via enumeration. We then perform ablation studies to dissect the contributions of the main components of PLP: the programmatic feature detectors, the decision tree learning, and the probabilistic grammar prior.

5.1 Tasks

We consider five diverse strategy games (Figure 2) that share a common state space ($\mathcal{S} = \bigcup_{H=1}^{\infty} \bigcup_{W=1}^{\infty} \{1, \ldots, V\}^{H \times W}$; variable-sized grids with discrete-valued cells) and action space ($\mathcal{A} = \mathbb{N}^2$; single "clicks" on any cell). These tasks feature high variability between different task instances; learning a robust policy requires substantial generalization. The tasks are also very challenging from the perspective of planning and reinforcement learning due to the unbounded action space, the absence of shaping or auxiliary rewards, and the arbitrarily long horizons that may be required to solve a task instance. In practice we evaluate policies for 60 time steps and count the episode as a loss if a win is not achieved within that horizon. We provide brief descriptions of the five tasks here and extended details in the appendix.

Nim: This classic game starts with two arbitrarily tall piles of matchsticks. Two players take turns; for our purposes, the second player is modeled as part of the environment and plays optimally. On each turn, the player chooses a pile and removes one or more matchsticks from that pile. The player who removes the *last* matchstick wins.

Checkmate Tactic: This game involves a simple checkmating pattern from Chess featuring a white king and queen and a black king. Pieces move as in chess, but the board may be an arbitrarily large or small rectangle. As in Nim, the second player (black) is modeled as part of the environment.

Chase: Here a rabbit runs away from a stick figure, but cannot move beyond a rectangular enclosure of bushes. The objective is to move the stick figure (by clicking on arrow keys) to catch the rabbit. Clicking on an empty cell creates a static wall. The rabbit can only be caught if it is trapped by a wall.

Stop the Fall: In this game, a parachuter falls down after a green button is clicked. Blocks can be built by clicking empty cells; they too are influenced by gravity. The game is won only if the parachuter is not directly adjacent to fire after the green button is clicked.

Reach for the Star: In this final game, a robot must navigate (via left and right arrow key clicks) to a star above its initial position. The only way to move vertically is to climb a block, which can be created by clicking an empty cell. Blocks are under the influence of gravity. Winning the game requires building and climbing stairs from the floor to the star.

Each task has 20 instances, which are randomly split into 11 training and 9 test. The instances for Nim, Checkmate Tactic, and Reach for the Star are procedurally generated; the instances for Stop the Fall and Chase are manually generated, as the variation between instances is not trivially parameterizable. See the appendix for more details.

5.2 Baselines

We compare PLP against four baselines: two from deep learning, one using local features alone, and one from program induction. We selected these baselines to encode varying degrees of domain-specific structure. For the deep and linear baselines, we pad the inputs so that all states are the same maximal width and height.

The first baseline ("Local Linear") uses a linear classifier on local features to determine whether to take an action. For each cell, it learns a boolean function of the values for the 8 surrounding cells, indicating whether to click the cell or not. We implement this as learning a single 3×3 filter which is convolved with the grid. Training data is then derived for each cell from the expert demonstrations: 1 if that cell was clicked in the expert demonstration, and 0 otherwise.

The second baseline ("FCN") extends the local model by considering a deeper fully convolutional network [Long et al., 2015]. The network has 8 layers which each have a kernel size of 3, stride 1 and padding 1. The number of channels is fixed to 4 for every layer except the first, which contains 8 channels. Each layer is separated by a ReLU nonlinearity. This architecture was chosen to reflect the receptive field sizes we expect are necessary to learn programs to solve the different games. It was trained in the same way as the local linear model above.

The third baseline ("CNN") is a standard convolutional neural network. Whereas the outputs for the first two baselines are binary per cell, the CNN outputs logits over actions as in multiclass classification. The architecture is: 64-channel convolution; max pooling; 64-channel fully-connected layer; $|\mathcal{A}|$-channel fully-connected layer. All kernels have size 3 and all strides and paddings are 1.

The fourth baseline ("Enumeration") is full policy enumeration. The grammar in Table 2 is extended to include logical disjunctions, conjunctions, and negations over constituent programs so that full policies are enumerated in DNF form. The number of disjunctions and conjunctions each follow a geometric distribution ($p = 0.5$). (Several other values of p were also tried without improvement.) Policies are then enumerated and mixed as in PLP learning; this baseline is thus identical to PLP learning but with the greedy Boolean learning removed.

5.3 Learning and Generalizing from Demonstrations

5.3.1 Effect of Number of Demonstrations

We first evaluate the test-time performance of PLP and baselines as the number of training demonstrations varies from 1 to 10. For each number of demonstrations, we run 10 trials, each featuring a distinct set of demonstrations drawn from the overall pool of 11 training instances. PLP learning is run for 10,000 iterations for each task. The mean and maximum trial performance offer complementary insight: the mean reflects the expected performance if demonstrations were selected at random; the maximum reflects the expected performance if the most useful demonstrations were selected, perhaps by an expert teacher (c.f. Shafto et al. [2014]).

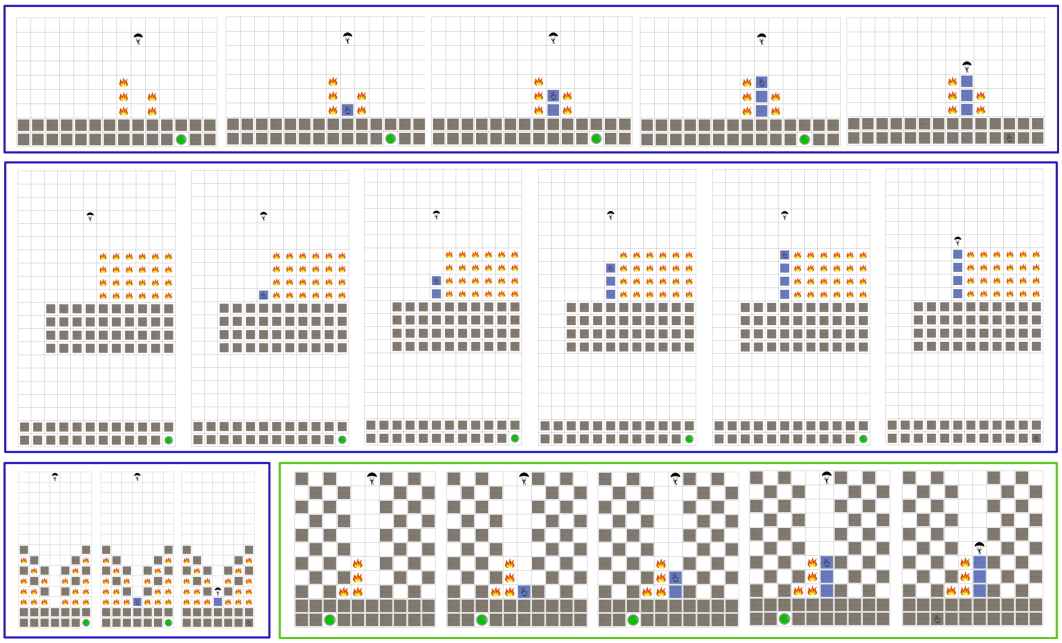

Figure 4: A PLP policy learned from three demonstrations of "Stop the Fall" (blue) generalizes perfectly to all test task instances (e.g. green). See the appendix for analogous results in the other four tasks.

```
at_action_cell(scanning(⇧, cell_is_value(☂), cell_is_value(■))) ∧
at_action_cell(scanning(⇧, cell_is_value(☂), cell_is_value(■))) ∧
at_action_cell(cell_is_value(□)) ∧
not at_action_cell(shifted(⇩, cell_is_value(■))) ∧
not at_action_cell(cell_is_value(●)) ∧
at_action_cell(shifted(⇩, cell_is_value(■)))
```

Figure 5: One of the clauses of the learned MAP policy for "Stop the Fall". The clause suggests clicking a cell if scanning up we find a parachuter before a drawn or static block; if the cell is empty; if the cell above is not drawn; if the cell is not a green button; and if a static block is immediately below. Note that this clause is slightly redundant and may trigger an unnecessary action for a task instance where there are no "fire" cells.

Results are shown in Figure 6. On the whole, PLP markedly outperforms all baselines, especially on the more difficult tasks (Chase, Stop the Fall, Reach for the Star). The baselines are limited for different reasons. The highly parameterized CNN baseline is able to perfectly fit the training data and win all *training* games (not shown), but given the limited training data and high variation from training to task, it severely overfits and fails to generalize. The FCN baseline is also able to fit the training data almost perfectly. Its additional structure permits better generalization in Nim, Checkmate Tactic, and Reach for the Star, but overall its performance is still far behind PLP. In contrast, the Local Linear baseline is unable to fit the training data; with the exception of Nim, its training performance is close to zero. Similarly, the training performance of the Enumeration baseline is near or at zero for all tasks beyond Nim. In Nim, there is evidently a low complexity program that works roughly half the time, but an optimal policy is more difficult to enumerate.

5.3.2 Effect of Number of Programs Searched

We now examine test-time performance of PLP and Enumeration as a function of the number of programs searched. For this experiment, we give both methods all 11 training demonstrations for

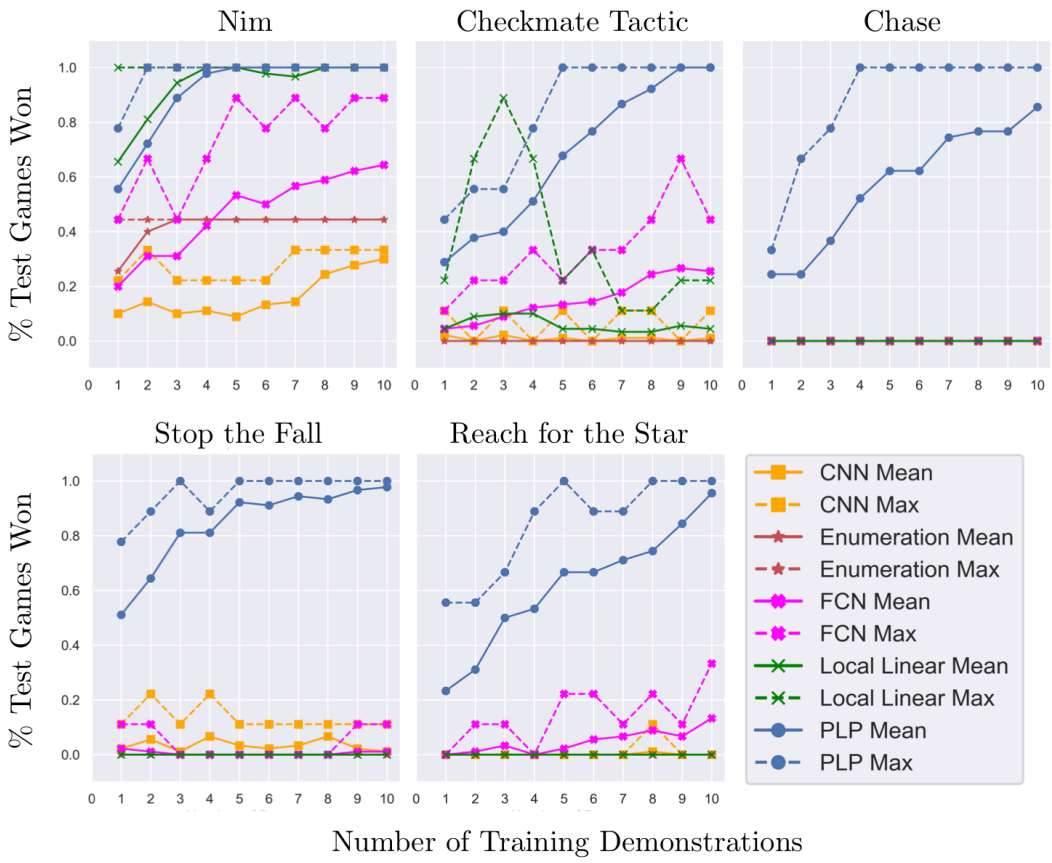

Figure 6: Performance on held-out test task instances as a function of the number of demonstrations comprising the training data for PLP (ours) and four baselines. Maximums and means are taken over 10 training sets.

each task. Results are shown in Figure 7. PLP requires fewer than 100 programs to learn a winning policy for Nim, fewer than 1,000 for Checkmate Tactic and Chase, and fewer than 10,000 for Stop the Fall and Reach for the Star. In contrast, Enumeration is unable to achieve nonzero performance for any task other than Nim, for which it achieves roughly 45% performance after 100 programs enumerated.

The lackluster performance of Enumeration is unsurprising given the combinatorial explosion of programs. For example, the optimal policy for Nim shown in Figure 3 involves six constituent programs, each with a parse tree depth of three or four. There are 108 unique constituent programs with parse tree depth three and therefore more than 13,506,156,000 full policies with 6 or fewer constituent programs. Enumeration would have to search roughly so many programs before arriving at a winning policy for Nim, which is by far the simplest task. In contrast, a winning PLP policy is learnable after fewer than 100 enumerations. In practical terms, PLP learning for Nim takes on the order of 1 second on a laptop without highly optimized code; after running Enumeration for six hours in the same setup, a winning policy is still not found.

5.3.3 Learned Policy Analysis

Here we analyze the learned PLP policies in an effort to understand their representational complexity and qualitative behavior. We enumerate 10,000 programs and train with 11 task instances for each task. Table 3 reports three statistics for the MAP policies learned for each task: the number of top-level programs (`at_action_cell` and `at_cell_with_value` calls), the total number of method calls (`shifted`, `scanning`, and `cell_is_value` as well); and the maximum parse tree depth among all programs in the policy. The latter term is the primary driver of learning complexity in practice. The

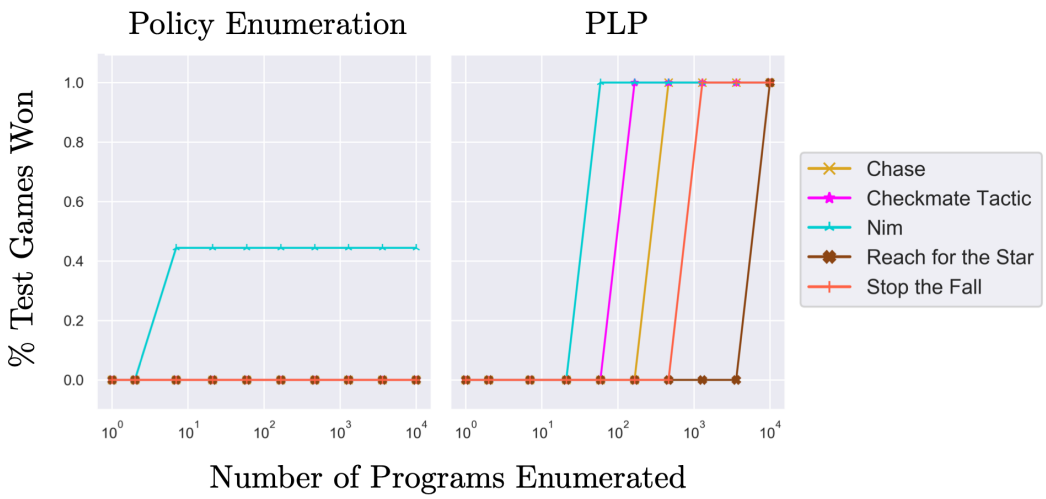

Figure 7: Performance on held-out test task instances as a function of the number of programs enumerated for PLP (ours) and the Enumeration baseline.

Task	Number of Programs	Method Calls	Max Depth
Nim	11	32	4
Checkmate Tactic	23	60	3
Chase	97	244	3
Stop the Fall	17	50	4
Reach for the Star	34	151	4

Table 3: Analysis of the MAP policies learned in PLP with 10,000 programs enumerated and all 11 training demonstrations. The number of top-level programs, the number of constituent method calls, and the maximum parse depth of a program in the policy is reported for each of the five tasks.

number of top-level programs ranges from 11 to 97; the number of method calls ranges from 32 to 244; and the maximum parse tree depth is 3 or 4 in all tasks. These numbers suggest that PLP learning is capable of discovering policies of considerable complexity, far more than what would be feasible with full policy enumeration. We also note that the learned policies are very sparse, given that the maximum number of available top-level programs is 10,000. Visualizing the policies (Figure 4; also see appendix) offers further confirmation of their strong performance.

An example of a clause in the learned MAP policy for Stop the Fall is shown in Figure 5. This clause is used to select the first action: the cell below the parachuter and immediately above the floor is clicked. Other clauses govern the policy following this first action, either continuing to build blocks above the first one, or clicking the green button to turn on gravity. From the example clause, we see that the learned policy may include some redundant programs, which unnecessarily decrease the prior without increasing the likelihood. Such redundancy is likely due to the approximate nature of our Boolean learning algorithm (greedy decision-tree learning). Posthoc pruning of the policies or alternative Boolean learning methods could address this issue. The example clause also reveals that the learned policy may take unnecessary (albeit harmless) actions in the case where there are no "fire" objects to avoid. In examining the behavior of the learned policies for Stop the Fall and the other games, we do not find any unnecessary actions taken, but this does not preclude the possibility in general. Despite these two qualifications, the clause and others like it are a positive testament to the interpretability and intuitiveness of the learned policies.

	Nim	CT	Chase	STF	RFTS
PLP	1.0	1.0	1.0	1.0	1.0
Features + NN	1.0	0.67	0.0	0.0	0.22
Features + NN + L_1 Reg	1.0	0.11	0.0	0.0	0.0
No Prior	1.0	0.44	0.78	1.0	1.0
Sparsity Prior	1.0	0.78	1.0	0.78	1.0

Figure 8: Performance on held-out test task instances with 10,000 programs enumerated and all 11 training demonstrations for PLP and four ablation models.

5.4 Ablation Studies

In the preceding experiments, we have seen that winning PLP policies can be learned from a few demonstrations in all five tasks. We now perform ablation studies to explore which aspects of the PLP class and learning algorithm contribute to the strong performance. We consider four ablated models.

The "Features + NN" model learns a neural network state-action binary classifier on the first 10,000 feature detectors enumerated from the domain-specific language. This model addresses the possibility that the features alone are powerful enough to solve the task when combined with a simple classifier. The neural network is a multilayer perceptron with two layers of dimension 100 and ReLU activations. The "Features + NN + L_1 Regularization" model is identical to the previous baseline except that an L_1 regularization term is added to the loss to encourage sparsity. This model addresses the possibility that the features alone suffice when we incorporate an Occam's razor bias similar to the one that exists in PLP learning. The "No Prior" model is identical to PLP learning, except that the probabilistic grammatical prior is replaced with a uniform prior. Similarly, the "Sparsity Prior" model replaces the prior with a prior that penalizes the number of top-level programs involved in the policy, without regard for the relative priors of the individual programs.

Results are presented in Figure 8. We see that the Features + NN and Features + NN + L_1 Regularization models suffice for Nim, but break down in the more complex tasks. The No Prior and Sparsity Prior models do better, performing optimally on three out of five tasks. The Sparsity Prior model performs best overall but still lags behind PLP. These results confirm that the main components — the feature detectors, the sparsity regularization, and the probabilistic grammatical prior — each add value to the overall framework.

6 Discussion and Conclusion

In an effort to efficiently learn policies from very few demonstrations that generalize substantially from training to test, we have introduced the PLP policy class and an approximate Bayesian inference algorithm for imitation learning. We have seen that the PLP policy class includes winning policies for a diverse set of strategy games, and moreover, that those policies can be efficiently learned from five or fewer demonstrations. We have also confirmed that it would be intractable to learn the same policies with enumeration, and that deep convolutional policies are prone to severe overfitting in this low data regime. In ablation studies, we found that each of the main components of the PLP representation and the learning algorithm contribute substantively to the overall strong performance.

Beyond imitation and policy learning, this work contributes to the long and ongoing discussion about the role of prior knowledge in machine learning and artificial intelligence. In the common historical narrative, early attempts to incorporate prior knowledge via manual feature engineering failed to scale, leading to the modern shift towards domain-agnostic deep learning methods [Sutton, 2019]. Now there is renewed interest in incorporating structure and inductive bias into contemporary methods, especially for problems where data is scarce or expensive. Convolutional neural networks are often cited as a testament to the power of inductive bias due to their locality structure and translational invariance. There is a growing call for similar models with additional inductive biases and a mechanism for exploiting them. We argue that encoding prior knowledge via a probabilistic grammar over feature detectors and learning to combine these feature detectors with Boolean logic is a promising path forward. More generally, we submit that "meta-feature engineering" of the sort exemplified here strikes an appropriate balance between the strong inductive bias of classical AI and the flexibility and scalability of modern methods.

Acknowledgments

We thank Anurag Ajay and Ferran Alet for helpful discussions and feedback on earlier drafts. We gratefully acknowledge support from NSF grants 1523767 and 1723381; from ONR grant N00014-13-1-0333; from AFOSR grant FA9550-17-1-0165; from ONR grant N00014-18-1-2847; from Honda Research; ; from the MIT-Sensetime Alliance on AI. and from the Center for Brains, Minds and Machines (CBMM), funded by NSF STC award CCF-1231216. KA acknowledges support from NSERC. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

Noah D Goodman, Joshua B Tenenbaum, Jacob Feldman, and Thomas L Griffiths. A rational analysis of rule-based concept learning. *Cognitive Science*, 32(1):108–154, 2008.

Noah D Goodman, Joshua B Tenenbaum, and Tobias Gerstenberg. Concepts in a probabilistic language of thought. Technical report, Center for Brains, Minds and Machines (CBMM), 2014.

Steven T Piantadosi, Joshua B Tenenbaum, and Noah D Goodman. The logical primitives of thought: Empirical foundations for compositional cognitive models. *Psychological Review*, 123(4):392, 2016.

David Wingate, Carlos Diuk, Timothy O'Donnell, Joshua Tenenbaum, and Samuel Gershman. Compositional policy priors. Technical report, Massachusetts Institute of Technology, 2013.

Danfei Xu, Suraj Nair, Yuke Zhu, Julian Gao, Animesh Garg, Li Fei-Fei, and Silvio Savarese. Neural task programming: Learning to generalize across hierarchical tasks. *International Conference on Robotics and Automation*, 2018.

Miguel Lázaro-Gredilla, Dianhuan Lin, J. Swaroop Guntupalli, and Dileep George. Beyond imitation: Zero-shot task transfer on robots by learning concepts as cognitive programs. *Science Robotics*, 4 (26), 2019.

Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

Dean A Pomerleau. Efficient training of artificial neural networks for autonomous navigation. *Neural Computation*, 3(1):88–97, 1991.

Stefan Schaal. Learning from demonstration. *Advances in Neural Information Processing Systems*, 1997.

Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. *International Conference on Machine Learning*, 2004.

Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, et al. Deep Q-learning from demonstrations. *AAAI Conference on Artificial Intelligence*, 2018.

Ashvin Nair, Bob McGrew, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Overcoming exploration in reinforcement learning with demonstrations. *International Conference on Robotics and Automation*.

Yan Duan, Marcin Andrychowicz, Bradly Stadie, Jonathan Ho, Jonas Schneider, Ilya Sutskever, Pieter Abbeel, and Wojciech Zaremba. One-shot imitation learning. *Advances in Neural Information Processing Systems*, 2017.

Chelsea Finn, Tianhe Yu, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. One-shot visual imitation learning via meta-learning. *Conference on Robot Learning*, 2017.

David Andre and Stuart J Russell. State abstraction for programmable reinforcement learning agents. *AAAI Conference on Artificial Intelligence*, 2002.

Finale Doshi-Velez, David Wingate, Nicholas Roy, and Joshua B. Tenenbaum. Nonparametric Bayesian policy priors for reinforcement learning. *Advances in Neural Information Processing Systems*, 2010.

David Wingate, Noah D Goodman, Daniel M Roy, Leslie P Kaelbling, and Joshua B Tenenbaum. Bayesian policy search with policy priors. *International Joint Conference on Artificial Intelligence*, 2011.

Scott D Niekum. *Semantically grounded learning from unstructured demonstrations*. PhD thesis, University of Massachusetts, Amherst, 2013.

Pravesh Ranchod, Benjamin Rosman, and George Konidaris. Nonparametric Bayesian reward segmentation for skill discovery using inverse reinforcement learning. *International Conference on Intelligent Robots and Systems*, 2015.

Christian Daniel, Herke Van Hoof, Jan Peters, and Gerhard Neumann. Probabilistic inference for determining options in reinforcement learning. *Machine Learning*, 104(2-3):337–357, 2016.

Sanjay Krishnan, Roy Fox, Ion Stoica, and Ken Goldberg. DDCO: Discovery of deep continuous options for robot learning from demonstrations. *Conference on Robot Learning*, 2017.

Joshua Brett Tenenbaum. *A Bayesian framework for concept learning*. PhD thesis, Massachusetts Institute of Technology, 1999.

Joshua B Tenenbaum and Thomas L Griffiths. Generalization, similarity, and Bayesian inference. *Behavioral and brain sciences*, 24(4):629–640, 2001.

Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally–guided Bayesian program induction. *Advances in Neural Information Processing Systems*, 2018a.

Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. *Advances in Neural Information Processing Systems*, 2018b.

Shimon Ullman. Visual routines. *Readings in Computer Vision*, pages 298–328, 1987.

Philip E Agre and David Chapman. Pengi: An implementation of a theory of activity. *AAAI Conference on Artificial Intelligence*, 1987.

Dana H Ballard, Mary M Hayhoe, Polly K Pook, and Rajesh PN Rao. Deictic codes for the embodiment of cognition. *Behavioral and Brain Sciences*, 20(4):723–742, 1997.

Jorma Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

Christopher D Manning, Christopher D Manning, and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT press, 1999.

Tom Michael Mitchell. Version spaces: an approach to concept learning. Technical report, Stanford University, 1978.

Leslie G Valiant. Learning disjunction of conjunctions. *International Joint Conference on Artificial Intelligence*, 1985.

J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

Thomas G Dietterich and Ryszard S Michalski. Learning to predict sequences. *Machine learning: An artificial intelligence approach*, 1986.

David Haussler. Quantifying inductive bias: AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, 36(2):177–221, 1988.

Rina Dechter and Robert Mateescu. And/or search spaces for graphical models. *Artificial Intelligence*, 171(2-3):73–106, 2007.

Tong Wang, Cynthia Rudin, Finale Doshi-Velez, Yimin Liu, Erica Klampfl, and Perry MacNeille. A Bayesian framework for learning rule sets for interpretable classification. *The Journal of Machine Learning Research*, 18(1):2357–2393, 2017.

Patrick Shafto, Noah D Goodman, and Thomas L Griffiths. A rational account of pedagogical reasoning: Teaching by, and learning from, examples. *Cognitive Psychology*, 71:55–89, 2014.

Richard Sutton. The bitter lesson, Mar 2019. URL `http://www.incompleteideas.net/IncIdeas/BitterLesson.html`.

7 Appendix

7.1 Approximate Bayesian Inference Pseudocode

Algorithm 1: Imitation learning as approximate Bayesian inference

input : Demonstrations $\mathcal{D}$, ensemble size K

```
/* Initialize empty lists                                                    */
state_actions, X, y, programs, program_priors, policy_particles, weights;
/* Compute all binary labels                                                 */
for (s, a*) ∈ D do
    for a ∈ A do
        state_actions.append((s, a));
        X.append([]);
        ℓ = (a == a*) ;                  // Label whether action was taken by expert
        y.append(ℓ);
    end
end
/* Iteratively grow the program set                                          */
while True do
    program, prior = generate_next_program() ;                  // Search the grammar
    programs.append(program);
    program_priors.append(prior);
    for i in 1, ..., |state_actions| do
        (s, a) = state_actions[i];
        x = program(s, a) ;                  // Apply program to get binary feature
        X[i].append(x);
    end
    /* See Algorithm 2                                                       */
    particles, iter_weights = infer_disjunctions_of_conjunctions(X, y, program_priors, K);
    policy_particles.extend(create_policies(particles, programs));
    weights.extend(iter_weights);
    /* Stop after max iters or program prior threshold is met                */
    if stop_condition() then
        return policy_particles, weights;
    end
end
```

Algorithm 2: Inferring disjunctions of conjunctions (Algorithm 1 subroutine)

input : Feature vectors X, binary labels y, feature priors p, ensemble size K

```
/* Initialize empty lists                                                    */
particles, weights;
for seed in 1, ..., K do
    tree = DecisionTree(seed) ;                  // Seed randomizes feature order
    tree.fit(X, y);
    /* Search the learned tree from root to True leaves                      */
    h = convert_tree_to_dnf(tree);
    likelihood = get_likelihood(X, y, h);
    prior = get_prior(h, p);
    weight = likelihood * prior;
    particles.append(h);
    weights.append(weight);
end
return particles, weights;
```

7.2 Environment Details

7.2.1 Nim

Task Description: There are two columns (piles) of matchsticks and empty cells. Clicking on a matchstick cell changes all cells above and including the clicked cell to empty; clicking on a empty cell has no effect. After each matchstick cell click, a second player takes a turn, selecting another matchstick cell. The second player is modeled as part of the environment transition and plays optimally. When there are multiple optimal moves, one is selected randomly. The objective is to remove the *last* matchstick cell.

Task Instance Distribution: Instances are generated procedurally. The height of the grid is selected randomly between 2 and 20. Th e initial number of matchsticks in each column is selected randomly between 1 and the height with the constraint that the two columns cannot be equal. All other grid cells are empty.

Expert Policy Description: The winning tactic is to "level" the columns by selecting the matchstick cell next to a empty cell and diagonally up from another matchstick cell. Winning the game requires perfect play.

7.2.2 Checkmate Tactic

Task Description: This task is inspired by a common checkmating pattern in Chess. Note that only three pieces are involved in this game (two kings and a white queen) and that the board size may be $H \times W$ for any H, W, rather than the standard 8×8. Initial states in this game feature the black king somewhere on the boundary of the board, the white king two cells adjacent in the direction away from the boundary, and the white queen attacking the cell in between the two kings. Clicking on a white piece (queen or king) *selects* that piece for movement on the next action. Note that a selected piece is a distinct value from a non-selected piece. Subsequently clicking on an empty cell moves the selected piece to that cell if that move is legal. All other actions have no effect. If the action results in a checkmate, the game is over and won; otherwise, the black king makes a random legal move.

Task Instance Distribution: Instances are generated procedurally. The height and width are randomly selected between 5 and 20. A column for the two kings is randomly selected among those not adjacent to the left or right sides. A position for the queen is randomly selected among all those spaces for which the queen is threatening checkmate between the kings. The board is randomly rotated among the four possible orientations.

Expert Policy Description: The winning tactic selects the white queen and moves it to the cell between the kings.

7.2.3 Chase

Task Description: This task features a stick figure agent, a rabbit adversary, walls, and four arrow keys. At each time step, the adversary randomly chooses a move (up, down, left, or right) that increases its distance from the agent. Clicking an arrow key moves the agent in the corresponding direction. Clicking a gray wall has no effect other than advancing time. Clicking an empty cell creates a new (blue) wall. The agent and adversary cannot move through gray or blue walls. The objective is to "catch" the adversary, that is, move the agent into the same cell. It is not possible to catch the adversary without creating a new wall; the adversary will always be able to move away before capture.

Task Instance Distribution: There is not a trivial parameterization to procedurally generate these task instances, so they are manually generated.

Expert Policy Description: The winning tactic advances time until the adversary reaches a corner, then builds a new wall next to the adversary so that it is trapped on three sides, then moves the agent to the adversary.

7.2.4 Stop the Fall

Task Description: This task involves a parachuter, gray static blocks, red "fire", and a green button that turns on gravity and causes the parachuter and blocks to fall. Clicking an empty cell creates a

blue static block. The game is won when gravity is turned on and the parachuter falls to rest without touching (being immediately adjacent to) fire.

Task Instance Distribution: There is not a trivial parameterization to procedurally generate these task instances, so they are manually generated.

Expert Policy Description: The winning tactic requires building a stack of blue blocks below the parachuter that is high enough to prevent contact with fire, and then clicking the green button.

7.2.5 Reach for the Star

Task Description: In this task, a robot must move to a cell with a yellow star. Left and right arrow keys cause the robot to move. Clicking on an empty cell creates a dynamic brown block. Gravity is always on, so brown objects fall until they are supported by another brown block. If the robot is adjacent to a brown block and the cell above the brown block is empty, the robot will move on top of the brown block when the corresponding arrow key is clicked. (In other words, the robot can only climb one block, not two or more.)

Task Instance Distribution: Instances are generated procedurally. A height for the star is randomly selected between 2 and 11 above the initial robot position. A column for the star is also randomly selected so that it is between 0 and 5 spaces from the right border. Between 0 and 5 padding rows are added above the star. The robot position is selected between 0 and 5 spaces away from where the start of the minimal stairs would be. Between 0 and 5 padding columns are added to the left of the agent. The grid is flipped along the vertical axis with probability 0.5.

Expert Policy Description: The winning tactic requires building stairs between the star and robot and then moving the robot up them.

7.3 Additional Results

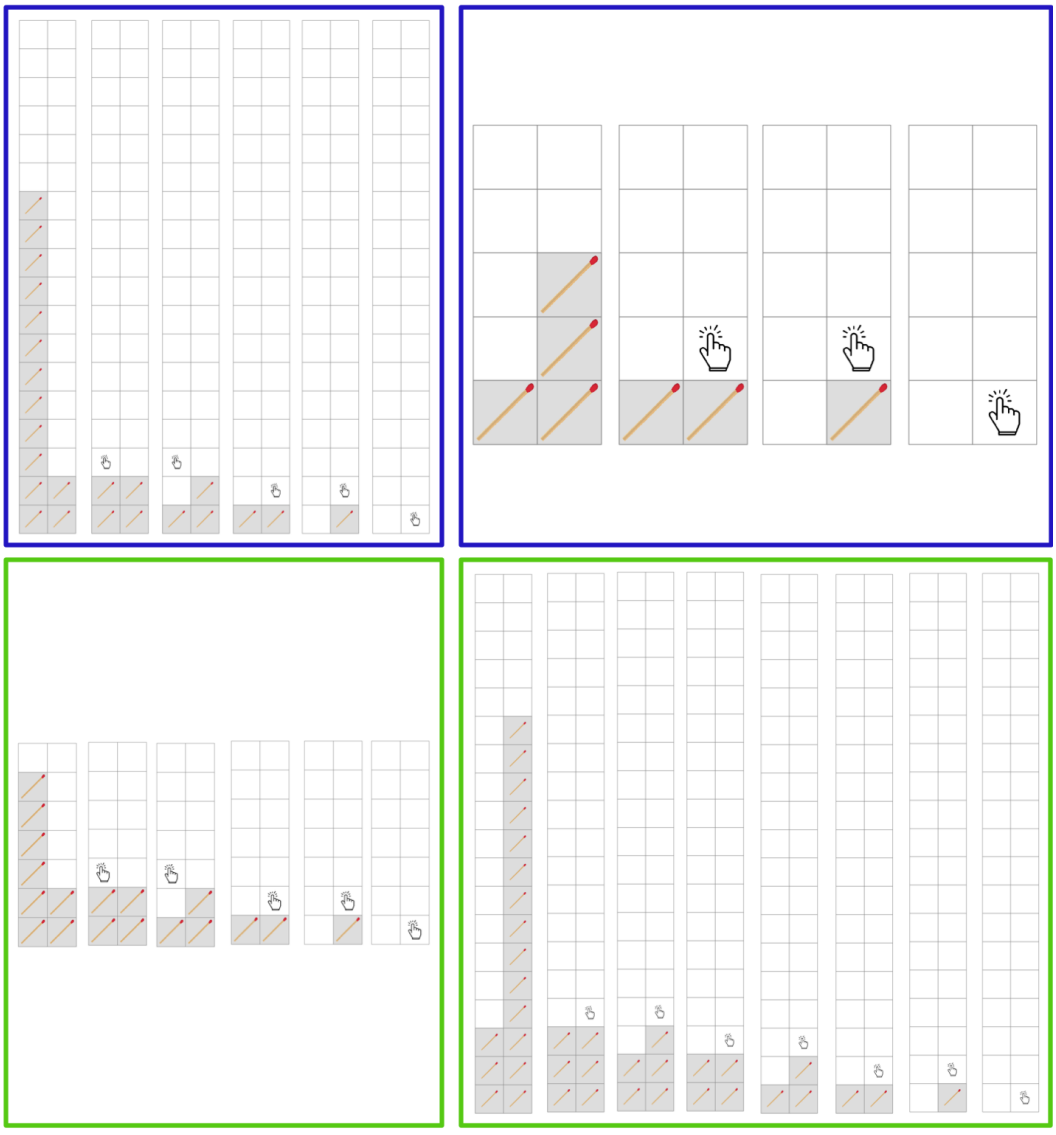

Figure 9: A PLP policy learned from two demonstrations of "Nim" (blue) generalizes perfectly to all test task instances (e.g. green).

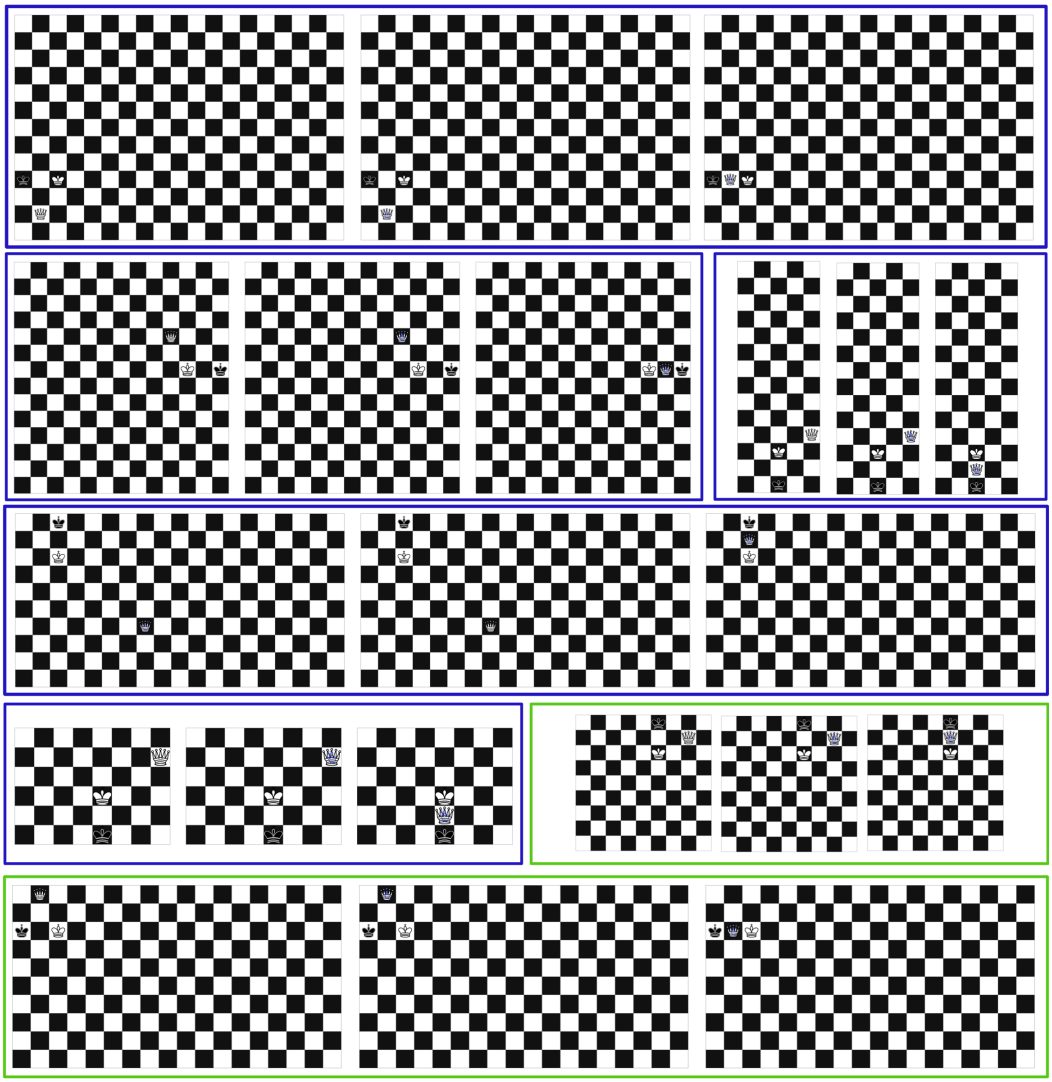

Figure 10: A PLP policy learned from five demonstrations of "Checkmate Tactic" (blue) generalizes perfectly to all test task instances (e.g. green).

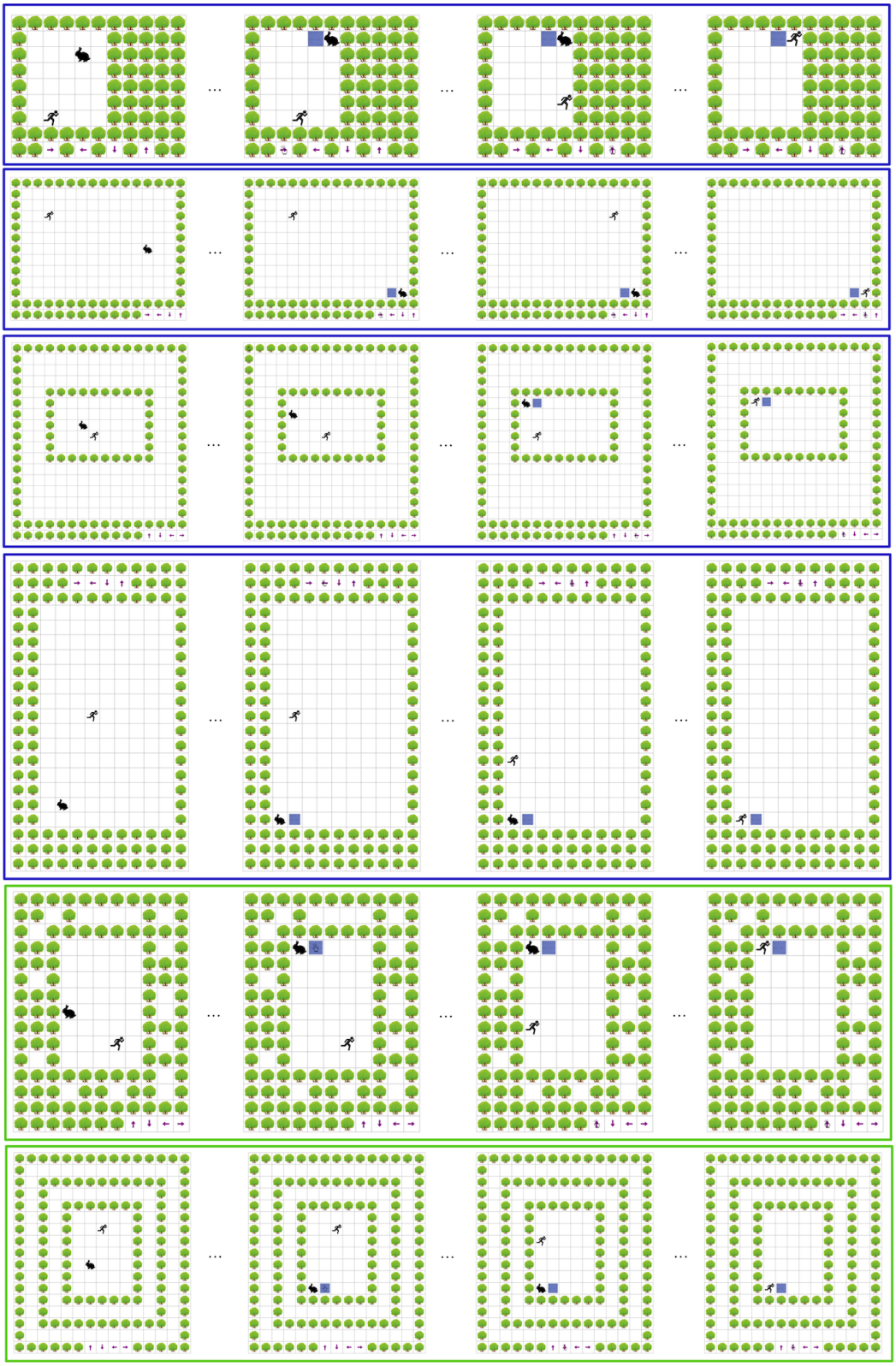

Figure 11: A PLP policy learned from four demonstrations of "Chase" (blue) generalizes perfectly to all test task instances (e.g. green).

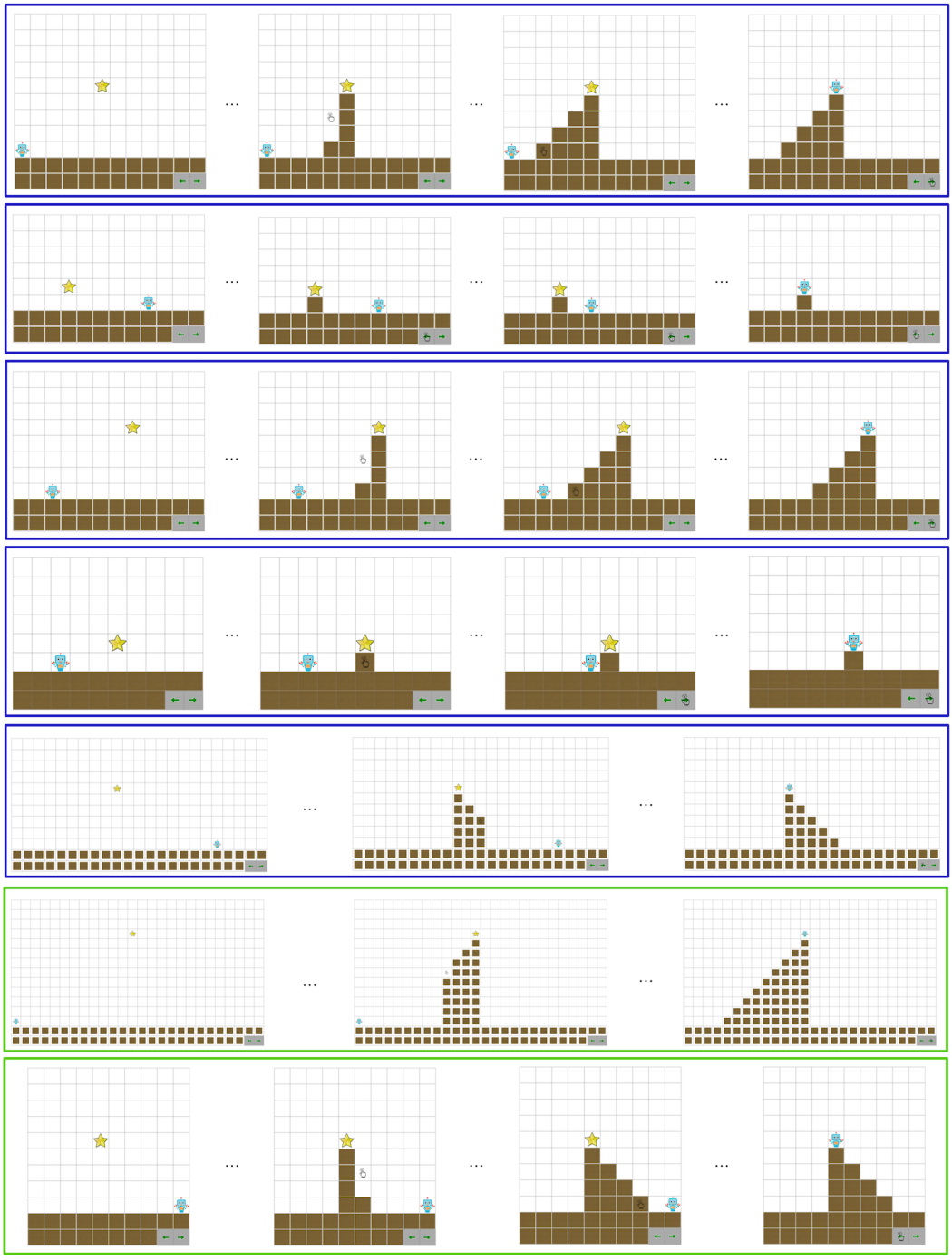

Figure 12: A PLP policy learned from five demonstrations of "Reach for the Star" (blue) generalizes perfectly to all test task instances (e.g. green).