Few-Shot Bayesian Imitation Learning with Logical Program Policies

Tom Silver, Kelsey R. Allen, Alex K. Lew, Leslie Kaelbling, Josh Tenenbaum
Massachusetts Institute of Technology
{tslvr, krallen, alexlew, lpk, jbt}@mit.edu

Abstract

Humans can learn many novel tasks from a very small number (1–5) of demonstrations, in stark contrast to the data requirements of nearly tabula rasa deep learning methods. We propose an expressive class of policies, a strong but general prior, and a learning algorithm that, together, can learn interesting policies from very few examples. We represent policies as logical combinations of programs drawn from a domain-specific language (DSL), 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 six 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. Our policy learning is 20–1,000x more data efficient than convolutional and fully convolutional policy learning and many orders of magnitude more computationally efficient than vanilla program induction. We argue that the proposed method is an apt choice for tasks that have scarce training data and feature significant, structured variation between task instances.

Introduction

People are remarkably good at learning and generalizing strategies for everyday tasks, like ironing a shirt or brewing a cup of coffee, from one or a few demonstrations. Websites like WikiHow.com and LifeHacker.com are filled with thousands of "how-to" guides for tasks that are hard to solve by pure reasoning or trial and error alone, but easy to learn and generalize from just one illustrated demo (Figure 1). We are interested in designing artificial agents with the same few-shot imitation learning capabilities.

A common approach to imitation learning is behavior cloning (BC), in which demonstrations are used as supervision to directly train a policy. BC is often thought to be too prone to overfitting to generalize from very little data. Indeed, we find that neural network policies trained with BC are suspectible to severe overfitting in our experiments. However, we argue that this failure is due not to BC in general, but rather, to an underconstrained policy class and a weak prior.

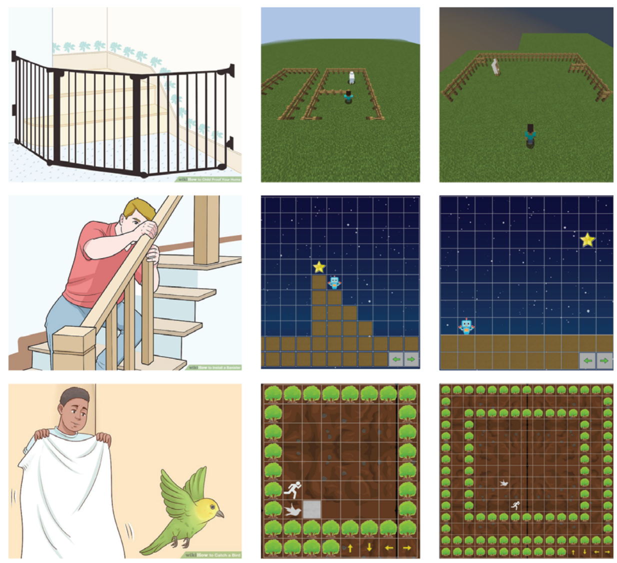


Figure 1: People can learn strategies for an enormous variety of tasks from one or a few demonstrations, e.g., "gate off an area," "build stairs," or "catch a bird" (left). We propose a policy class and learning algorithm for similarly data-efficient imitation learning. Given 1–5 demos of tasks like "Fence In," "Reach for the Star," and "Chase" (middle), we learn policies that generalize substantially (right).

More structured policies with strong Occam's razor priors can be found in two lines of work: logical and relational (policy) learning (Džeroski, De Raedt, and Blockeel 1998; Natarajan et al. 2011), and program (policy) synthesis (Wingate et al. 2013; Sun et al. 2018). Policies expressed in predicate logic are easy to learn, but difficult to scale, since each possible predicate must be hand-engineered by the researcher. Programmatic policies can be automatically generated by searching a small domain-specific language (DSL), but learning even moderately sophisticated policies can require an untenably large search in program space.

We propose Logical Program Policies (`LPP`): an expressive, structured, and efficiently learnable policy class that combines the strengths of logical and programmatic policies. Our first main idea is to consider policies that have logical "top level" structure and programmatic feature detectors (predicates) at the "bottom level." The feature detectors are expressions in a domain-specific language (DSL). By logically combining feature detectors, we can derive an infinitely large, rich policy class from a small DSL. This "infinite use of finite means" is in contrast to prior work in relational RL where each feature is individually engineered, making it labor-intensive to apply in complex settings.

Our second main idea is to exploit the logical structure of `LPP` to obtain an efficient imitation learning algorithm, overcoming the intractability of general program synthesis. To find policies in `LPP`, we incrementally enumerate feature detectors, apply them to the demonstrations, invoke an off-the-shelf Boolean learning method, and score each candidate policy with a likelihood and prior. What would be an intractable search over full policies is effectively reduced to a manageable search over feature detectors. We thus have an efficient approximate Bayesian inference method for $\mathsf{p}(\pi|\mathcal{D})$, the posterior distribution of policies π given demonstrations $\mathcal{D}$.

While `LPP` and the proposed learning method are agnostic to the particular choice of the DSL and application domain, we focus here on six strategy games which are played on 2D grids of arbitrary size (see Figure 3). In these games, a state consists of an assignment of discrete values to each grid cell and an action is a single grid cell (a "click" 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 six games from one shared, small DSL. In experiments, we find that policies learned from five or fewer demonstrations can generalize perfectly in all six games. 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, Shelhamer, and Darrell 2015)). Overall, our experiments suggest that `LPP` offers an efficient, flexible framework for learning rich, generalizable policies from very little data.

Problem Statement

In imitation learning, we are given a dataset $\mathcal{D}$ of expert trajectories $(s_0, a_0, ..., s_{T-1}, a_{T-1}, s_T)$ where $s_t \in \mathcal{S}$ are states and $a_t \in \mathcal{A}$ are actions. We suppose that the trajectories are sampled from a Markov process $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, \mathcal{G})$, with transition distribution $T(s' \mid s, a)$ and goal states $\mathcal{G} \subset \mathcal{S}$, and that actions are sampled from an expert policy $\pi^* : \mathcal{S} \times \mathcal{A} \to [0, 1]$, where $\pi^*(a \mid s)$ is a state-conditional distribution over actions. For imitation learning, we must specify (1) a hypothesis class of policies Π and (2) an algorithm for learning a policy $\pi \in \Pi$ from $\mathcal{D}$ that matches the expert π^*. We assume that the expert π^* is optimal with respect to $\mathcal{M}$, so we report the fraction of trials in which a learned policy π reaches goal states in $\mathcal{G}$ from held-out initial states in $\mathcal{M}$ to evaluate performance.

The `LPP` Policy Class

We seek a policy class Π with a concise parameterization that can be reasonably specified by a human programmer, and for which there is a tractable learning algorithm for recovering $\pi^* \in \Pi$ from demonstrations.

We consider policies that are parameterized by state-action classifiers $h : \mathcal{S} \times \mathcal{A} \to \{0, 1\}$. When $h(s, a) = 0$, action a will never be taken in state s; when $h(s, a) = 1$, a may be taken. This parameterization allows us to handle arbitrarily large action spaces (variable grid sizes). Given $h(s, a)$, we can derive a corresponding policy $\pi(a \mid s)$ that samples a uniformly at random among those a such that $h(s, a) = 1$. In other words, $\pi(a \mid s) \propto h(s, a)$. This stochastic policy formulation reflects the fact that the demonstrator may randomly select among several optimal actions. For completeness, we define $\pi(a \mid s) \propto 1$ if $\forall a, h(s, a) = 0$. Specifying a policy class Π thus reduces to specifying a class of functions $\mathcal{H}$ from which to learn an h.

One option for $\mathcal{H}$ is to consider *logical* rules that compute Boolean expressions combining binary features derived from (s, a). Although this enables fast inference using well-understood Boolean learning algorithms, it requires the AI programmer to hand-engineer informative binary features, which will necessarily vary from task to task. Another option is to consider *programmatic* rules: rules that are expressions in some general-purpose DSL for predicates on state-action pairs. In this case, the AI programmer need only specify a small core of primitives for the DSL, from which task-specific policies can be derived during inference. The challenge here is that finding a good policy in the infinitely large class of programs in the DSL is difficult; simple methods like enumeration are much too slow to be useful.

We combine the complementary strengths of logical and program-based policies to define the Logical Program Policies (`LPP`) class. Policies in `LPP` have a logical "top level" and a programmatic "bottom level." The bottom level is comprised of feature detector programs $f : \mathcal{S} \times \mathcal{A} \to \{0, 1\}$. These programs are expressions in a DSL and can include, for example, loops and conditional statements. A feature detector program takes a state s and an action a as input and returns a binary output, which provides one bit of information about whether a should be taken in s. The top level is comprised of a logical formula h over the outputs of the bottom level. Without loss of generality, we can express the formula in disjunctive normal form:

$$\begin{aligned} h(s, a) \triangleq (f_{1,1}(s, a) \wedge ... \wedge f_{1,n_1}(s, a)) \vee ... \\ \vee (f_{m,1}(s, a) \wedge ... \wedge f_{m,n_m}(s, a)) \end{aligned} \tag{1}$$

where the f's are possibly negated. `LPP` thus includes all policies that correspond to logical formulae over finite subsets of feature detector programs expressed in the DSL.

Imitation Learning as Bayesian Inference

We now address the imitation learning problem of finding a policy π that fits the expert demonstrations $\mathcal{D}$. Rather than finding a single LPP policy, we will infer a full posterior distribution over policies $\mathsf{p}(\pi \mid \mathcal{D})$. From a Bayesian perspective, maintaining the full posterior is principled; from a practical perspective, the full posterior leads to modest performance gains over a single MAP policy. Once we have inferred $\mathsf{p}(\pi \mid \mathcal{D})$, we will ultimately take MAP actions according to $\arg\max_{a \in \mathcal{A}} \mathbb{E}_{\mathsf{p}(\pi|\mathcal{D})}[\pi(a \mid s)]$.

Probabilistic Model $\mathsf{p}(\pi, \mathcal{D})$

We begin by specifying a probabilistic model over policies and demonstrations $\mathsf{p}(\pi, \mathcal{D})$, which factors into a prior distribution $\mathsf{p}(\pi)$ over policies in `LPP`, and a likelihood $\mathsf{p}(\mathcal{D} \mid \pi)$ giving the probability that an expert generates demonstrations $\mathcal{D}$ by following the policy π.

We choose the prior distribution $\mathsf{p}(\pi)$ to encode a preference for those policies which use fewer, simpler feature detector programs. Recall that a policy $\pi \in$ `LPP` is parameterized by a logical formula $h(s, a) = \bigvee_{i=1 \ldots M} \left(\bigwedge_{j=1 \ldots N_i} f_{i,j}(s, a)^{b_{ij}} (1 - f_{i,j}(s, a))^{1 - b_{ij}} \right)$, in which each of the $f_{i,j}$ is a binary feature detector expressed in a simple DSL and the b_{ij} are binary parameters that determine whether a given feature detector is negated. We set the prior probability of such a policy to depend only on the number and sizes of the programmatic components $f_{i,j}$: namely, $p(\pi) \propto \prod_{i=1}^{M} \prod_{j=1}^{N_i} \mathsf{p}(f_{i,j})$, the probability of generating each of the $f_{i,j}$ independently from a probabilistic context-free grammar $\mathsf{p}(f)$ (Manning and Schütze 1999).[1] The grammar we use in this work is shown in Table 1.

The likelihood of a dataset $\mathcal{D}$ given a policy π is $\mathsf{p}(\mathcal{D} \mid \pi) \propto \prod_{i=1}^{N} \prod_{j=1}^{T_i} \pi(a_{ij} \mid s_{ij})$.

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

Table 1: The prior $\mathsf{p}(f)$ over programs, specified as a probabilistic context-free grammar (PCFG).

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

Algorithm 1: `LPP` imitation learning

input: Demos $\mathcal{D}$, ensemble size K, max iters L
Create anti-demos $\overline{\mathcal{D}} = \{(s, a') : (s, a) \in \mathcal{D}, a' \neq a\}$;
Set labels $y[(s, a)] = 1$ if $(s, a) \in \mathcal{D}$ else 0;
Initialize approximate posterior q;
for i *in* $1, \ldots, L$ **do**
  f_i = generate_next_feature();
  $X = \{(f_1(s, a), \ldots, f_i(s, a))^T : (s, a) \in \mathcal{D} \cup \overline{\mathcal{D}}\}$
  μ_i, w_i = logical_inference(X, y, $\mathsf{p}(f)$, K);
  update_posterior(q, μ_i, w_i);
end
return q;

We now have a prior $\mathsf{p}(\pi)$ and likelihood $\mathsf{p}(\mathcal{D} \mid \pi)$, and we wish to compute an approximate posterior $\mathsf{q}(\pi) \approx \mathsf{p}(\pi \mid \mathcal{D})$. We take q to be a weighted mixture of K policies $\mu_1, \ldots, \mu_K$ (in our experiments, $K = 25$) and initialize it so that each μ_i is equally weighted and equal to the uniform policy, $\mu_i(a \mid s) \propto 1$. Our core insight is a way to exploit the structure of `LPP` to efficiently search the space of policies and update the mixture q to better match the posterior.

Our algorithm is given a set of demonstrations $\mathcal{D}$. The state-action pairs (s, a) in $\mathcal{D}$ comprise positive examples — inputs for which $h(s, a) = 1$. We start by computing a set of "anti-demonstrations" $\overline{\mathcal{D}} = \{(s, a') \mid (s, a) \in \mathcal{D}, a' \neq a\}$, which serve as approximate negative examples. ($\overline{\mathcal{D}}$ is approximate because it may contain false negatives, but they will generally constitute only a small fraction of the set.)

We now have a binary classification problem with positive examples $\mathcal{D}$ and negative examples $\overline{\mathcal{D}}$. The main loop of our algorithm considers progressively larger feature representations of these examples. At iteration i, we use only the simplest feature detectors f_1, ..., f_i, where "simplest" here means "of highest probability under the probabilistic grammar $\mathsf{p}(f)$." We can enumerate features in this order by performing a best-first search through the grammar.

Given a finite set of feature detectors $f_1, \ldots, f_i$, we can convert any state-action pair (s, a) into a length-i binary feature vector $\mathbf{x} \in \{0, 1\}^i = (f_1(s, a), \ldots, f_i(s, a))^T$. We do this conversion on $\mathcal{D}$ and $\overline{\mathcal{D}}$ to obtain a design matrix $X_i \in \{0, 1\}^{|\mathcal{D} \cup \overline{\mathcal{D}}| \times i}$. The remaining problem of learning a binary classifier as a logical combination of binary features is very well understood (Mitchell 1978; Quinlan 1986). In this work, we use an off-the-shelf stochastic greedy decision-tree learner (Pedregosa et al. 2011).

Given a learned decision tree, we can eas-

[1] 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.

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

Table 2: 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., P $= \mathcal{S} \times \mathcal{A} \rightarrow \{0, 1\}$), 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.

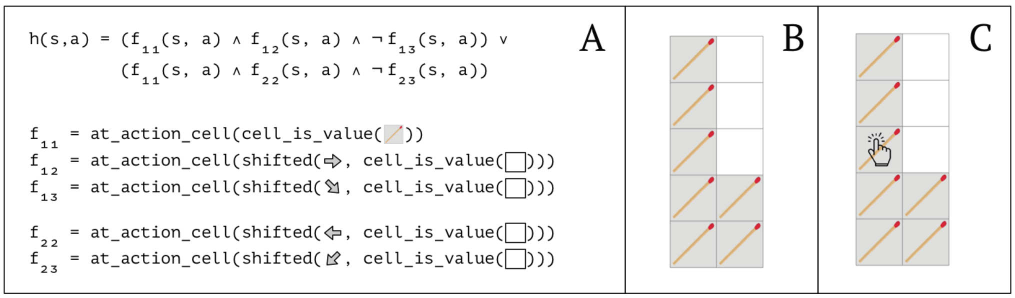


Figure 2: Example of a policy in `LPP` for the "Nim" game. (A) $h(s, a)$ is a logical combination of programs from a DSL. For example, f_{12} returns True if the cell to the right of the action a has value □. The induced policy is $\pi(a \mid s) \propto h(s, a)$. (B) Given state s, (C) there is one action selected by h. This policy encodes the "leveling" tactic, which wins the game.

ily read off a logical formula $h(s, a) = \bigvee_{j=1\ldots M} \left(\bigwedge_{l=1\ldots N_j} f_{j,l}(s, a)^{b_{jl}}(1 - f_{j,l}(s, a))^{1-b_{jl}}\right)$, in which each of the $f_{j,l}$ is one of the i feature detectors under consideration at iteration i. This induces a candidate policy $\mu_*(a|s) \propto h(s, a)$. We can evaluate its prior probability $\mathsf{p}(\mu_*)$ and its likelihood $\mathsf{p}(\mathcal{D} \mid \mu_*)$, then decide whether to include μ_* in our mixture q, based on whether its unnormalized posterior probability is greater than that of the lowest-scoring existing mixture component. The mixture is always weighted according to our model over π and $\mathcal{D}$, so that $\mathsf{q}(\mu_j) = \frac{\mathsf{p}(\mu_j|\mathcal{D})}{\sum_{i=1}^{K} \mathsf{p}(\mu_i|\mathcal{D})}$. In practice, we run the decision-tree learner several times (5 in experiments) with different random seeds to generate several distinct candidate policies at each iteration of the algorithm. We can stop the process after a fixed number of iterations, or when the prior probabilities of the enumerated programs f_i fall below a threshold: any policy that uses a feature detector f_i with prior probability $\mathsf{p}(f_i) < \mathsf{p}(\mu_j, \mathcal{D})$ for all μ_j in q's support has no chance of meriting inclusion in our mixture.

Once we have an approximation q to the posterior, we can use it to derive a final policy for use at test time:

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

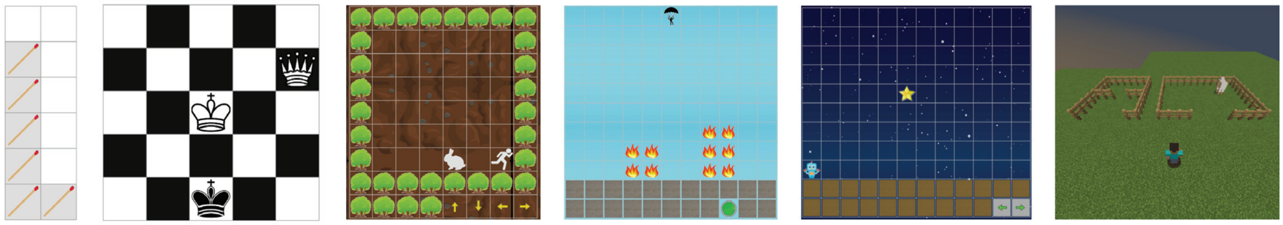
We could alternatively use the full distribution over actions to guide exploration, e.g., in combination with reinforcement learning (Hester et al. 2018). In this work, we focus on exploitation and therefore require only the maximum *a posteriori* actions, for use with a deterministic final policy.

Experiments and Results

We now present experiments to evaluate the data efficiency, computational complexity, and generalization of `LPP` versus several baselines. We also analyze the learned policies, examine qualitative performance, and conduct ablation studies to measure the contributions of the components of `LPP`. All experiments were performed on a single laptop running macOS Mojave with a 2.9 GHz Intel Core i9 processor and 32 GB of memory.

Tasks

We consider six diverse strategy games (Figure 3) that share a common state space ($\mathcal{S} = \bigcup_{h,w \in \mathbb{N}} V^{hw}$; variable-sized grids with discrete-valued cells) and action space $\mathcal{A} = \mathbb{N} \times \mathbb{N}$; single "clicks" on any cell). For a grid of dimension $h \times w$, we only consider clicks on the grid, i.e., $\{1, ..., h\} \times \{1, ..., w\}$. Grid sizes vary within tasks. These tasks feature high variability between different task instances; learning a robust policy requires substantial generalization. The tasks are also very challenging due to the unbounded action space, the absence of shaping or auxiliary rewards, and the arbitrarily long horizons that may be required to solve



Nim | Checkmate Tactic | Chase | Stop the Fall | Reach for the Star | Fence In

Figure 3: The strategy games studied in this work.

a task instance. Each task has a maximum episode length of 60 and counts as a failure if the episode terminates without a success. There are 11 training and 9 test instances per task. Instances of Nim, Checkmate Tactic, and Reach for the Star are procedurally generated; instances of Stop the Fall, Chase, and Fence In are manually generated, as the variation between instances is not trivially parameterizable.

Domain-Specific Language

Recall that each feature detector program takes a state and action as input and returns a Boolean value. In our tasks, states are full grid layouts and actions are single grid cells ("clicks"). The specific DSL of feature detectors that we use in this work (Table 2) is inspired by early work in visual routines (Ullman 1987; Hay et al. 2018). Each program implements a procedure for attending to some grid cell and checking that a local condition holds nearby. Given input (s, a), a program begins by initializing an implicit 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`). Next, the program will check a condition at or near the attended cell. The simplest condition is `cell_is_value`, which checks whether the attended cell has a certain 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 starts at the currently attended cell and "scans" along some direction, repeatedly shifting the attention pointer by a specified offset and checking whether either of two specified conditions hold. If, while scanning, the first condition becomes satisfied before the second, the `scanning` condition returns 1. Otherwise, it returns 0. Thus the overall DSL contains five methods, which are summarized in Table 2. See Figure 2 for a complete example of a policy in `LPP` using this DSL.

Baselines

Local Linear Network (LLN): A single 3×3 convolutional filter is trained to classify whether each cell in s should be "clicked," based only on the 8 surrounding cells. **FCN**: A deep fully convolutional network (Long, Shelhamer, and Darrell 2015) is trained with the same inputs and outputs as "Local Linear." The network has 8 convolutional layers with kernel size 3, stride 1, padding 1, 4 channels (8 in the input layer), and ReLU nonlinearities. This architecture was chosen to reflect the receptive field sizes we expect are necessary for the tasks. **CNN**: A standard convolutional neural network is trained with full grid inputs and discrete action outputs. Grids are padded so that all have the same maximal height and width. 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. **Vanilla Program Induction (VPI)**: Full policies are enumerated from a DSL grammar that includes logical disjunctions, conjunctions, and negations over the feature detector DSL. 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 `LPP` learning; this baseline is thus identical to `LPP` learning but with the greedy Boolean learning removed.

Effect of Number of Demonstrations

We first evaluate the test-time performance of `LPP` and baselines as the number of training demonstrations varies from 1 to 10. For each number of demonstrations, we run leave-one-out cross validation: 10 trials, each featuring a distinct set of demonstrations drawn from the overall pool of 11 training demonstrations. `LPP` 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. Results are shown in Figure 4. On the whole, `LPP` markedly outperforms all baselines, especially on the more difficult tasks. 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, Reach for the Star, and Fence In than the CNN, but overall its performance is still far behind `LPP`. In contrast, the LLN baseline is unable to fit the training data; with the exception of Nim, its training performance is close to zero. Similarly, the training perfor-

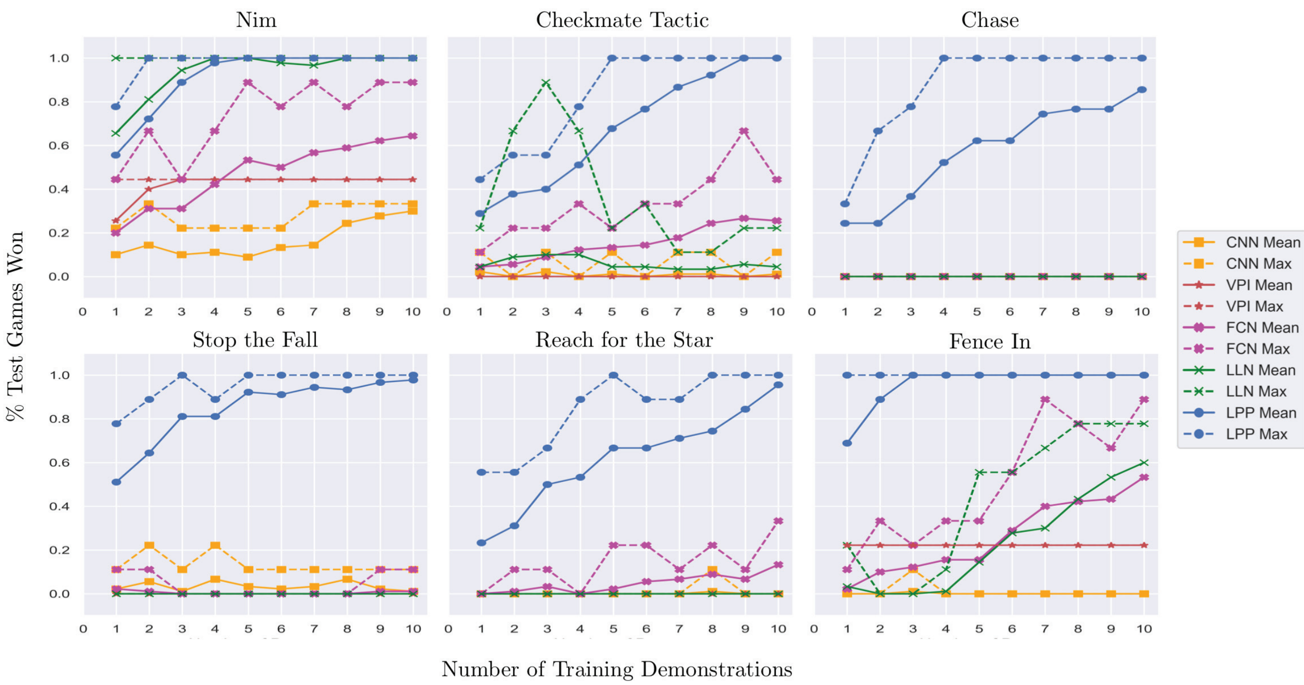


Figure 4: Performance on held-out test task instances as a function of the number of training demonstrations for `LPP` (ours) and four baselines. Maximums and means are over 10 training sets.

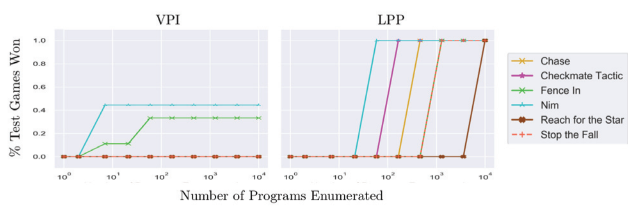


Figure 5: Performance on held-out test task instances as a function of the number of programs enumerated for the Vanilla Program Induction (VPI) baseline and `LPP` (ours).

mance of the VPI 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 find.

Effect of Number of Programs Searched

We now examine test-time performance of `LPP` and VPI as a function of the number of programs searched. For this experiment, we give both methods all 11 training demonstrations for each task. Results are shown in Figure 5. `LPP` 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, Reach for the Star, and Fence In. In contrast, VPI 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 VPI is unsurprising given the

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

Figure 6: Performance on held-out test task instances for `LPP` and four ablation models.

combinatorial explosion of programs. For example, the optimal policy for Nim shown in Figure 2 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 six or fewer constituent programs. VPI 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 `LPP` policy is learnable after fewer than 100 enumerations. In practical terms, `LPP` learning for Nim takes on the order of 1 second on a laptop without highly optimized code; after running VPI for six hours in the same setup, a winning policy is still not found.

Ablation Studies

We now perform ablation studies to explore which aspects of the `LPP` 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 DSL. This model addresses the possibility that the features alone are powerful enough to solve the task when combined with a simple classifier. The NN 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 LPP learning. The "No Prior" model is identical to LPP learning, except that the grammatical prior is replaced with a uniform prior. Similarly, the "Sparsity Prior" model uses 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 6. They confirm that the each component — the feature detectors, the sparsity regularization, and the grammatical prior — adds value to the overall framework.

Related Work

Sample efficiency and generalization are two of the main concerns in imitation learning (Schaal 1997; Abbeel and Ng 2004). To cope with limited data, demonstrations can be used in combination with additional RL (Hester et al. 2018; Nair et al. 2018). 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 policy class (Andre and Russell 2002; Doshi-Velez et al. 2010; Wingate et al. 2011), e.g., hierarchical or compositional policies (Niekum 2013). Our work fits into the third tradition; our contribution is a new policy class with a structured prior that enables efficient learning.

LPP policies are logical at the "top level" and programmatic at the "bottom level." Logical representations for RL problems have been considered in many previous works, particularly in *relational RL* (Džeroski, De Raedt, and Blockeel 1998; Natarajan et al. 2011). Also notable is work by Shah et al. (2018), who learn linear temporal logic specifications from demonstration using a finite grammatical prior. While some LPP programs may be seen as fixed-arity logical relations in the classical sense, others are importantly more general and powerful, involving loops and a potentially arbitrary number of atoms. For example, one program suffices to check whether a Queen's diagonal path is clear in Chess; no relation over a fixed number of squares can capture the same feature. Furthermore, relational RL assumes that relations are fixed, finite, and given, typically hand-designed by the programmer. In LPP, the programmer instead supplies a DSL describing infinitely many features.

LPP learning is a particular type of *program synthesis*, which more broadly refers to a search over programs, including but not limited to the case where we have a grammar over programs, the programs are mappings, and input-output examples are available. When a grammar is given, the problem is sometimes called *syntax-guided synthesis* (Alur et al. 2013). Most relevant is work by Alur, Radhakrishna, and Udupa (2017), who propose a "divide and conquer" approach that uses greedy decision tree learning in combination with enumeration from a grammar of "conditions", similar to our "No Prior" baseline.

Recent work has also examined *neural program synthesis* (NPS) wherein a large dataset of (input, output, program) examples is used to train a guidance function for program enumeration (Parisotto et al. 2016; Bunel et al. 2018; Huang et al. 2019). In practice, NPS methods are still limited to programs involving $\sim$10 primitives. The neural guidance can delay, but not completely avoid, the combinatorial explosion of search in program space. LPP learning is not a generic program induction method, but rather, an algorithm that exploits the logical structure of LPP programs to dramatically speed up search, sometimes finding programs with $\sim$250 primitives.

The interpretation of policy learning as an instance of program synthesis is explored in prior work (Wingate et al. 2011; Sun et al. 2018; Verma et al. 2018). In particular, Lázaro-Gredilla et al. (2019) learn object manipulation concepts from before/after image pairs that can be transferred between 2D simulation and a real robot. In this work, we focus on the problem of *efficient inference* and compare against vanilla program induction in experiments.

Discussion and Conclusion

In an effort to efficiently learn policies from very few demonstrations that generalize substantially, we have introduced the LPP policy class and an approximate Bayesian inference algorithm for imitation learning. We have seen that the LPP 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. In ongoing work we are studying how to scale our approach to a wider range of tasks, starting with more sophisticated DSLs that include counting or simple data structures. However, even our current DSL is surprisingly general. For instance, in preliminary experiments, we find that our current algorithm can learn a generalizing policy for Atari Breakout from just one demonstration.

Beyond policy learning, this work contributes to the long and ongoing discussion about the role of prior knowledge in AI. In the common historical narrative, early attempts to incorporate prior knowledge via 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 inductive bias into contemporary methods, especially for problems where data is scarce. 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.

Acknowledgements

We 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; and from the Center for Brains, Minds and Machines (CBMM), funded by NSF STC award CCF-1231216. KA acknowledges support from NSERC.

References

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

Alur, R.; Bodik, R.; Juniwal, G.; Martin, M. M.; Raghothaman, M.; Seshia, S. A.; Singh, R.; Solar-Lezama, A.; Torlak, E.; and Udupa, A. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design*.

Alur, R.; Radhakrishna, A.; and Udupa, A. 2017. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 319–336. Springer.

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

Bunel, R.; Hausknecht, M.; Devlin, J.; Singh, R.; and Kohli, P. 2018. Leveraging grammar and reinforcement learning for neural program synthesis. *International Conference on Learning Representations*.

Doshi-Velez, F.; Wingate, D.; Roy, N.; and Tenenbaum, J. B. 2010. Nonparametric Bayesian policy priors for reinforcement learning. *Advances in Neural Information Processing Systems*.

Duan, Y.; Andrychowicz, M.; Stadie, B.; Ho, J.; Schneider, J.; Sutskever, I.; Abbeel, P.; and Zaremba, W. 2017. One-shot imitation learning. *Advances in Neural Information Processing Systems*.

Džeroski, S.; De Raedt, L.; and Blockeel, H. 1998. Relational reinforcement learning. In *International Conference on Inductive Logic Programming*, 11–22. Springer.

Finn, C.; Yu, T.; Zhang, T.; Abbeel, P.; and Levine, S. 2017. One-shot visual imitation learning via meta-learning. *Conference on Robot Learning*.

Hay, N.; Stark, M.; Schlegel, A.; Wendelken, C.; Park, D.; Purdy, E.; Silver, T.; Phoenix, D. S.; and George, D. 2018. Behavior is everything–towards representing concepts with sensorimotor contingencies. *AAAI Conference on Artificial Intelligence*.

Hester, T.; Vecerik, M.; Pietquin, O.; Lanctot, M.; Schaul, T.; Piot, B.; Horgan, D.; Quan, J.; Sendonaris, A.; Osband, I.; et al. 2018. Deep Q-learning from demonstrations. *AAAI Conference on Artificial Intelligence*.

Huang, D.-A.; Nair, S.; Xu, D.; Zhu, Y.; Garg, A.; Fei-Fei, L.; Savarese, S.; and Niebles, J. C. 2019. Neural task graphs: Generalizing to unseen tasks from a single video demonstration. In *Computer Vision and Pattern Recognition*.

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

Long, J.; Shelhamer, E.; and Darrell, T. 2015. Fully convolutional networks for semantic segmentation. *IEEE Conference on Computer Vision and Pattern Recognition*.

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

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

Nair, A.; McGrew, B.; Andrychowicz, M.; Zaremba, W.; and Abbeel, P. 2018. Overcoming exploration in reinforcement learning with demonstrations. *International Conference on Robotics and Automation*.

Natarajan, S.; Joshi, S.; Tadepalli, P.; Kersting, K.; and Shavlik, J. 2011. Imitation learning in relational domains: A functional-gradient boosting approach. In *Twenty-Second International Joint Conference on Artificial Intelligence*.

Niekum, S. D. 2013. *Semantically grounded learning from unstructured demonstrations*. Ph.D. Dissertation, University of Massachusetts, Amherst.

Parisotto, E.; Mohamed, A.-r.; Singh, R.; Li, L.; Zhou, D.; and Kohli, P. 2016. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*.

Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; and Duchesnay, E. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12:2825–2830.

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

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

Shah, A.; Kamath, P.; Shah, J. A.; and Li, S. 2018. Bayesian inference of temporal task specifications from demonstrations. In *Advances in Neural Information Processing Systems*, 3804–3813.

Sun, S.-H.; Noh, H.; Somasundaram, S.; and Lim, J. 2018. Neural program synthesis from diverse demonstration videos. In *International Conference on Machine Learning*.

Sutton, R. 2019. The bitter lesson.

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

Verma, A.; Murali, V.; Singh, R.; Kohli, P.; and Chaudhuri, S. 2018. Programmatically interpretable reinforcement learning. *International Conference on Machine Learning*.

Wingate, D.; Goodman, N. D.; Roy, D. M.; Kaelbling, L. P.; and Tenenbaum, J. B. 2011. Bayesian policy search with policy priors. *International Joint Conference on Artificial Intelligence*.

Wingate, D.; Diuk, C.; O'Donnell, T.; Tenenbaum, J.; and Gershman, S. 2013. Compositional policy priors. Technical report, Massachusetts Institute of Technology.