SYNTHESIZING PROGRAMMATIC POLICIES THAT INDUCTIVELY GENERALIZE

Anonymous authors
Paper under double-blind review

ABSTRACT

Deep reinforcement learning has successfully solved a number of challenging control tasks. However, learned policies typically have difficulty generalizing to novel environments. We propose an algorithm for learning *programmatic state machine policies* that can capture repeating behaviors. By doing so, they have the ability to generalize to instances requiring an arbitrary number of repetitions, a property we call *inductive generalization*. However, state machine policies are hard to learn since they consist of a combination of continuous and discrete structure. We propose a learning framework called *adaptive teaching*, which learns a state machine policy by imitating a teacher; in contrast to traditional imitation learning, our teacher adaptively updates itself based on the structure of the student. We show how our algorithm can be used to learn policies that inductively generalize to novel environments, whereas traditional neural network policies fail to do so.

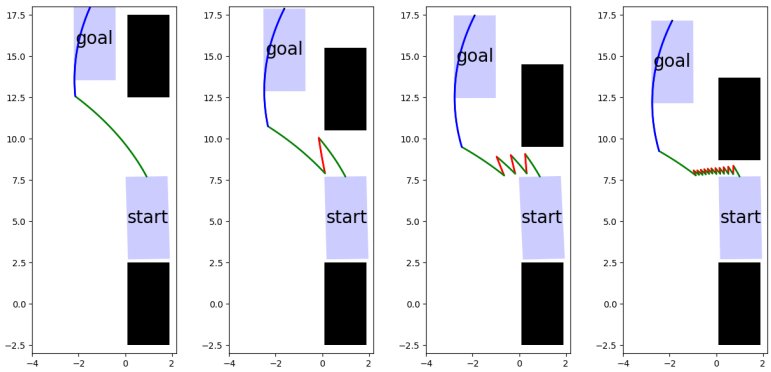
1 INTRODUCTION

Existing deep reinforcement learning (RL) approaches have difficulty generalizing to novel environments (Packer et al., 2018). More specifically, consider a task that requires performing a repeating behavior—we would like to be able to learn a policy that generalizes to instances requiring an arbitrary number of repetitions. We refer to this property as *inductive generalization*. In supervised learning, specialized neural network architectures have been proposed that exhibit inductive generalization on tasks such as list manipulation (Cai et al., 2017), but it is not obvious how those techniques would generalize to the control problems discussed in this paper. Alternatively, algorithms have been proposed for learning programmatic policies that generalize well (Verma et al., 2019), but existing approaches have focused on simple stateless policies that cannot represent repeating behaviors.

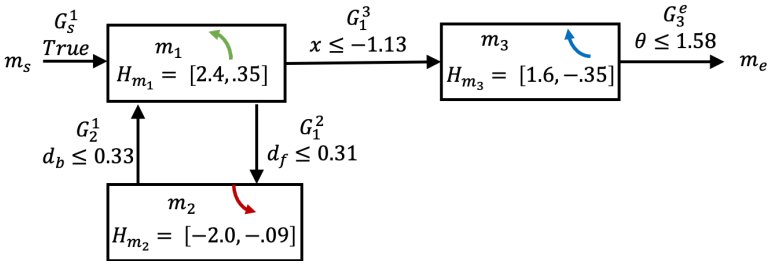
We propose an algorithm for learning *programmatic state machine policies*. Such a policy consists of a set of internal states, called *modes*, each of which is associated with a controller that is applied while in that mode. The policy also includes transition predicates that describe how the mode is updated. These policies are sufficiently expressive to capture tasks of interest—e.g., they can perform repeating tasks by cycling through some subset of modes during execution. Additionally, state machine policies are strongly biased towards policies that inductively generalize, that deep RL policies lack. In other words, this policy class is both *realizable* (i.e., it contains a "right" policy that solves the problem for all environments) and *identifiable* (i.e., we can learn the right policy from limited data).

However, state machine policies are challenging to learn because their discrete state transitions make it difficult to use gradient-based optimization. We use a standard solution where we "soften" the state transitions by making them probabilistic. However, these techniques alone are insufficient; they still run into local optima due to the the constraints on the structure of the policy function, as well as the relatively few parameters they possess.

To address this issue, we propose an approach called *adaptive teaching*, where we alternatingly learn a *teacher*, which is an overparameterized version of a *student*, which is a state machine policy trained, in turn, to mimic the teacher. Because the teacher is overparameterized, it can more easily accomplish the task compared to the student (but does not generalize as well as the student). Furthermore, the teacher is regularized to favor strategies similar to the ones taken by the student, to ensure the student can successfully mimic the teacher. As the student improves, the teacher improves as well. This alternating optimization can naturally be derived within the framework of variational inference, where the teacher encodes the variational distribution (Wainwright et al., 2008).



(a) d = 15 (train) (b) d = 13 (train) (c) d = 12 (train) (d) d = 11.2 (test)



(e) State machine based policy. Edges that have trivially false switching conditions are dropped.

Figure 1: Running example: retrieving an autonomous car from tight parking spots.

We implement our algorithm and evaluate it on a set of reinforcement learning problems focused on tasks that require inductive generalization. We show that traditional deep RL approaches perform well on the original task, but fail to generalize inductively, whereas our state machine policies successfully generalize beyond the training distribution.

Example. Consider the autonomous car in Figure 1, which consists of a blue car (the agent) parked between two stationary black cars. The system state is (x, y, θ, d), where (x, y) is the center of the car, θ is the orientation, and d is the distance between the two black cars. The actions are (v, ψ), where v is velocity and ψ is steering angle (we consider velocity control since the speed is low). The dynamics are standard bicycle dynamics. The goal is to drive out of the parked spot to an adjacent lane while avoiding collisions. This task is easy when d is large (Figure 1a). It is somewhat more involved when d is small, since it requires multiple maneuvers (Figures 1b and 1c). However, it becomes challenging when d is very small (Figure 1d). A standard RL algorithm will train a policy that performs well on the distances seen during training but does not generalize to smaller distances. In contrast, our goal is to train an agent on scenarios (a), (b), and (c), that generalizes to scenario (d).

In Figure 1e, we show a state machine policy synthesized by our algorithm for this task. We use d_f and d_b to denote the distances between the agent and the front and back black cars, respectively. This policy has three different modes (besides a start mode m_s and an end mode m_e). Roughly speaking, this policy says (i) immediately shift from mode m_s to m_1, and drive the car forward and to the left, (ii) continue until close to the car in front; then, transition to mode m_2, and drive the car backwards and to the right, (iii) continue until close to the car behind; then, transition back to mode m_1, (iv) iterate between m_1 and m_2 until the car can safely exit the parking spot; then, transition to mode m_3, and drive forward and to the right to make the car parallel to the lane. This policy inductively generalizes since it captures the iterative behavior of driving forward and then backward until exiting the parking spot. Thus, it successfully solves the scenario in Figure 1d.

Related work. State machines have been used represent policies that have internal state (typically called *memory*). To learn these policies, gradient ascent methods assume a fixed structure and optimize over real-valued parameters (Meuleau et al., 1999; Peshkin et al., 2001; Aberdeen & Baxter, 2002), whereas policy iteration methods uses dynamic programming to extend the structure (Hansen,

1998). Our method combines both, but similarly to Poupart & Boutilier (2004), the structure space is bounded. In addition, programmatic state machines use programs to represent state transitions and actions rules, and as a result can perform well while remaining small in size. Hierarchies of Abstract Machines (HAM)s also use programmatic state machines for hierarchical reinforcement learning, but assumed a fixed, hand-designed structure (Parr & Russell, 1998; Andre & Russell, 2002).

Next, there has been growing interest in using program synthesis to aid machine learning (Lake et al., 2015; Ellis et al., 2015; 2018; Valkov et al., 2018; Young et al., 2019). Our work is most closely related to recent work using imitation learning to learn programmatic policies (Verma et al., 2018; Bastani et al., 2018; Zhu et al., 2019; Verma et al., 2019). However, these approaches all assume a domain-specific program synthesizer that can learn programmatic policies given a supervised dataset. Building such a synthesizer for state machine policies is challenging since they contain both discrete and continuous parameters and internal state. Thus, our student does not learn based on examples provided by the teacher, but is trained to mimic the internal structure of the teacher.

Our inductive generalization goal is related to that of meta-learning (Finn et al., 2017); however, whereas meta-learning trains on a few examples from the novel environment, our goal is to generalize without additional training. Our work is also related to guided policy search, which uses a teacher in the form of a trajectory optimizer to train a neural network student (Levine & Koltun, 2013). However, training programmitic policies is more challenging since the teacher must mirror the structure of the student. Finally, it has recently been shown that overparameterization is essential in helping neural networks avoid local minima (Allen-Zhu et al., 2019). Relaxing optimization problems by adding more parameters is a well established technique; in many cases, re-parameterization can make difficult non-convex problems solve efficiently (Carlone & Calafiore, 2018).

2 Problem Formulation

Dynamics. We are interested in synthesizing control policies for deterministic, continuous-time dynamical systems with continuous state and action spaces. In particular, we consider partially observable Markov decision processes (POMDP) $\langle \mathcal{X}, \mathcal{A}, \mathcal{O}, F, Z, X_0, \phi_S, \phi_G \rangle$ with states $\mathcal{X} \subseteq \mathbb{R}^{d_X}$, actions $\mathcal{A} \subseteq \mathbb{R}^{d_A}$, observations $\mathcal{O} \subseteq \mathbb{R}^{d_O}$, deterministic dynamics $F : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$ (i.e., $\dot{\mathbf{x}} = F(\mathbf{x}, \mathbf{a})$), deterministic observation function $Z : \mathcal{X} \to \mathcal{O}$, and initial state distribution $\mathbf{x}_0 \sim X_0$.

We consider a safety specification $\phi_S : \mathcal{X} \to \mathbb{R}$ and a goal specification $\phi_G : \mathcal{X} \to \mathbb{R}$. Then, the agent aims to reach a goal state $\phi_G(\mathbf{x}) \leq 0$ while staying in safe states $\phi_S(\mathbf{x}) \leq 0$. A positive value for $\phi_S(\mathbf{x})$ (resp., $\phi_G(\mathbf{x})$) quantifies the degree to which $\mathbf{x}$ is unsafe (resp., away from the goal).

Policies. We consider policies $\pi : \mathcal{O} \times \mathcal{S} \to \mathcal{A} \times \mathcal{S}$ that keep internal memory; we assume the memory is initialized to a constant $\mathbf{s}_0$. Given such a policy π, we sample a rollout (or trajectory) $\tau = (\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_N)$ with horizon $N \in \mathbb{N}$ by sampling $\mathbf{x}_0 \sim X_0$ and then performing a discrete-time simulation $\mathbf{x}_{n+1} = \mathbf{x}_n + F(\mathbf{x}_n, \mathbf{a}_n) \cdot \Delta$, where $(\mathbf{a}_n, \mathbf{s}_{n+1}) = \pi(Z(\mathbf{x}_n), \mathbf{s}_n)$ and $\Delta \in \mathbb{R}_{>0}$ is the time increment. Since F, Z, and π are deterministic, τ is fully determined by $\mathbf{x}_0$ and π; τ can also be represented as a list of actions combined with the initial state i.e $\tau = \langle \mathbf{x}_0, (\mathbf{a}_0, \mathbf{a}_1, \cdots, \mathbf{a}_N) \rangle$.

The degree to which ϕ_S and ϕ_G are satisfied along a trajectory is quantified by a reward function $R(\pi, \mathbf{x}_0) = -\phi_G(\mathbf{x}_N)^+ - \sum_{n=0}^{N} \phi_S(\mathbf{x}_n)^+$, where $x^+ = \max(0, x)$. The optimal policy π^* in some class Π is one which maximizes the expected reward $\mathbb{E}_{\mathbf{x}_0 \sim X_0}[R(\pi, \mathbf{x}_0)]$.

Inductive generalization. Beyond optimizing reward, we want a policy that *inductively generalizes* to unseen environments. Formally, we actually consider two initial state distributions: a training distribution X_0^{train}, and a test distribution X_0^{test} that includes the extreme states never encountered during training. Then, the goal is to train a policy according to X_0^{train}—i.e.,

$$\pi^* = \arg\max_{\pi \in \Pi} \ \mathbb{E}_{\mathbf{x}_0 \sim X_0^{\text{train}}}[R(\pi, \mathbf{x}_0)], \tag{1}$$

but measure its performance according X_0^{test}—i.e., $\mathbb{E}_{\mathbf{x}_0 \sim X_0^{\text{test}}}[R(\pi, \mathbf{x}_0)]$.

3 Programmatic State Machine Policies

To achieve inductive generalization, we aim to synthesize programmatic policies in the form of state machines. At a high level, state machines can be thought of as compositions of much simpler policies,

where the internal state of the state machines (called its mode) indicates which simple policy is currently being used. Thus, state machines are capable of encoding complex nonlinear control tasks such as iteratively repeating a complex sequence of actions (e.g., the car example in Figure 1). At the same time, state machines are substantially more structured than more typical policy classes such as neural networks and decision trees.

More precisely, a state machine π is a tuple $\langle \mathcal{M}, \mathcal{H}, \mathcal{G}, m_s, m_e \rangle$. The *modes* $m_i \in \mathcal{M}$ of π are the internal memory of the state machine. Each mode $m_i \in \mathcal{M}$ corresponds to an *action function* $H_{m_i} \in \mathcal{H}$, which is a function $H_{m_i} : \mathcal{O} \to \mathcal{A}$ mapping observations to actions. When in mode m_i, the agent takes action $\mathbf{a} = H_{m_i}(\mathbf{o})$. Furthermore, each pair of modes (m_i, m_j) corresponds to a *switching condition* $G_{m_i}^{m_j} \in \mathcal{G}$, which is a function $G_{m_i}^{m_j} : \mathcal{O} \to \mathbb{R}$. When an agent in mode m_i observes $\mathbf{o}$ such that $G_{m_i}^{m_j}(\mathbf{o}) \geq 0$, then the agent transitions from mode m_i to mode m_j. If there are multiple modes m_j with non-negative switching weight $G_{m_i}^{m_j}(\mathbf{o}) \geq 0$, then the agent transitions to the one that is greatest in magnitude; if there are several modes of equal weight, we take the first one according to a fixed ordering. Finally, $m_s, m_e \in \mathcal{M}$ are the start and end modes, respectively; the state machine mode is initialized to m_s, and the state machine terminates when it transitions to m_e.

Formally, $\pi(\mathbf{o}_n, \mathbf{s}_n) = (\mathbf{a}_n, \mathbf{s}_{n+1})$, where $\mathbf{a}_n = H_{\mathbf{s}_n}(\mathbf{o}_n)$, $\mathbf{s}_0 = m_s$ and

$$\mathbf{s}_{n+1} = \begin{cases} m = \mathrm{darg\,max}_m \, G_{\mathbf{s}_n}^m(\mathbf{o}_n) & \text{if } G_{\mathbf{s}_n}^m(\mathbf{o}_n) \geq 0 \\ \mathbf{s}_n & \text{otherwise.} \end{cases} \tag{2}$$

where $\mathrm{darg\,max}$ is a deterministic $\arg\max$ that breaks ties as described above.

Action functions and switching conditions are specified by *grammars* that encode the space of possible functions as a space of programs. Different grammars can be used for different problems. Typical grammars for action functions include constants $\{C_\alpha : \mathbf{o} \mapsto \alpha\}$ and proportional controls $\{P_{\alpha_0,\alpha_1}^i : \mathbf{o} \mapsto \alpha_0(\mathbf{o}[i] - \alpha_1)\}$. A typical grammar for switching conditions is the grammar

$$B \ ::= \ \mathbf{o}[i] \leq \alpha_0 \mid \mathbf{o}[i] \geq \alpha_0 \mid B_1 \wedge B_2 \mid B_1 \vee B_2$$

of Boolean predicates over the current observation $\mathbf{o}$, where $\mathbf{o}[i]$ is the ith component of $\mathbf{o}$. In all these grammars, $\alpha_i \in \mathbb{R}$ are parameters to be learned. The grammar for switching conditions also has discrete parameters encoding the choice of expression. For example, in Figure 1, the action functions are constants, and the switching conditions are inequalities over components of $\mathbf{o}$.

4 Framework for Synthesizing Programmatic Policies

We now describe our adaptive teaching framework for synthesizing state machine policies. In this section, the teacher is abstractly represented as a collection of trajectories $\tau_{\mathbf{x}_0}$ (i.e., an open-loop controller consisting of a fixed sequence of actions) for each initial state $\mathbf{x}_0$. A key insight is that we can parameterize $\tau_{\mathbf{x}_0}$ in a way that mirrors the structure of the state machine student. As we discuss in Section 4.2, we parameterize $\tau_{\mathbf{x}_0}$ as a "loop-free" state machine. Intuitively, our algorithm efficiently computes $\tau_{\mathbf{x}_0}$ (from multiple initial states $\mathbf{x}_0$) using gradient-based optimization, and then "glues" them together using maximum likelihood to construct a state machine policy.

4.1 Adaptive Teaching via Variational Inference

We derive the adaptive teaching formulation by reformulating the learning problem in the framework of probabilistic reinforcement learning, and also consider policies π that are probabilistic state machines (see Section 4.3). Then, we use a variational approach to break the problem into the teacher and the student steps. In this approach, the log-likelihood of a policy π is defined as follows:

$$\ell(\pi) = \log \mathbb{E}_{p(\tau|\pi)}[e^{\lambda R(\tau)}] \tag{3}$$

where $p(\tau \mid \pi)$ is the probability of sampling rollout τ when using policy π from a random initial state $\mathbf{x}_0$, $\lambda \in \mathbb{R}_{\geq 0}$ is a hyperparameter, and $R(\tau)$ is the reward assigned to τ. We have

$$\ell(\pi) = \log \mathbb{E}_{q(\tau)} \left[e^{\lambda R(\tau)} \cdot \frac{p(\tau \mid \pi)}{q(\tau)} \right] \geq \mathbb{E}_{q(\tau)}[\lambda R(\tau) + \log p(\tau|\pi) - \log q(\tau)] \tag{4}$$

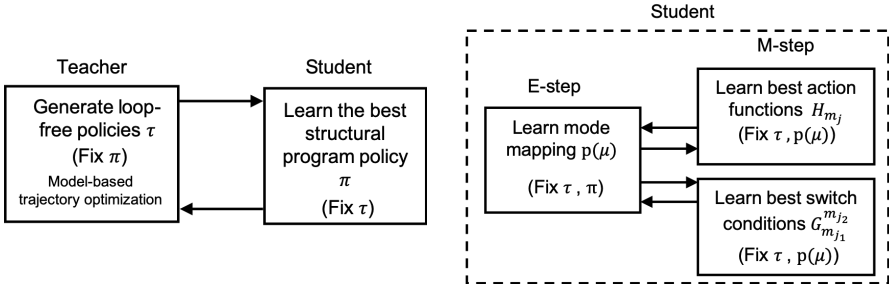


Figure 2: Flowchart connecting the different components of the algorithm.

where $q(\tau)$ is the variational distribution and the inequality follows from Jensen's inequality. Thus, we can optimize π by maximizing the lower bound Eq (4) on $\ell(\pi)$. Since the first and third term of Eq (4) are constant with respect to π, we have

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{q(\tau)}[\log p(\tau|\pi)]. \tag{5}$$

Next, the optimal choice for q (i.e., to minimize the gap in the inequality in Eq (4)) is

$$q^* = \arg\min_{q} D_{\mathrm{KL}}(q(\tau) \parallel e^{\lambda R(\tau)} \cdot p(\tau \mid \pi)/Z) \tag{6}$$

where Z is a normalizing constant. We choose q to have the form $q(\tau) = p(\mathbf{x}_0) \cdot \delta(\tau - \tau_{\mathbf{x}_0})$ where δ is the Dirac delta function, $p(\mathbf{x}_0)$ is the initial state distribution, and $\tau_{\mathbf{x}_0}$ are the parameters to be optimized, where $\tau_{\mathbf{x}_0}$ encodes a trajectory from $\mathbf{x}_0$. Then, up to constants, the objective of Eq (6) equals

$$\mathbb{E}_{p(\mathbf{x}_0)}\left[\log p(\mathbf{x}_0) + \mathbb{E}_{\delta(\tau-\tau_{\mathbf{x}_0})}[\log \delta(\tau - \tau_{\mathbf{x}_0})] - (\lambda R(\tau_{\mathbf{x}_0}) + \log p(\tau_{\mathbf{x}_0} \mid \pi, \mathbf{x}_0))\right].$$

The first term is constant; the second term is degenerate, but it is also constant. Thus, we have

$$q^* = \arg\max_{\{\tau_{\mathbf{x}_0}\}} \mathbb{E}_{p(\mathbf{x}_0)}\left[\lambda R(\tau_{\mathbf{x}_0}) + \log p(\tau_{\mathbf{x}_0} \mid \pi, \mathbf{x}_0)\right]. \tag{7}$$

Thus, we can optimize Eq (3) by alternatingly optimizing Eq (5) and Eq (7).

We interpret these equations as adaptive teaching. At a high level, the teacher (i.e., the variational distribution q^* in Eq (7)) is used to guide the optimization of the student (i.e., the state machine policy π^* in Eq (5)). Rather than compute the teacher in closed form, we approximate it by sampling finitely many initial states $\mathbf{x}_0^k \sim X_0$ and then computing the optimal rollout from $\mathbf{x}_0^k$. Formally, on the ith iteration, the teacher and student are updated as follows:

$$\textbf{Teacher} \quad q_i^* = \sum_{k=1}^{K} \delta(\tau_k^i) \tag{8}$$

$$\text{where} \quad \tau_k^i = \arg\max_{\tau} \ \lambda R(\tau) + \log p(\tau \mid \pi^{i-1}, \mathbf{x}_0^k) \qquad (\mathbf{x}_0^k \sim X_0)$$

$$\textbf{Student} \quad \pi_i^* = \arg\max_{\pi} \ \sum_{k=1}^{K} \log p(\tau_k^i \mid \pi, \mathbf{x}_0^k) \tag{9}$$

The teacher objective Eq (8) is to both maximize the reward $R(\tau)$ from a random initial state $\mathbf{x}_0$ and to maximize the probability $p(\tau \mid \pi, \mathbf{x}_0)$ of obtaining the rollout τ from initial state $\mathbf{x}_0$ according to the current student π. The latter encourages the teacher to match the structure of the student. Furthermore, the teacher is itself updated at each step to account for the changing structure of the student. The student objective Eq (4) is to imitate the distribution of rollouts according to the teacher. Figure 2 shows the different components of our algorithm.

4.2 Teacher: Computing Loop-Free Policies

We begin by describing how the teacher solves the trajectory optimization problem Eq (8)—i.e., computing τ_k for a given initial state $\mathbf{x}_0^k$.

Parameterization. One approach is to parameterize τ as an arbitrary action sequence $(\mathbf{a}_0, \mathbf{a}_1, ...)$ and use gradient-based optimization to compute τ. However, this approach can perform poorly—even though we regularize τ towards the student, it could exhibit behaviors that are hard for the student to capture. Instead, we parameterize τ in a way that mirrors the student. In particular, we parameterize τ like a state machine, but rather than having modes and switching conditions that adaptively determine the sequence of action functions to be executed and the duration of execution, the sequence of action functions is fixed and each action function is executed for a fixed duration.

More precisely, we represent τ as an *loop-free policy* $\tau = \langle \mathcal{H}, \mathcal{T} \rangle$. To execute τ, each action function $H_i \in \mathcal{H}$ is applied for the corresponding duration $T_i \in \mathcal{T}$, after which H_{i+1} is applied. The action functions are from the same grammar of action functions for the student.

The obvious way to represent a duration T_i is as a number of time steps $T_i \in \mathbb{N}$. However, with this choice, we cannot use continuous optimization to optimize T_i. Instead, we fix the number of discretization steps P for which H_i is executed, and vary the time increment $\Delta_i = T_i/P$—i.e., $\mathbf{x}_{n+1} \approx \mathbf{x}_n + F(\mathbf{x}_n, H_i(\mathbf{o})) \cdot \Delta_i$. We enforce $\Delta_i \leq \Delta_{\text{max}}$ for a small Δ_{max} to ensure that the discrete-time approximation of the dynamics is sufficiently accurate.

Optimization. We use model-based trajectory optimization to compute loop-free policies. The main challenge is handling the term $p(\tau \mid \pi, \mathbf{x}_0)$ in the objective. Thus, we perform trajectory optimization in two phases. First, we use the a sampling-based optimization algorithm to obtain a set of good trajectories $\tau^1, ..., \tau^L$. Then, we apply gradient-based optimization, replacing $p(\cdot \mid \pi, \mathbf{x}_0)$ with a term that regularizes τ to be close to $\{\tau^\ell\}_{\ell=1}^L$.

The first phase proceeds as follows: (i) sample $\tau^1, \cdots, \tau^L$ using π from $\mathbf{x}_0$, and let p^ℓ be the probability of τ^ℓ according to π, (ii) sort these samples in decreasing order of objective $p^\ell \cdot e^{\lambda R(\tau^\ell)}$, and (iii) discard all but the top ρ samples. This phase essentially performs one iteration of CEM (Mannor et al., 2003). Then, in the second phase, we replace the probability expression with

$$p(\tau \mid \pi, \mathbf{x}_0) \approx \frac{\sum_{\ell=1}^{\rho} p^\ell \cdot e^{-d(\tau, \tau^\ell)}}{\sum_{\ell=1}^{\rho} p^\ell}, \tag{10}$$

which we use gradient-based optimization to optimize. Here, $d(\tau, \tau^\ell)$ is a distance metric between two loop-free policies, defined as the L_2 distance between the parameters of τ and τ^ℓ.

4.3 Student: Learning Structured State Machine Policies via Imitation

Next, we describe how the student solves the maximum likelihood problem Eq (9) to compute π^*.

Probabilistic state machines. Although the output of our algorithm is a student policy that is a deterministic state machine, our algorithm internally relies on distributions over states induced by the student policy to guide the teacher. Thus, we represent the student policy as a probabilistic state machine during learning. To do so, we simply make the action functions H_{m_j} and switching conditions $G_{m_{j_1}}^{m_{j_2}}$ probabilistic—instead of constant parameters in the grammar for action functions and switching conditions, now we have Gaussian distributions $\mathcal{N}(\alpha, \sigma)$. Then, when executing π, we obtain i.i.d. samples of the parameters $H'_{m_j} \sim H_{m_j}$ and $\{(G_{m_j}^{m'_j})' \sim G_{m_j}^{m'_j}\}_{m'_j}$ every time we switch to mode m_j, and act according to H'_{m_j} and $\{(G_{m_j}^{m'_j})'\}$ until the mode switches again. By re-sampling these parameters on every mode switch, we avoid dependencies across different parts of a rollout or different rollouts. On the other hand, by not re-sampling these parameters within a mode switch, we ensure that the structure of π remains intact within a mode.

Optimization. Each τ_k can be decomposed into segments (k, i) where action function $H_{k,i}$ is executed for duration $T_{k,i}$. Furthermore, for the student π, let H_{m_j} be the action function distribution for mode m_j and $G_{m_{j_1}}^{m_{j_2}}$ be the switching condition distribution for mode m_{j_1} to mode m_{j_2}. Note that H_{m_j} and $G_{m_{j_1}}^{m_{j_2}}$ are distributions whereas $H_{k,i}$ and $T_{k,i}$ are constants. We have

$$p(\tau_k \mid \pi, \mathbf{x}_0^k) = \prod_i p(H_{k,i} \mid \pi, \mathbf{x}_0^k) \cdot p(T_{k,i} \mid \pi, \mathbf{x}_0^k).$$

For each (k, i), let $\mu_{k,i}$ be the latent random variable indicating the ith mode used by π starting from $\mathbf{x}_0^k$; in particular, $\mu_{k,i}$ is a categorical random variable that takes values in the modes $\{m_j\}$. And

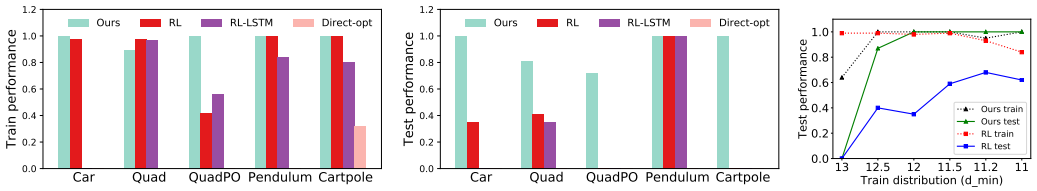


Figure 3: Comparison of performances on train (left) and test (middle) distributions. Our approach outperforms the baselines on all benchmarks in terms of test performance. An empty bar indicates that the policy learned for that experiment failed on all runs. We also plot test performance for different choices of training distribution for the Car benchmark (right).

$\mu_{k,i} = m_j$ means that $H_{k,i} \sim H_{m_j}$ and $T_{k,i}$ is determined by the sampled switching conditions from distributions $\{G^{m'_j}_{m_j}\}$. Assuming the latent variable $\mu_{k,i}$ allows the student to compute π^* by computing $H^*_{m_j}$ and $G^{m_{j2}}_{m_{j1}}{}^*$ separately.

Now, we use a standard expectation maximization (EM) approach to optimizing π, where the E-step computes the distributions $p(\mu_{i,k} = m_j)$ assuming π is fixed, and the M-step optimizes π assuming the probabilities $p(\mu_{i,k} = m_j)$ are fixed. See Appendix A for details.

5 Experiments

Benchmarks. We use 5 control problems, each with different training and test distributions (summarized in Figure 6 in Appendix B): (i) Car, the benchmark in Figure 1, (ii) Quad, where the goal is to maneuver a 2D quadcopter through an obstacle course by controlling its vertical acceleration, where we vary the obstacle course length, (iii) QuadPO, a variant where the obstacles are unobserved but periodic (so the agent can perform well using a repeating motion), (iv) Pendulum, where we vary the pendulum mass, and (v) Cart-Pole, where we vary the time horizon and pole length.

Baselines. We compare against: (i) RL: PPO with a feedforward neural network policy, (ii) RL-LSTM: PPO with an LSTM, (iii) Direct-Opt: learning a state machine policy directly via numerical optimization. Hyper-parameters are chosen to maximize performance on the training distribution. Each algorithm is trained 5 times; we choose the one that performs best on the training distribution.

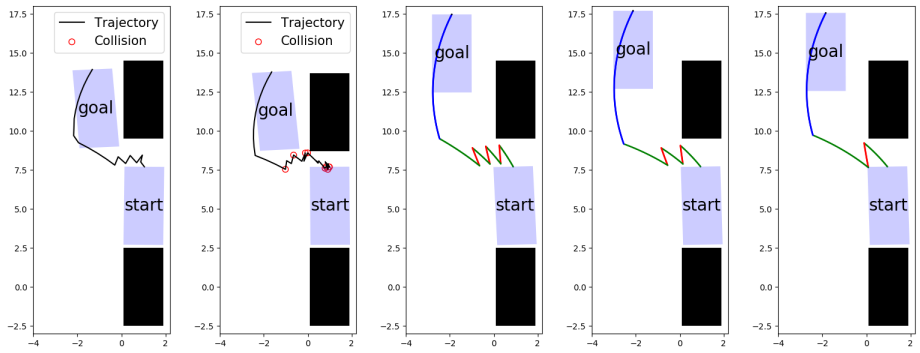
Results. Figure 3 shows results on both training and test distributions. We measure performance as the fraction of rollouts (out of 1000) that both satisfy the safety specfication and reach the goal.

Inductive generalization. For all benchmarks, our policy generalizes well on the test distribution. In three cases, we generalize perfectly (all runs satisfy the metric). For Quad and QuadPO, the policies result in collisions on some runs, but only towards the end of the obstacle course.

Comparison to RL. The RL policies mostly achieve good training performance, but generalize poorly since they over-specialize to states seen during training. The only exception is Pendulum, which has a very small state space; even in this case, the RL policy takes longer to reach the goal than our state machine policy (see Figure 7 in Appendix B). For QuadPO, the RL policy does not achieve a good training performance since the states are partially observed. We may expect the LSTM policies to alleviate this issue. However, the LSTM policies often perform poorly even on the training distribution, and also generalize worse than the feedforward neural network policies.

We empirical analyze the policies. Figure 4 shows the trajectory taken by (a) the RL policy, compared to (c) our policy, from a training initial state. The RL policy does not exhibit a repeating behavior, which causes it to fail on the trajectory from a test state shown in (b). Similarly, Figure 5 (Right) compares the actions taking by our policy to those taken by the RL policy on Quad and QuadPO. Our policy produces smooth repeating actions, whereas the RL policy does not.

Comparison to direct-opt. The state machine policies learned using direct-opt perform poorly even in training, illustrating the need to use adaptive teaching to learn state machine policies.



(a) d = 12.0 (train) (b) d = 11.2 (test) (c) Original. (d) User change 1. (e) User change 2.

Figure 4: (a-c) The RL policy generates unstructured trajectories, and therefore does not generalize from (a) the training distribution to (b) the test distribution. In contrast, our state machine policy in (c) generates a highly structured trajectory that generalizes well. (c-e) A user can modify our state machine policy to improve performance. In (d), the user sets the steering angle to 0.5, and in (e), the user sets the thresholds in the switching conditions $G_{m_1}^{m_2}, G_{m_2}^{m_1}$ to 0.1.

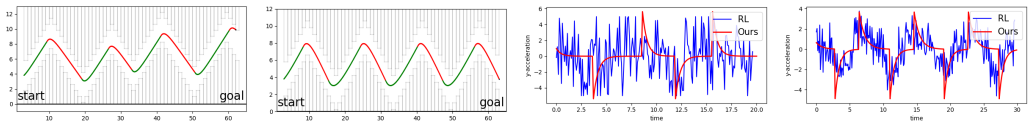


Figure 5: Left: Trajectories for the Quad (leftmost) and QuadPO (second from the left) benchmarks using our state machine policy. Right: Graph of veritcal acceleration over time for both our policy (red) and the neural network policy (blue), for Quad (second from the right) and QuadPO (rightmost).

Varying the training distribution. We study how test performance changes as we vary the training distribution on the Car benchmark. We vary X_0^{train} as $d \sim [d_{\text{min}}, 13.5]$, where $d_{\text{min}} = \{13, 12.5, 12, 11.5, 11.2, 11\}$, but fix X_0^{test} to $d \sim [11, 12]$. Figure 3 (right) shows how test performance varies with d_{min} for both our policy and the RL policy. Our policy inductively generalizes for a wide range of training distributions. In contrast, the test performance of the RL policy initially increases as the train distribution gets bigger, but it eventually starts declining. The reason is that its training performance actually starts to decline. Thus, in some settings, our approach can outperform RL policies even on the training distribution.

Interpretability. An added benefit of our state machine policies is interpretability. In particular, we demonstrate the interpretability of our policies by showing how a user can modify a learned state machine policy. Consider the policy from Figure 1e for the autonomous car. We manually make the following changes: (i) increase the steering angle in H_{m_1} to its maximum value 0.5, and (ii) decrease the gap maintained between the agent and the black cars by changing the switching condition $G_{m_1}^{m_2}$ to $d_f \leq 0.1$ and $G_{m_2}^{m_1}$ to $d_b \leq 0.1$. Figure 4 demonstrates these changes—it shows trajectories obtained using (c) the original policy, (d) the first modified policy, and (e) the second modified policy. There is no straightforward way to make these kinds of changes to a neural network policy.

Conclusion. We have proposed an algorithm for learning state machine policies that inductively generalize to novel environments. Our approach is based on a framework called adaptive teaching that alternatively learns a student that imitates a teacher and a teacher who adapts to the structure of the student. We demonstrate that our policies inductively generalize better than RL policies.

REFERENCES

Douglas Aberdeen and Jonathan Baxter. Scaling internal-state policy-gradient methods for pomdps. In *ICML*, pp. 3–10, 2002.

Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 242–252, Long Beach, California, USA, 09–15 Jun 2019. PMLR. URL http://proceedings.mlr.press/v97/allen-zhu19a.html.

David Andre and Stuart J Russell. State abstraction for programmable reinforcement learning agents. In *AAAI/IAAI*, pp. 119–125, 2002.

Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Advances in Neural Information Processing Systems*, pp. 2494–2504, 2018.

Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017. URL https://openreview.net/forum?id=BkbY4psgg.

Luca Carlone and Giuseppe C. Calafiore. Convex relaxations for pose graph optimization with outliers. *IEEE Robotics and Automation Letters*, 3(2):1160–1167, 2018. doi: 10.1109/LRA.2018.2793352. URL https://doi.org/10.1109/LRA.2018.2793352.

Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Unsupervised learning by program synthesis. In *Advances in neural information processing systems*, pp. 973–981, 2015.

Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In *Advances in Neural Information Processing Systems*, pp. 6060–6069, 2018.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1126–1135. JMLR. org, 2017.

Eric A Hansen. Solving pomdps by searching in policy space. In *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, pp. 211–219. Morgan Kaufmann Publishers Inc., 1998.

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

Sergey Levine and Vladlen Koltun. Guided policy search. In *International Conference on Machine Learning*, pp. 1–9, 2013.

Shie Mannor, Reuven Y Rubinstein, and Yohai Gat. The cross entropy method for fast policy search. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pp. 512–519, 2003.

Nicolas Meuleau, Leonid Peshkin, Kee-Eung Kim, and Leslie Pack Kaelbling. Learning finite-state controllers for partially observable environments. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pp. 427–436. Morgan Kaufmann Publishers Inc., 1999.

Charles Packer, Katelyn Gao, Jernej Kos, Philipp Krähenbühl, Vladlen Koltun, and Dawn Song. Assessing generalization in deep reinforcement learning. *arXiv preprint arXiv:1810.12282*, 2018.

Ronald Parr and Stuart J Russell. Reinforcement learning with hierarchies of machines. In *Advances in neural information processing systems*, pp. 1043–1049, 1998.

Leonid Peshkin, Nicolas Meuleau, and Leslie Kaelbling. Learning policies with external memory. *arXiv preprint cs/0103003*, 2001.

Pascal Poupart and Craig Boutilier. Bounded finite state controllers. In *Advances in neural information processing systems*, pp. 823–830, 2004.

Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. Houdini: Lifelong learning as program synthesis. In *Advances in Neural Information Processing Systems*, pp. 8701–8712, 2018.

Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. *arXiv preprint arXiv:1804.02477*, 2018.

Abhinav Verma, Hoang Minh Le, Yisong Yue, and Swarat Chaudhuri. Imitation-projected policy gradient for programmatic reinforcement learning. *CoRR*, abs/1907.05431, 2019. URL `http://arxiv.org/abs/1907.05431`.

Martin J Wainwright, Michael I Jordan, et al. Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, 1(1–2):1–305, 2008.

Halley Young, Osbert Bastani, and Mayur Naik. Learning neurosymbolic generative models via program synthesis. In *International Conference on Machine Learning*, pp. 7144–7153, 2019.

He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. An inductive synthesis framework for verifiable reinforcement learning. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 686–701. ACM, 2019.

A EXPECTATION MAXIMIZATION FOR STUDENT OPTIMIZATION

A.1 COMPUTING $p(\tau \mid \pi, \mathbf{x}_0)$

First, note that we have

$$p(H_{k,i} \mid \pi, \mathbf{x}_0^k) = \sum_j p(H_{k,i} \mid H_{m_j}) \cdot p(\mu_{k,i} = m_j).$$

Similarly, the duration $T_{k,i}$ is determined both by the current mode $\mu_{i,k} = m_{j_1}$, and by the switching conditions $G^-_{m_{j_1}} = \{G^{m_{j_2}}_{m_{j_1}}\}_{m_{j_2}}$ from the current mode m_{j_1} into some other mode m_{j_2}. More precisely, let $\gamma_{k,i}$ denote the trajectory on the (k, i) segment of τ_k, and let $\zeta(\gamma_{k,i}, G^-_{m_j})$ denote the earliest time at which a switching condition $G \in G^-_{m_j}$ becomes true along $\gamma_{k,i}$. Since $G \in G^-_{m_j}$ are distributions, $\zeta(\gamma_{k,i}, G^-_{m_j})$ is a distribution on transition times. Then, we have

$$p(T_{i,k} \mid \pi, \mathbf{x}_0^k) = \sum_{m_{j_1}} \sum_{m_{j_2}} p(\mu_{i,k} = m_{j_1}) \cdot p(\mu_{i+1,k} = m_{j_2}) \cdot p(T_{i,k} \mid G^{m_{j_2}}_{m_{j_1}}, G^-_{m_{j_1}})$$

$$p(T_{k,i} \mid G^{m_{j_2}}_{m_{j_1}}, G^-_{m_{j_1}}) = p(T_{k,i} = \zeta(\gamma_{k,i}, G^{m_{j_2}}_{m_{j_1}})) \cdot \prod_{m_{j_3} \neq m_{j_2}} p(T_{k,i} < \zeta(\gamma_{k,i}, G^{m_{j_3}}_{m_{j_1}})).$$

In other words, $T_{i,k}$ is the duration until $G^{m_{j_2}}_{m_{j_1}}$ triggers, conditioned on none of the conditions $G^{m_{j_3}}_{m_{j_1}}$ triggering (where $m_{j_3} \neq m_{j_2}$).

A.2 OPTIMIZING THE STUDENT POLICY

We use expectation minimization (EM) to optimize π. The E-step computes the probability distributions $p(\mu_{k,i} = m_j)$ for a fixed π, and the M-step optimizes H_{m_j} and $G^{m_{j_2}}_{m_{j_1}}$ given $p(\mu_{k,i} = m_j)$.

E-step. Assuming π is fixed, we have

$$p(\mu_{k,i} = m_j \mid \pi, \{\tau_k\}) = \frac{p(H_{k,i} \mid H_{m_j}) \cdot p(T_{k,i} = \zeta(\gamma_{k,i}, G^-_{m_j}))}{\sum_{m'_j} p(H_{k,i} \mid H_{m'_j}) \cdot p(T_{k,i} = \zeta(\gamma_{k,i}, G^-_{m'_j}))}. \tag{11}$$

M-step. Assuming $p(\mu_{i,k} = m_j)$ is fixed, we solve

$$\arg\max_{\{H_{m_j}\}} \sum_{k,i} p(\mu_{k,i} = m_j) \cdot \log p(H_{k,i} \mid H_{m_j}) \tag{12}$$

$$\begin{aligned}\arg\max_{\{G^{m_{j_2}}_{m_{j_1}}\}} \sum_{k,i} & p(\mu_{k,i} = m_{j_1}) \cdot p(\mu_{k,i+1} = m_{j_2}) \cdot \log p(T_{k,i} = \zeta(\gamma_{k,i}, G^{m_{j_2}}_{m_{j_1}})) \\ & + p(\mu_{k,i} = m_{j_1}) \cdot (1 - p(\mu_{k,i+1} = m_{j_2})) \cdot \log p(T_{k,i} < \zeta(\gamma_{k,i}, G^{m_{j_2}}_{m_{j_1}}))\end{aligned} \tag{13}$$

For $G^{m_{j_2}}_{m_{j_1}}$, the first term handles the case $\mu_{k,i+1} = m_{j_2}$, where we maximize the probability that $G^{m_{j_2}}_{m_{j_1}}$ makes the transition at duration $T_{k,i}$, and the second term handles the case $\mu_{k,i+1} \neq m_{j_2}$, where we maximize the probability that $G^{m_{j_2}}_{m_{j_1}}$ does not make the transition until after duration $T_{k,i}$.

We briefly discuss how to solve these equations. For action functions, suppose that H encodes the distribution $\mathcal{N}(\boldsymbol{\alpha}_H, \sigma_H^2)$ over action function parameters. Then, we have

$$\alpha^*_{H_{m_j}} = \frac{\sum_{k,i} p(\mu_{k,i} = m_j) \cdot \alpha_{H_{k,i}}}{\sum_{k,i} p(\mu_{k,i} = m_j)}$$

$$(\sigma^*_{H_{m_j}})^2 = \frac{\sum_{k,i} p(\mu_{k,i} = m_j) \cdot (\alpha_{H_{k,i}} - \alpha^*_{H_{m_j}})(\alpha_{H_{k,i}} - \alpha^*_{H_{m_j}})^T}{\sum_{k,i} p(\mu_{k,i} = m_j)}$$

Solving for the parameters of $G^{m_{j_2}}_{m_{j_1}}$ is more challenging, since there can be multiple kinds of expressions in the grammar that are switching conditions, which correspond to discrete parameters.

We first enumerate over these discrete choices; for each one, we encode Eq (13) as a numerical optimization and solve it to get the means. We compute the standard deviations by computing the deviation on a custom metric involving the parameters and the times. Computing the optimal parameters for switching conditions is more expensive than doing so for action functions. Thus, on each student iteration, we iteratively solve Eq (11) and Eq (12) multiple times, but only solve Eq (13) once.

B ADDITIONAL RESULTS

Bench	#A	#O	X_0^{train}	X_0^{test}	# modes	A_G	C_G
Car	2	5	d $\sim$ [12,13.5]m	d $\sim$ [11,12]m	3	Constant	Boolean tree (depth 1)
Quad	1	8	x dist = 40m	x dist = 80m	2	Proportional	Boolean tree (depth 1)
QuadPO	1	4	x dist = 60m	x dist = 120m	2	Proportional	Boolean tree (depth 1)
Pendulum	1	2	mass $\sim$ [1,1.5]kg	mass $\sim$ [1.5,5]kg	2	Constant	Boolean tree (depth 2)
Cartpole	1	4	time = 5s, len = 0.5	time = 300s, len = 1.0	2	Constant	Boolean tree (depth 2)
Acrobot	1	4	masses = [0.2,0.5]	masses = [0.5,2]	2	Constant	Boolean tree (depth 2)
Mountain car	1	2	power = [5,15]e-4	power = [3,5]e-4	2	Constant	Boolean tree (depth 1)

Figure 6: Summary of our benchmarks. #A is the action dimension, #O is the observation dimension, X_0^{train} is the set of initial states used for training, X_0^{test} is the set of initial states used to test the inductive generalizability, # modes is the number of modes in the state machine policy, and A_G and C_G are the grammars for action functions and switching conditions, respectively. Depth of C_G indicates the number of levels in the Boolean tree.

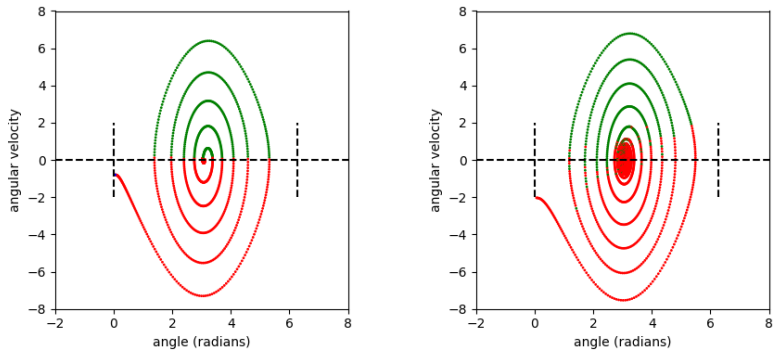


Figure 7: Trajectories taken by our state machine policy (left) and the RL policy (middle) on Pendulum for a test environment (i.e., heavier pendulum). Green (resp., red) indicates positive (resp., negative) torque. Our policy performs optimally by using positive torque when angular velocity ≥ 0 and negative torque otherwise. In contrast, the RL policy performs suboptimally (especially in the beginning of the trajectory).

		Performance on Train dist.		Performance on Test dist.	
Bench	Algorithm	G	T_G	G	T_G
Acrobot	Ours	0.08	7.9s	0.02	31.8s
	RL	0.16	6.5s	0.0	45.2s
	Direct-opt	$\perp$	$\perp$	$\perp$	$\perp$
Mountain car	Ours	0.001	168.5s	0.008	290.1s
	RL	0.0	98.7s	0.0	214.7s
	Direct-opt	0.006	25.0s	2.18	216.0s

Figure 8: Experiment results for additional benchmarks. G is the average goal error (closer to 0 is better). T_G is the average number of timesteps to reach the goal (lower the better). $\perp$ indicates timeout. We can see that both our approach and RL generalizes for these benchmarks.