Interpretable, Verifiable, and Robust Reinforcement Learning via Program Synthesis

Osbert Bastani[1(✉)], Jeevana Priya Inala[2], and Armando Solar-Lezama[3]

[1] University of Pennsylvania, Philadelphia, PA 19104, USA
obastani@seas.upenn.edu
[2] Microsoft Research, Redmond, WA 98052, USA
jinala@microsoft.com
[3] Massachusetts Institute of Technology, Cambridge, MA 02139, USA
asolar@csail.mit.edu

Abstract. Reinforcement learning is a promising strategy for automatically training policies for challenging control tasks. However, state-of-the-art deep reinforcement learning algorithms focus on training deep neural network (DNN) policies, which are black box models that are hard to interpret and reason about. In this chapter, we describe recent progress towards learning policies in the form of programs. Compared to DNNs, such *programmatic policies* are significantly more interpretable, easier to formally verify, and more robust. We give an overview of algorithms designed to learn programmatic policies, and describe several case studies demonstrating their various advantages.

Keywords: Interpretable reinforcement learning · Program synthesis

1 Introduction

Reinforcement learning is a promising strategy for learning control policies for challenging sequential decision-making tasks. Recent work has demonstrated its promise in applications including game playing [34,43], robotics control [14,31], software systems [13,30], and healthcare [6,37]. A typical strategy is to build a high-fidelity simulator of the world, and then use reinforcement learning to train a control policy to act in this environment. This policy makes decisions (e.g., which direction to walk) based on the current state of the environment (e.g., the current image of the environment captured by a camera) to optimize the cumulative reward (e.g., how quickly the agent reaches its goal).

There has been significant recent progress on developing powerful *deep* reinforcement learning algorithms [33,41], which train a policy in the form of a deep neural network (DNN) by using gradient descent on the DNN parameters to optimize the cumulative reward. Importantly, these algorithms treat the underlying environment as a black box, making them very generally applicable.

A. Holzinger et al. (Eds.): xxAI 2020, LNAI 13200, pp. 207–228, 2022.
https://doi.org/10.1007/978-3-031-04083-2_11

A key challenge in many real-world applications is the need to ensure that the learned policy continues to act correctly once it is deployed in the real world. However, DNN policies are typically very difficult to understand and analyze, making it hard to make guarantees about their performance. The reinforcement learning setting is particularly challenging since we need to reason not just about isolated predictions but about sequences of highly connected decisions.

As a consequence, there has been a great deal of recent interest in learning policies in the form of programs, called *programmatic policies*. Such policies include existing interpretable models such as decision trees [9], which are simple programs composed of if-then-else statements, as well as more complex ones such as state machines [26] and list processing programs [27,50]. In general, programs have been leveraged in machine learning to achieve a wide range of goals, such as representing high-level structure in images [16,17,25,46,47,53] and classifying sequence data such as trajectories or text [12,42].

Programmatic policies have a number of advantages over DNN policies that make it easier to ensure they act correctly. For instance, programs tend to be significantly more interpretable than DNNs; as a consequence, human experts can often understand and debug behaviors of a programmatic policy [26,27,50]. In addition, in contrast to DNNs, programs have discrete structure, which make them much more amenable to formal verification [3,9,39], which can be used to prove correctness properties of programmatic policies. Finally, there is evidence that programmatic policies are more robust than their DNN counterparts—e.g., they generalize better to changes in the task or robot configuration [26].

A key challenge with learning programmatic policies is that state-of-the-art reinforcement learning algorithms cannot be applied. In particular, these algorithms are based on the principle of gradient descent on the policy parameters, yet programmatic policies are typically non-differentiable (or at least, their optimization landscape contains many local minima). As a consequence, a common strategy to learning these policies is to first learn the DNN policy using deep reinforcement learning, and then using imitation learning to compress the DNN into a program. Essentially, this strategy reduces the reinforcement learning problem for programmatic policies into a supervised learning problem, for which efficient algorithms often exist—e.g., based on program synthesis [21]. A refinement of this strategy is to adaptively update the DNN policy to mirror the programmatic policy, which reduces the gap between the DNN and the program [26,49].

In this chapter, we provide an overview of recent progress in this direction. We begin by formalizing the reinforcement learning problem (Sect. 2); then, we describe interesting kinds of programmatic policies that have been studied (Sect. 3), algorithms for learning programmatic policies (Sect. 4), and case studies demonstrating the value of programmatic policies (Sect. 5).

2 Background on Reinforcement Learning

We consider a reinforcement learning problem formulated as a *Markov decision process (MDP)* $M = (S, A, P, R)$ [36], where S is the set of states, A is the set of actions, $P(s' \mid a, s) \in [0, 1]$ is the probability of transitioning from state $s \in S$

to state $s' \in S$ upon taking action $a \in A$, and $R(s,a) \in \mathbb{R}$ is the reward accrued by taking action a in state s.

Given an MDP M, our goal is to train an agent that acts in M in a way that accrues high cumulative reward. We represent the agent as a policy $\pi : S \to A$ mapping states to actions. Then, starting from a state $s \in S$, the agent selects action $a = \pi(s)$ according to the policy, observes a reward $R(s,a)$, transitions to the next state $s' \sim P(\cdot \mid s,a)$, and then iteratively continues this process starting from s'. For simplicity, we assume that a deterministic initial state $s_1 \in S$ along with a fixed, finite number of steps $H \in \mathbb{N}$. Then, we formalize the trajectory taken by the agent as a *rollout* $\zeta \in (S \times A \times \mathbb{R})^H$, which is a sequence of state-action-reward tuples $\zeta = ((s_1, a_1, r_1), ..., (s_H, a_H, r_H))$. We can sample a rollout by taking $r_t = R(s_t, a_t)$ and $s_{t+1} \sim P(\cdot \mid s_t, a_t)$ for each $t \in [H] = \{1, ..., H\}$; we let $D^{(\pi)}(\zeta)$ denote the distribution over rollouts induced by using policy π.

Now, our goal is to choose a policy $\pi \in \Pi$ in a given class of policies Π that maximizes the expected reward accrued. In particular, letting $J(\zeta) = \sum_{t=1}^{H} r_t$ be the cumulative reward of rollout ζ, our goal is to compute

$$\hat{\pi} = \arg\max_{\pi \in \Pi} J(\pi) \qquad \text{where} \qquad J(\pi) = \mathbb{E}_{\zeta \sim D^{(\pi)}}[J(\zeta)],$$

i.e., the policy $\pi \in \Pi$ that maximizes the expected cumulative reward over the induced distribution of rollouts $D^{(\pi)}(\zeta)$.

As an example, we can model a robot navigating a room to reach a goal as follows. The state $(x, y) \in S = \mathbb{R}^2$ represents the robot's position, and the action $(v, \phi) \in A = \mathbb{R}^2$ represents the robot's velocity v and direction ϕ. The transition probabilities are $P(s' \mid s, a) = \mathcal{N}(f(s,a), \Sigma)$, where

$$f((x,y),(v,\phi)) = (x + v \cdot \cos\phi \cdot \tau, y + v \cdot \sin\phi \cdot \tau),$$

where $\tau \in \mathbb{R}_{>0}$ is the time increment, and where $\Sigma \in \mathbb{R}^{2\times 2}$ is the variance in the state transitions due to stochastic perturbations. Finally, the rewards are the distance to the goal—i.e., $R(s,a) = -\|s - g\|_2 + \lambda \cdot \|a\|_2$, where $g \in \mathbb{R}^2$ is the goal and $\lambda \in \mathbb{R}_{>0}$ is a hyperparameter. Intuitively, the optimal policy $\hat{\pi}$ for this MDP takes actions in a way that maximizes the time the robot spends close to the goal g, while avoiding very large (and therefore costly) actions.

3 Programmatic Policies

The main difference in programmatic reinforcement learning compared to traditional reinforcement learning is the choice of policy class Π. In particular, we are interested in cases where Π is a space of programs of some form. In this section, we describe specific choices that have been studied.

3.1 Traditional Interpretable Models

A natural starting point is learning policies in the form of traditional interpretable models, including decision trees [10] and rule lists [52]. In particular, these models can be thought of as simple programs composed of simple primitives such as if-then-else rules and arithmetic operations. For example, in Fig. 1, we

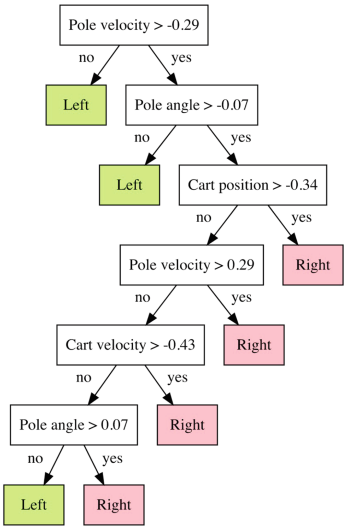
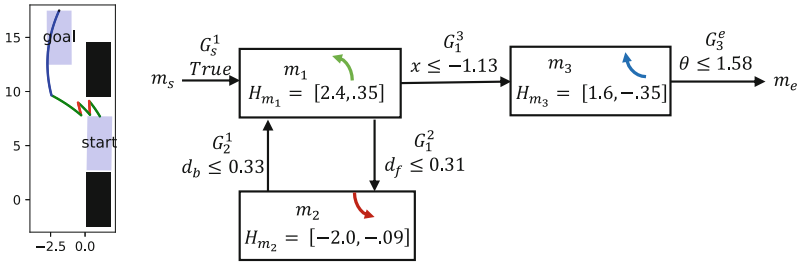


Fig. 1. A decision tree policy trained to control the cart-pole model; it achieves near-perfect performance. Adapted from [9].

show an example of a decision tree policy trained to control the cart-pole robot, which consists of a pole balanced on a cart and the goal is to move the cart back and forth to keep the pole upright [11]. Here, the state consists of the velocity and angle of each the cart and the pole (i.e., $S \subseteq \mathbb{R}^4$), and the actions are to move the cart left or right (i.e., $A = \{\text{left}, \text{right}\}$). As we discuss in Sect. 5, these kinds of policies provide desirable properties such as interpretability, robustness, and verifiability. A key shortcoming is that they have difficulty handling more complex inputs, e.g., sets of other agents, sequences of observations, etc. Thus, we describe programs with more sophisticated components below.



(a) Example of a rollout

(b) State machine policy

Fig. 2. (a) A depiction of the task, which is to drive the blue car (the agent) out from between the two stationary black cars. (b) A state machine policy trained to solve this task. Adapted from [26]. (Color figure online)

3.2 State Machine Policies

A key shortcoming of traditional interpretable models is that they do not possess internal state—i.e., the policy cannot propagate information about the current time step to the next time step. In principle, for an MDP, keeping internal state is not necessary since the state variable contains all information necessary to act optimally. Nevertheless, in many cases, it can be helpful for the policy to keep internal state—for instance, for motions such as walking or swimming that repeat iteratively, it can be helpful to internally keep track of progress within the current iteration. In addition, if the state is partially observed (i.e., the policy only has access to $o = h(s)$ instead of the full state s), then internal state may be necessary to act optimally [28]. In the context of deep reinforcement learning, recurrent neural networks (RNNs) can be used to include internal state [23].

For programmatic policies, a natural analog is to use polices based on finite-state machines. In particular, *state machine policies* are designed to be interpretable while including internal state [26]. Its internal state records one of a finite set of possible *modes*, each of which is annotated with (i) a simple policy for choosing the action when in this mode (e.g., a linear function of the state), and (ii) rules for when to transition to the next mode (e.g., if some linear inequality becomes satisfied, then transition to a given next mode). These policies are closely related to *hybrid automata* [2,24], which are models of a subclass of dynamical systems called *hybrid systems* that include both continuous transitions (modeled by differential equations) and discrete, discontinuous ones (modeled by a finite-state machine). In particular, the closed-loop system consisting of a state-machine policy controlling a hybrid system is also a hybrid system.

As an example, consider Fig. 2; the blue car (the agent) is parked between two stationary black cars, and its goal is to drive out of its parking spot into the goal position while avoiding collisions. The state is $(x, y, \theta, d) \in \mathbb{R}^4$, where (x, y) is the center of the car, θ is its orientation, and d is the distance between the two black cars. The actions are $(v, \psi) \in \mathbb{R}^2$, where v is the velocity and ψ is the steering angle. The transitions are the standard bicycle dynamics [35].

In Fig. 2b, we show the 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, it 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.

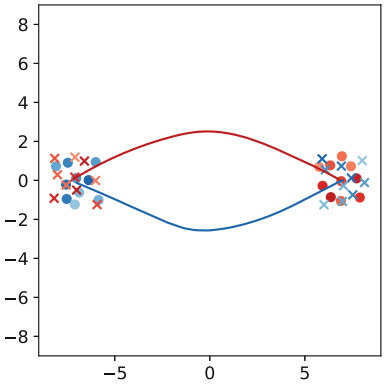
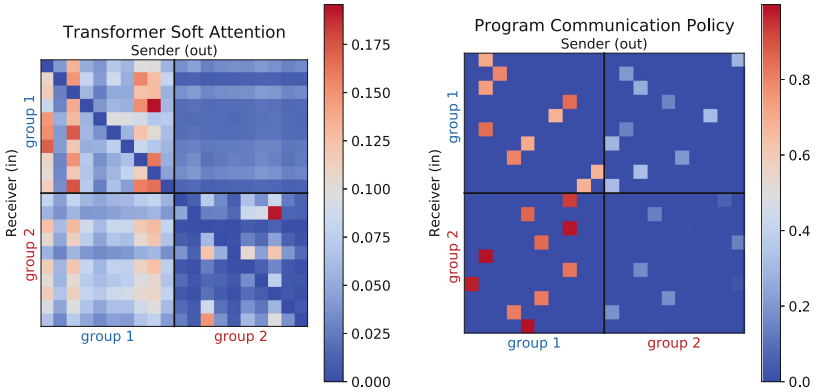


Fig. 3. Two groups of agents (red vs. blue) at their initial positions (circles) trying to reach their goal positions (crosses). The solid line shows the trajectory taken by a single agent in each group. (Color figure online)



(a) DNN attention (b) Program attention

$$R_1 : \mathsf{argmax}(\mathsf{map}(-d^{i,j}, \mathsf{filter}(\theta^{i,j} \geq -1.85, \ell))),$$
$$R_2 : \mathsf{random}(\mathsf{filter}(d^{i,j} \geq 3.41, \ell)).$$

(c) Programmatic attention rules

Fig. 4. (a) Soft attention computed by a DNN for the agent along the y-axis deciding whether to focus on the agent along the x-axis. (b) Sparse attention computed by a program. (c) Program used by each agent to select other agents to focus on. Adapted from [27].

3.3 List Processing Programs

Another kind of programmatic policy is list processing programs, which are compositions of components designed to manipulate lists—e.g., the map, filter,

and fold operators [18]; the set of possible components can be chosen based on the application. In contrast to state machine policies, list processing programs are designed to handle situations where the state includes lists of elements. For example, in multi-agent systems, the full state consists of a list of states for each individual agent [27]. In this case, the program must compute a single action based on the given list of states. Alternatively, for environments with variable numbers of objects, the set of object positions must be encoded as a list. Finally, they can also be used to choose actions based on the history of the previous k states [50], which achieves a similar goal as state machine policies.

As an example, consider the task in Fig. 3, where agents in group 1 (blue) are navigating from the left to their goal on the right, while agents in group 2 (red) are navigating from the right to their goal on the left. The system state $s \in \mathbb{R}^{2k}$ is a list containing the position (x_i, y_i) of each agent of the k agents. An action $a \in \mathbb{R}^{2k}$ consists of the velocities (v_i, w_i) to be applied by each agent. We consider a strategy where we use a single policy $\pi : S \times [k] \to \mathbb{R}^2$, which takes as input the system state along with the index of the current agent $i \in [k]$, and produces the action $\pi(s, i)$ to be taken by agent i. This policy is applied to each agent to construct the full list of actions.

To solve this task, each agent must determine which agents to focus on; in the example in Fig. 3, it is useful to attend to the closest neighbor in the same group (to avoid colliding with them), as well as with an arbitrary agent from the opposite group (to coordinate so their trajectories do not collide).

For now, we describe programmatic policies for each agent designed to select a small number of other agents to focus on. This list of agents can in principle be processed by a second programmatic policy to determine the action to choose; however, in Sect. 3.4, we describe a strategy that combines them with a neural network policy to select actions. Figure 4c shows an example of a programmatic policy that each agent can use to choose other agents to focus on for the task in Fig. 3. This program consists of two rules, each of which selects a single agent to focus on; the program returns the set consisting of both selected agents. In each of these rules, agent i is selecting over other agents j in the list ℓ; $d^{i,j}$ is the distance between them and $\theta^{i,j}$ is the angle between them. Intuitively, rule R_1 chooses the nearest other agent j such that $\theta^{i,j} \in [-1.85, \pi]$, which is likely an agent in the same group as agent i that is directly in front of agent i; thus, agent i needs to focus on it to avoid colliding into it. In contrast, R_2 chooses a random agent from the agents that are far away, which is likely an agent in the other group; thus, agent i can use this information to avoid the other group.

3.4 Neurosymbolic Policies

In some settings, we want part of the policy to be programmatic, but other parts of the policy to be DNNs. We refer to policies that combine programs and DNNs as *neurosymbolic policies*. Intuitively, the program handles part of the computation that we would like to be interpretable, whereas the DNN handles the remainder of the computation (potentially the part that cannot be easily approximated by an interpretable model).

One instance of this strategy is to leverage programs as the attention mechanism for a transformer model [27]. At a high level, a transformer [48] is a DNN that operates on a list of inputs. These models operate by first choosing a small subset of other elements of the list to focus on (the attention layer), then uses a fully-connected layer to decide what information from the other agents is useful (the value layer), and finally uses a second fully-connected layer to compute the result (output layer). For example, transformers can be applied to multi-agent systems since it has to reason over the list of other agents.

A neurosymbolic transformer is similar to a transformer but uses programmatic policies for the attention layer; the value layer and the output layer are still neural networks. This architecture makes the attention layer interpretable—e.g., it is easy to understand and visualize why an agent attends to another agent, while still retaining much of the complexity of the original transformer.

For example, the program shown in Fig. 4c can be used to select other agents to attend to in a neurosymbolic transformer; unlike a DNN attention layer, this program is interpretable. An added advantage is that the program produces sparse attention weights; in contrast, a DNN attention layer produces soft attention weights, so every agent needs to attend to every other agent, even if the attention weight is small. Figure 4a shows the soft attention computed by a DNN, and Fig. 4b shows the sparse attention computed by a program.

4 Synthesizing Programmatic Policies

Next, we describe our algorithms for training programmatic policies. We begin by describing the general strategy of first training a deep neural network (DNN) policy using deep reinforcement learning, and then using imitation learning in conjunction with the DNN policy to reduce the reinforcement learning problem for programmatic policies to a supervised learning problem (Sect. 4.1 and 4.2). Then, we describe a refinement of this strategy where the DNN is adaptively updated to better mirror the current programmatic policy (Sect. 4.3). Finally, all of these strategies rely on a subroutine for solving the supervised learning problem; we briefly discuss approaches to doing so (Sect. 4.4).

4.1 Imitation Learning

We focus on the setting of continuous state and action spaces (i.e., $S \subseteq \mathbb{R}^n$ and $A \subseteq \mathbb{R}^m$), but our techniques are applicable more broadly. A number of algorithms have been proposed for computing optimal policies for a given MDP M and policy class Π [44]. For continuous state and action spaces, state-of-the-art deep reinforcement learning algorithms [33,41] consider a parameteric policy class $\Pi = \{\pi_\theta \mid \theta \in \Theta\}$, where the parameters $\Theta \subseteq \mathbb{R}^d$ are real-valued—e.g., π_θ is a DNN and θ are its parameters. Then, they compute π^* by optimizing over θ. One strategy is to use gradient descent on the objective—i.e.,

$$\theta' \leftarrow \theta + \eta \cdot \nabla_\theta J(\pi_\theta).$$

Algorithm 1. Training programmatic policies using imitation learning.

procedure IMITATIONLEARN(M, Q^*, m, n)
  Train oracle policy $\pi^* \leftarrow$ TrainDNN(M)
  Initialize training dataset $Z \leftarrow \varnothing$
  Initialize programmatic policy $\hat{\pi}_0 \leftarrow \pi^*$
  for $i \in \{1, ..., n\}$ **do**
    Sample m trajectories to construct $Z_i \leftarrow \{(s, \pi^*(s)) \sim D^{(\hat{\pi}_{i-1})}\}$
    Aggregate dataset $Z \leftarrow Z \cup Z_i$
    Train programmatic policy $\hat{\pi}_i \leftarrow$ TrainProgram(Z)
  end for
  return Best policy $\hat{\pi} \in \{\hat{\pi}_1, ..., \hat{\pi}_N\}$ on cross validation
end procedure

In particular, the policy gradient theorem [45] encodes how to compute an unbiased estimator of this objective in terms of $\nabla_\theta \pi_\theta$. In general, most state-of-the-art approaches rely on gradient descent on the policy parameters θ. However, such approaches cannot be applied to training programmatic policies, since the search space of programs is typically discrete.

Instead, a general strategy is to use imitation learning to reduce the reinforcement learning problem to a supervised learning problem. At a high level, the idea is to first use deep reinforcement learning to learn an high-performing DNN policy π^*, and then train the programmatic policy $\hat{\pi}$ to imitate π^*.

A naïve strategy is to use an imitation learning algorithm called behavioral cloning [4], which uses π^* to explore the MDP, collects state-action pairs $Z = \{(s, a)\}$ pairs occurring in rollouts $\zeta \sim D^{(\pi^*)}$, and then trains $\hat{\pi}$ using supervised learning on the dataset Z—i.e.,

$$\hat{\pi} = \arg\min_{\pi \in \Pi} \sum_{(s,a) \in Z} \mathbb{1}(\pi(s) = a). \tag{1}$$

Intuitively, the key shortcoming with this approach is that if $\hat{\pi}$ makes a mistake compared to the DNN policy π^*, then it might reach a state s that is very different from the states in the dataset Z. Thus, $\hat{\pi}$ may not know the correct action to take in state s, leading to poor performance. As a simple example, consider a self-driving car, and suppose π^* drives perfectly in the center of lane, whereas $\hat{\pi}$ deviates slightly from the center early in the rollout. Then, it reaches a state never seen in the training data Z, which means $\hat{\pi}$ does not know how to act in this state, so it may deviate further.

State-of-the-art imitation learning algorithms are designed to avoid these issues. One simple but effective strategy is the Dataset Aggregation (DAGGER) algorithm [38], which iteratively retrains the programmatic policy based on the distribution of states it visits. The first iteration is the same as behavioral cloning; in particular, it generates an initial dataset Z_0 using π^* and trains an initial programmatic policy $\hat{\pi}_0$. In each subsequent iteration i, it generates a dataset Z_i using the previous programmatic policy $\hat{\pi}_{i-1}$, and then trains $\hat{\pi}_i$ on Z_i.

This strategy is summarized in Algorithm 1; it has been successfully leveraged to train programmatic policies to solve reinforcement learning problems [50].

4.2 *Q*-Guided Imitation Learning

One shortcoming of Algorithm 1 is that it does not account for the fact that certain actions are more important than others [9]. Instead, the loss function in Eq. 1 treats all state-action pairs in the dataset Z as being equally important. However, in practice, one state-action pair (s, a) may be significantly more consequential than another one (s', a')—i.e., making a mistake $\hat{\pi}(s) \neq a$ might degrade performance by significantly more than a mistake $\hat{\pi}(s') \neq a$.

For example, consider the toy game of Pong in Fig. 8; the goal is to move the paddle to prevent the ball from exiting the screen. Figure 5a shows a state where the action taken is very important; the paddle must be moved to the right, or else the ball cannot be stopped from exiting. In contrast, Fig. 5b shows a state where the action taken is unimportant. Ideally, our algorithm would upweight the former state-action pair and downweight the latter.

One way to address this issue is by leveraging the Q-function, which measures the quality of a state-action pair—in particular, $Q^{(\pi)}(s, a) \in \mathbb{R}$ is the cumulative reward accrued by taking action a in state s, and then continuing with policy π. Traditional imitation learning algorithms do not have access to $Q^{(\pi^*)}$, since π^* is typically a human expert, and it would be difficult to elicit these values. However, the Q function $Q^{(\pi^*)}$ for the DNN policy π^* is computed as a byproduct of many deep reinforcement learning algorithms, so it is typically available in our setting. Given $Q^{(\pi^*)}$, a natural alternative to Eq. 1 is

$$\hat{\pi} = \arg\min_{\pi \in \Pi} \sum_{(s,a) \in Z} (Q^{(\pi^*)}(s, a) - Q^{(\pi^*)}(s, \hat{\pi}(s))). \tag{2}$$

Intuitively, the term $Q^{(\pi^*)}(s, a) - Q^{(\pi^*)}(s, \hat{\pi}(s))$ measures the degradation in performance by taking the incorrect action $\hat{\pi}(s)$ instead of a. Indeed, it can be proven that this objective exactly encodes the gap in performance between $\hat{\pi}$ and π^*—i.e., in the limit of infinite data, it is equivalent to computing

$$\hat{\pi} = \arg\min_{\pi \in \Pi} \{J(\pi^*) - J(\hat{\pi})\}.$$

Finally, a shortcoming of Eq. 2 is that it is not a standard supervised learning problem. To address this issue, we can instead optimize the lower bound

$$Q^{(\pi^*)}(s, a) - Q^{(\pi^*)}(s, \hat{\pi}(s)) \leq \left(Q^{(\pi^*)}(s, a) - \arg\min_{a' \in A} Q^{(\pi^*)}(s, a') \right) \cdot \mathbb{1}(\hat{\pi}(s) = a),$$

which yields the optimization problem

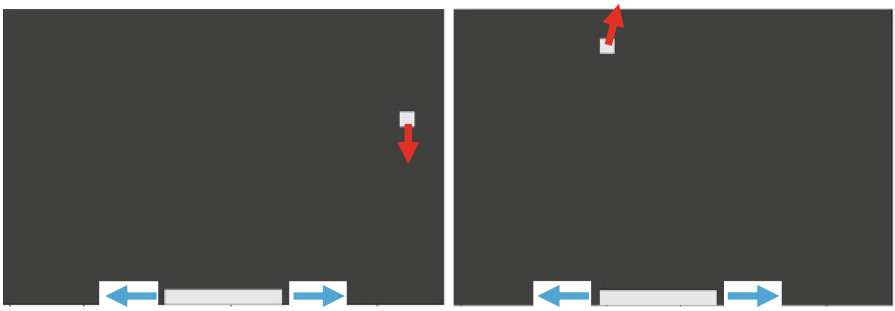
$$\hat{\pi} = \arg\min_{\pi \in \Pi} \sum_{(s,a) \in Z} \left(Q^{(\pi^*)}(s, a) - \arg\min_{a' \in A} Q^{(\pi^*)}(s, a') \right) \cdot \mathbb{1}(\hat{\pi}(s) = a).$$

This strategy is proposed in [9] and shown to learn significantly more compact policies compared to the original DAgger algorithm.

4.3 Updating the DNN Policy

Another shortcoming of Algorithm 1 is that it does not adjust the DNN policy π^* to account for limitations on the capabilities of the programmatic policy $\hat{\pi}$. Intuitively, if $\hat{\pi}$ cannot accurately approximate π^*, then π^* may suggest actions that lead to states where $\hat{\pi}$ cannot perform well, even if π^* performs well in these states. There has been work on addressing this issue. For example, *coaching* can be used to select actions that are more suitable for $\hat{\pi}$ [22]. Alternatively, π^* can be iteratively updated using gradient descent to better reflect $\hat{\pi}$ [49].

A related strategy is *adaptive teaching*, where rather than choosing π^* to be a DNN, it is instead a policy whose structure mirrors that of $\hat{\pi}$ [26]. In this case, we can directly update π^* on each training iteration to reflect the structure of $\hat{\pi}$. As an example, in the case of state machine policies, π^* can be chosen to be a "loop-free" policy, which consists of a linear sequence of modes. These modes can then be mapped to the modes of $\hat{\pi}$, and regularized so that their local policies and mode transitions mirror that of $\hat{\pi}$. Adaptive teaching has been shown to be an effective strategy for learning state machine policies [26].



(a) Critical state (b) Non-critical state

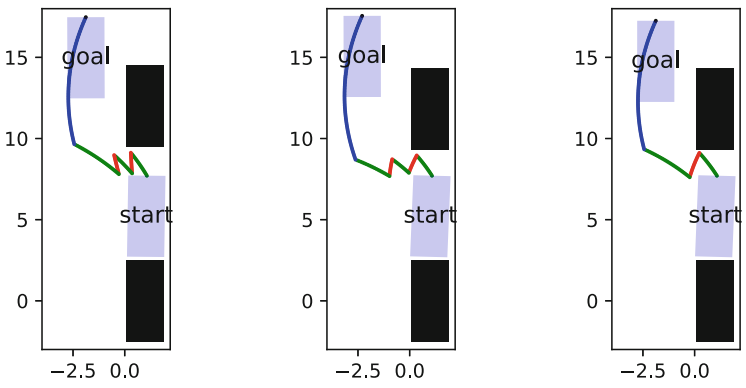
Fig. 5. A toy game of Pong. The paddle is the gray bar at the bottom, and the ball is the gray square. The red arrow shows the direction the ball is traveling, and the blue arrows show the possible actions (move paddle left vs. right). We show examples where the action taken in this state (a) does, and (b) does not significantly impact the cumulative reward accrued from this state. (Color figure online)

4.4 Program Synthesis for Supervised Learning

Recall that imitation learning reduces the reinforcement learning problem for programmatic policies to a supervised learning problem. We briefly discuss algorithms for solving this supervised learning problem. In general, this problem is an instance of *programming by example* [19,20], which is a special case of program synthesis [21] where the task is specified by a set of input-output examples. In our setting, the input-output examples are the state-action pairs in the dataset Z used to train the programmatic policy at each iteration of Algorithm 1.

An added challenge applying program synthesis in machine learning settings is that traditional programming by example algorithms are designed to compute a program that correctly fits *all* of the training examples. In contrast, in machine learning, there typically does not exist a single program that fits all of the training examples. Instead, we need to solve a *quantitative* synthesis problem where the goal is to minimize the number of errors on the training data.

One standard approach to solving such program synthesis problems is to simply enumerate over all possible programmatic policies $\pi \in \Pi$. In many cases, Π is specified as a context-free grammar, in which case standard algorithms can be used to enumerate programs in that grammar (typically up to a bounded depth) [5]. In addition, domain-specific techniques can be used to prune provably suboptimal portions of the search space to speed up enumeration [12]. For particularly large search spaces, an alternative strategy is to use a stochastic search algorithm that heuristically optimzes the objective; for example, Metropolis Hastings can be used to adaptively sample programs (e.g., with the unnormalized probability density function taken to be the objective value) [27,40].



(a) Original (b) Change (i) (c) Change (ii)

Fig. 6. A human expert can modify our state machine policy to improve performance. (a) A trajectory using the original state machine policy shown in Fig. 2(b). (b) The human expert sets the steering angle to the maximum value 0.5. (c) The human expert sets the thresholds in the mode transitions so the blue car drives as close to the black cars as possible. Adapted from [26]. (Color figure online)

5 Case Studies

In this section, we describe a number of case studies that demonstrate the value of programmatic policies, demonstrating their interpretability (Sect. 5.1), verifiability (Sect. 5.2), and robustness (Sect. 5.3).

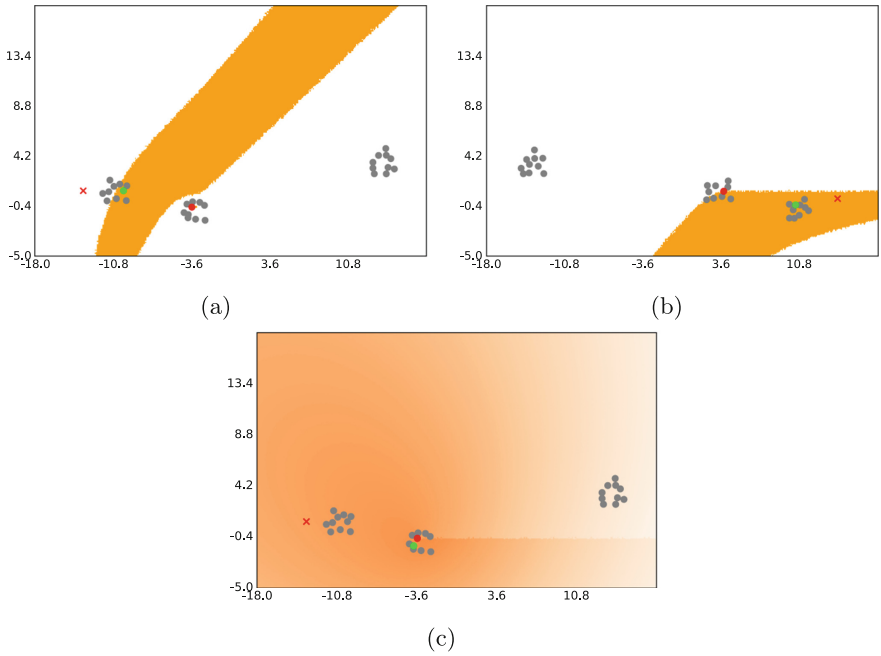


Fig. 7. Visualization of the programmatic attention layer in Fig. 4c, which has two rules R_1 and R_2. In this task, there are three groups of agents. The red circle denotes the agent currently choosing an action, the red cross denotes its goal, and the green circle denotes the agent selected by the rule. (a, b) Visualization of rule R_1 for two different states; orange denotes the region where the filter condition is satisfied—i.e., R_1 chooses a random agent in this region. (c) Visualization of rule R_2, showing the score output by the map operator; darker values are higher—i.e., the rule chooses the agent with the darkest value. Adapted from [27]. (Color figure online)

5.1 Interpretability

A key advantage of programmatic policies is that they are interpretable [26,27,50]. One consequence of their interpretability is that human experts can examine programmatic policies and modify them to improve performance. As an example, consider the state machine policy shown in Fig. 2b in Sect. 3. We have manually made the following changes to this policy: (i) increase the steering angle in mode m_1 to its maximum value 0.5 (so the car steers as much as possible when exiting the parking spot), and (ii) decrease the gap maintained between the agent and the black cars by changing the condition for transitioning from mode m_1 to mode m_2 to $d_f \leq 0.1$, and from mode m_2 to mode m_1 to $d_b \leq 0.1$ (so the blue car drives as far as possible without colliding with a black car before changing directions). Figure 6 visualizes the effects changes; in particular, it shows trajectories obtained using the original policy, the policy with change (i), and the policy with change (ii). As can be seen, the second modified policy exits the

parking spot more quickly than the original policy. There is no straightforward way to make these kinds of changes to improve a DNN policy.

Similarly, we describe how it is possible to interpret programmatic attention layers in neurosymbolic transformers. In particular, Fig. 7 visualizes the synthesized programmatic attention policy described in Sect. 3 for a multi-agent control problem; in this example, there are three groups of agents, each trying to move towards their goals. Figures 7a & 7b visualize rule R_1 in two different states. In particular, R_1 selects a random far-away agent in the orange region to focus on. Note that in both states, the orange region is in the direction of the goal of the agent. Intuitively, the agent is focusing on an agent in the other group that is between itself and the goal; this choice enables the agent to plan a path to its goal that avoids colliding with the other group. Next, Fig. 7c visualizes rule R_2; this rule simply focuses on a nearby agent, which enables the agent to avoid collisions with other agents in the same group.

5.2 Verification

Another key advantage of programmatic policies is that they are significantly easier to formally verify. Intuitively, because they make significant use of discrete control flow structures, it is easier for formal methods to prune branches of the search space corresponding to unreachable program paths.

Verification is useful when there is an additional safety constraint that must be satisfied by the policy in addition to maximizing cumulative reward. A common assumption is that the agent should remain in a safe subset of the state space $S_{\text{safe}} \subseteq S$ during the entire rollout. Furthermore, in these settings, it is often assumed that the transitions are deterministic—i.e., the next state is $s' = f(s, a)$ for some deterministic transition function $f : S \times A \to S$. Finally, rather than considering a single initial state, we instead consider a subset of initial states $S_1 \subseteq S_{\text{safe}}$. Then, we consider the safety constraint that for any rollout ζ starting from $s_1 \in S_1$, we have $s_t \in S_{\text{safe}}$ for all $t \in [H]$; we use $\phi(\pi) \in \{\mathsf{true}, \mathsf{false}\}$ to indicate whether a given policy π satisfies this constraint. Our goal is to solve

$$\pi^* = \underset{\pi \in \Pi_{\text{safe}}}{\arg\max}\, J(\pi) \qquad \text{where} \qquad \Pi_{\text{safe}} = \{\pi \in \Pi \mid \phi(\pi)\}.$$

A standard strategy for verifying safety is to devise a logical formula that encodes a safe rollout; in particular, we can encode our safety constraint as follows:

$$\phi(\pi) \equiv \forall \vec{s}\,.\, \left[(s_1 \in S_1) \wedge \bigwedge_{t=1}^{H} (a_t = \pi(s_t) \wedge s_{t+1} = f(s_t, a_t))\right] \Rightarrow \bigwedge_{t=1}^{H} (s_t \in S_{\text{safe}}),$$

where $\vec{s} = (s_1, ..., s_H)$ are the free variables, and we use $\equiv$ to distinguish equality of logical formulas from equality of variables within a formula. Intuitively, this formula says that if (i) s_1 is an initial state, and (ii) the actions are chosen by π and the transitions by f, then all states are safe.

(a) Recurrent region (b) Failure case

Fig. 8. (a) A recurrent region; proving that the ball always returns to this region implies that the policy plays correctly for an infinite horizon. (b) A failure case found by verification, where the paddle fails to keep the ball from exiting. (Color figure online)

With this expression for $\phi(\pi)$, to prove safety, it suffices to prove that $\neg\phi(\pi) \equiv \mathsf{false}$. The latter equivalence is an instance of Satisfiability Modulo Theory (SMT), and can automatically be checked by an SMT solver [15] as long as predicates of the form $s \in S_{\text{safe}}$, $a = \pi(s)$, and $s' = f(s, a)$ can be expressed in a theory that is supported by the SMT solver. A standard setting is where S_{safe} is a polytope, and π and f are piecewise affine; in these cases, each of these predicates can be expressed as conjunctions and disjunctions of linear inequalities, which are typically supported (e.g., the problem can be reduced to an integer program).

As an example, this strategy has been used to verify that the decision tree policy for a toy game of pong shown in Fig. 1 in Sect. 3 is correct—i.e., that it successfully blocks the ball from exiting. In this case, we can actually prove correctness over an infinite horizon. Rather than prove that the ball does not exit in H steps, we instead prove that for any state s_1 where the ball is in the top half of the screen (depicted in blue in Fig. 8a), the ball returns to this region after H steps. If this property is true, then the ball never exits the screen.

For this property, the SMT solver initially identified a failure case where the ball exits the screen, which is shown in Fig. 8b; in this corner case, the is at the very edge of the screen, and the paddle fails to keep the ball from exiting. This problem can be fixed by manually examining the decision tree and modifying it to correctly handle the failure case; the modified decision tree has been successfully proven to be correct—i.e., it always keeps the ball in the screen.

In another example, we used bounded verification to verify that the state machine policy in Fig. 2b does not result in any collisions for parallel parking task in Fig. 2a. We used dReach [29], an SMT solver designed to verify safety for hybrid systems, which are dynamical systems that include both continuous transitions (modeled using differential equations) and discrete, discontinuous ones (modeled using a finite-state machine). In particular, dReach performs bounded reachability analysis, where it unrolls the state machine modes up to

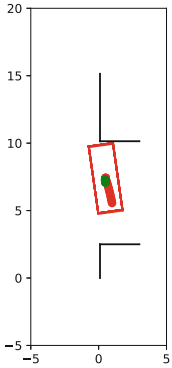


Fig. 9. A failure case found using our verification algorithm with tolerance parameter $\delta = 0.24$ for the state machine policy in Fig. 2b on the parallel parking task in Fig. 2a. Here, the car collides with the car in the front.

some bound. Furthermore, dReach is *sound* and δ-*complete*—i.e., if it says the system is safe, then it is guaranteed to be safe, and if it says the system is unsafe, then there exists some δ-bounded perturbation that renders the system unsafe. Thus, we can vary δ to quantify the robustness of the system to perturbations.

With $\delta = 0.1$, dReach proved that the policy in Fig. 2b is indeed safe for up to an unrolling of 7 modes of the state machine, which was enough for the controller to complete the task from a significant fraction of the initial state space. However, with $\delta = 0.24$, dReach identified a failure case where the car would collide with the car in the front (under some perturbations of the original model); this failure case is shown in Fig. 9. We manually fixed this problem by inspecting the state machine policy in Fig. 2b and modifying the switching conditions $G^{m_2}_{m_1}$ and $G^{m_1}_{m_2}$ to $d_f \leq 0.5$ and $d_b \leq 0.5$, respectively. With these changes, dReach proved that the policy is safe for $\delta = 0.24$.

More generally, similar strategies can be used to verify robustness and stability of programmatic controllers [9,50]. It can also be extended to compute regions of attraction—for instance, to show that a decision tree policy provably stabilizes a pendulum to the origin [39]. To improve performance, one strategy is to compose a provably safe programmatic policy with a higher performing but potentially unsafe DNN policy using *shielding* [1,7,8,32,51]; intuitively, this strategy uses the DNN policy as long as the programmatic policy can ensure safety. Finally, the techniques so far have focused on safety after training the policy; in some settings, it can be desirable to continue running reinforcement learning after deploying the policy to adapt to changing environments. To enable safety during learning, one strategy is to prove safety while accounting for uncertainty in the current model of the environment [3].

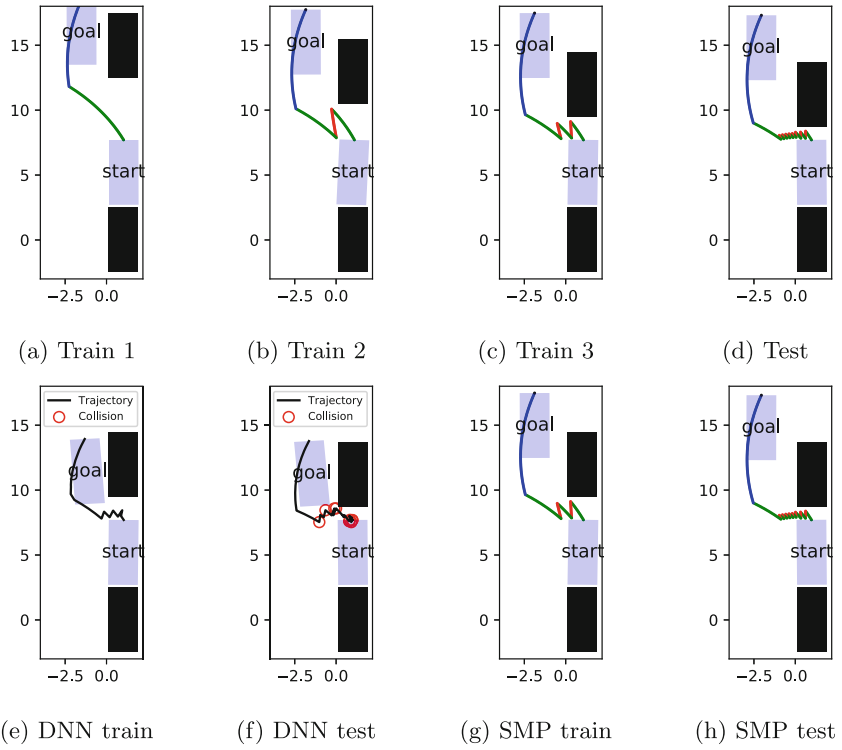
5.3 Robustness

Another advantage of programmatic policies is that they tend to be more robust than DNN policies—i.e., they generalize well to states outside of the distribution on which the policy was trained. For example, it has been shown that a programmatic policy trained to drive a car along one race track can generalize to other race tracks not seen during training, while DNN policies trained in the same way do not generalize as well [50]. We can formalize this notion by considering separate training and test distributions over tasks—e.g., the training distribution over tasks might include driving on just a single race track, whereas the test distribution includes driving on a number of additional race tracks. Then, a policy is robust if it performs well on the test distribution over tasks even when it is trained on the training distribution of tasks.

A special case is *inductive* generalization, where the tasks are indexed by natural numbers $i \in \mathbb{N}$, the training distribution is over small i, and the test distribution is over large i [26]. As a simple example, i may indicate the horizon over which the task is trained; then, a robust policy is one that is trained on short horizon tasks but generalizes to long horizon tasks.

Going back to the parallel parking task from Fig. 2 in Sect. 3; for this task, we can consider inductively generalization of a policy in terms of the number of back-and-forth motions needed to solve the task [26]. In particular, Figs. 10a, 10b, and 10c depict training tasks with relatively few back-and-forth motions, and Fig. 10d depicts a test task with a much larger number of back-and-forth motions. As shown in Fig. 10e, a DNN policy trained using deep reinforcement learning can solve additional tasks from the training distribution; however, Fig. 10f shows that this policy does not generalize to tasks from the test distribution. In contrast, a state machine policy performs well on both additional tasks from the training distribution (Fig. 10g) as well as tasks from the test distribution (Fig. 10h). Intuitively, the state machine policy is learning to the correct back-and-forth motion needed to solve the parallel parking problem. It can do so since (i) it is sufficiently expressive to represent the "correct" solution, yet (ii) it is sufficiently constrained that it learns a systematic policy. In contrast, the DNN policy can likely represent the correct solution, but because it is highly underconstrained, it finds an alternative solution that works on the training tasks, but does not generalize well to the test tasks. Thus, programmatic policies provide a promising balance between expressiveness and structure needed to solve challenging control tasks in a generalizable way.

For an illustration of these distinctions, we show the sequence of actions taken as a function of time by a programmatic policy compared to a DNN policy in Fig. 11. Here, the task is to fly a 2D quadcopter through an obstacle course by controlling its vertical acceleration. As can be seen, the state machine policy produces a smooth repeating pattern of actions; in contrast, the DNN policy acts highly erratically. This example further illustrates how programmatic policies are both complex (evidenced by the complexity of the red curve) yet structured (evidenced by the smoothness of the red curve and its repeating pattern). In contrast, DNN policies are expressive (as evidenced by the complexity of the red curve), but lack the structure needed to generalize robustly.



(a) Train 1 (b) Train 2 (c) Train 3 (d) Test

(e) DNN train (f) DNN test (g) SMP train (h) SMP test

Fig. 10. (a, b, c) Training tasks for the autonomous driving problem in Fig. 2. (d) Test task, which is harder due to the increased number of back-and-forth motions required. (a) The trajectory taken by the DNN policy on a training task. (b) The trajectory taken by the DNN policy on a test task; as can be seen, it has several unsafe collisions. (c) The trajectory taken by the state machine policy (SMP) on a training task. (d) The trajectory taken by the SMP on a test task; as can be seen, it generalizes well to this task. Adapted from [26].

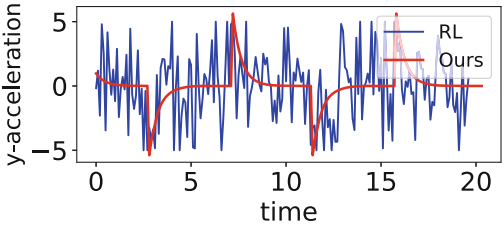


Fig. 11. The vertical acceleration (i.e., action) selected by the policy as a function of time, for each our programmatic policy (red) and a DNN policy (blue), for a 2D quadcopter task. Adapted from [26]. (Color figure online)

6 Conclusions and Future Work

In this chapter, we have describe an approach to reinforcement learning where we train programmatic policies such as decision trees, state machine policies, and list processing programs, instead of DNN policies. These policies can be trained using algorithms based on imitation learning, which first train a DNN policy using deep reinforcement learning and then train a programmatic policy to imitate the DNN policy. This strategy reduces the reinforcement learning problem to a supervised learning problem, that can be solved by existing algorithms such as program synthesis. Through a number of case studies, we have demonstrated that compared to DNN policies, programmatic policies are highly interpretable, are easier to formally verify, and generalized more robustly.

We leave a number of directions for future work. One important challenge is that synthesizing programmatic policies remains costly. Many state-of-the-art program synthesis algorithms rely heavily on domain-specific pruning strategies to improve performance, including strategies targeted at machine learning applications [12]. Leveraging these strategies can significantly increase the complexity of programmatic policies that can be learned in a tractable way.

Another interesting challenge is scaling verification algorithms to more realistic problems. The key limitation of existing approaches is that even if the programmatic policy has a compact representation, the model of the environment often does not. A natural question in this direction is whether we can learn programmatic models of the environment that are similarly easy to formally verify, while being a good approximation of the true environment.

Finally, we have described one strategy for constructing neurosymbolic policies that combine programs and DNNs—i.e., the neurosymbolic transformer. We believe a number of additional kinds of model compositions may be feasible—for example, leveraging a neural network to detect objects and then using a program to reason about them, or using programs to perform high-level reasoning such as path planning while letting a DNN policy take care of low-level control.

References

1. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: Thirty-Second AAAI Conference on Artificial Intelligence (2018)
2. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.-H.: Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) HS 1991-1992. LNCS, vol. 736, pp. 209–229. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57318-6_30
3. Anderson, G., Verma, A., Dillig, I., Chaudhuri, S.: Neurosymbolic reinforcement learning with formally verified exploration. In: Neural Information Processing Systems (2020)
4. Bain, M., Sammut, C.: A framework for behavioural cloning. In: Machine Intelligence 15, pp. 103–129 (1995)

5. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: DeepCoder: learning to write programs. In: International Conference on Learning Representations (2017)
6. Bastani, H., et al.: Deploying an artificial intelligence system for COVID-19 testing at the greek border. Available at SSRN (2021)
7. Bastani, O.: Safe reinforcement learning with nonlinear dynamics via model predictive shielding. In: 2021 American Control Conference (ACC), pp. 3488–3494. IEEE (2021)
8. Bastani, O., Li, S., Xu, A.: Safe reinforcement learning via statistical model predictive shielding. In: Robotics: Science and Systems (2021)
9. Bastani, O., Pu, Y., Solar-Lezama, A.: Verifiable reinforcement learning via policy extraction. arXiv preprint arXiv:1805.08328 (2018)
10. Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and Regression Trees. Routledge (2017)
11. Brockman, G., et al.: OpenAI gym. arXiv preprint arXiv:1606.01540 (2016)
12. Chen, Q., Lamoreaux, A., Wang, X., Durrett, G., Bastani, O., Dillig, I.: Web question answering with neurosymbolic program synthesis. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 328–343 (2021)
13. Chen, Y., Wang, C., Bastani, O., Dillig, I., Feng, Yu.: Program synthesis using deduction-guided reinforcement learning. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 587–610. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_30
14. Collins, S., Ruina, A., Tedrake, R., Wisse, M.: Efficient bipedal robots based on passive-dynamic walkers. Science **307**(5712), 1082–1085 (2005)
15. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
16. Ellis, K., Ritchie, D., Solar-Lezama, A., Tenenbaum, J.B.: Learning to infer graphics programs from hand-drawn images. arXiv preprint arXiv:1707.09627 (2017)
17. Ellis, K., Solar-Lezama, A., Tenenbaum, J.: Unsupervised learning by program synthesis (2015)
18. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. ACM SIGPLAN Not. **50**(6), 229–239 (2015)
19. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. ACM Sigplan Not. **46**(1), 317–330 (2011)
20. Gulwani, S.: Programming by examples. Dependable Softw. Syst. Eng. **45**(137), 3–15 (2016)
21. Gulwani, S., Polozov, O., Singh, R., et al.: Program synthesis. Found. Trends® Program. Lang. **4**(1–2), 1–119 (2017)
22. He, H., Eisner, J., Daume, H.: Imitation learning by coaching. Adv. Neural. Inf. Process. Syst. **25**, 3149–3157 (2012)
23. Heess, N., Hunt, J.J., Lillicrap, T.P., Silver, D.: Memory-based control with recurrent neural networks. arXiv preprint arXiv:1512.04455 (2015)
24. Henzinger, T.A.: The theory of hybrid automata. In: Inan, M.K., Kurshan, R.P. (eds.) Verification of Digital and Hybrid Systems. NATO ASI Series, vol. 170, pp. 265–292. Springer, Berlin (2000). https://doi.org/10.1007/978-3-642-59615-5_13
25. Huang, J., Smith, C., Bastani, O., Singh, R., Albarghouthi, A., Naik, M.: Generating programmatic referring expressions via program synthesis. In: International Conference on Machine Learning, pp. 4495–4506. PMLR (2020)

26. Inala, J.P., Bastani, O., Tavares, Z., Solar-Lezama, A.: Synthesizing programmatic policies that inductively generalize. In: International Conference on Learning Representations (2020)
27. Inala, J.P., et al.: Neurosymbolic transformers for multi-agent communication. In: Neural Information Processing Systems (2020)
28. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artif. Intell. **101**(1–2), 99–134 (1998)
29. Kong, S., Gao, S., Chen, W., Clarke, E.: dReach: δ-reachability analysis for hybrid systems. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 200–205. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_15
30. Kraska, T., et al.: SageDB: a learned database system. In: CIDR (2019)
31. Levine, S., Finn, C., Darrell, T., Abbeel, P.: End-to-end training of deep visuomotor policies. J. Mach. Learn. Res. **17**(1), 1334–1373 (2016)
32. Li, S., Bastani, O.: Robust model predictive shielding for safe reinforcement learning with stochastic dynamics. In: 2020 IEEE International Conference on Robotics and Automation (ICRA), pp. 7166–7172. IEEE (2020)
33. Lillicrap, T.P., et al.: Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971 (2015)
34. Mnih, V., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
35. Pepy, R., Lambert, A., Mounier, H.: Path planning using a dynamic vehicle model. In: 2006 2nd International Conference on Information & Communication Technologies, vol. 1, pp. 781–786. IEEE (2006)
36. Puterman, M.L.: Markov decision processes. Handb. Oper. Res. Manage. Sci. **2**, 331–434 (1990)
37. Raghu, A., Komorowski, M., Celi, L.A., Szolovits, P., Ghassemi, M.: Continuous state-space models for optimal sepsis treatment: a deep reinforcement learning approach. In: Machine Learning for Healthcare Conference, pp. 147–163. PMLR (2017)
38. Ross, S., Gordon, G., Bagnell, D.: A reduction of imitation learning and structured prediction to no-regret online learning. In: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, pp. 627–635. JMLR Workshop and Conference Proceedings (2011)
39. Sadraddini, S., Shen, S., Bastani, O.: Polytopic trees for verification of learning-based controllers. In: Zamani, M., Zufferey, D. (eds.) NSV 2019. LNCS, vol. 11652, pp. 110–127. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28423-7_8
40. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. ACM SIGARCH Comput. Archit. News **41**(1), 305–316 (2013)
41. Schulman, J., Levine, S., Abbeel, P., Jordan, M., Moritz, P.: Trust region policy optimization. In: International Conference on Machine Learning, pp. 1889–1897. PMLR (2015)
42. Shah, A., Zhan, E., Sun, J.J., Verma, A., Yue, Y., Chaudhuri, S.: Learning differentiable programs with admissible neural heuristics. In: NeurIPS (2020)
43. Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
44. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (2018)
45. Sutton, R.S., McAllester, D.A., Singh, S.P., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. In: Advances in Neural Information Processing Systems, pp. 1057–1063 (2000)

46. Tian, Y., et al.: Learning to infer and execute 3D shape programs. In: International Conference on Learning Representations (2018)
47. Valkov, L., Chaudhari, D., Srivastava, A., Sutton, C., Chaudhuri, S.: HOUDINI: lifelong learning as program synthesis. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems, pp. 8701–8712 (2018)
48. Vaswani, A., et al.: Attention is all you need. In: Advances in Neural Information Processing Systems, pp. 5998–6008 (2017)
49. Verma, A., Le, H.M., Yue, Y., Chaudhuri, S.: Imitation-projected programmatic reinforcement learning. In: Neural Information Processing Systems (2019)
50. Verma, A., Murali, V., Singh, R., Kohli, P., Chaudhuri, S.: Programmatically interpretable reinforcement learning. In: International Conference on Machine Learning, pp. 5045–5054. PMLR (2018)
51. Wabersich, K.P., Zeilinger, M.N.: Linear model predictive safety certification for learning-based control. In: 2018 IEEE Conference on Decision and Control (CDC), pp. 7130–7135. IEEE (2018)
52. Wang, F., Rudin, C.: Falling rule lists. In: Artificial Intelligence and Statistics, pp. 1013–1022. PMLR (2015)
53. Young, H., Bastani, O., Naik, M.: Learning neurosymbolic generative models via program synthesis. In: International Conference on Machine Learning, pp. 7144–7153. PMLR (2019)