Programmatically Interpretable Reinforcement Learning

Abhinav Verma [1] **Vijayaraghavan Murali** [1] **Rishabh Singh** [2] **Pushmeet Kohli** [3] **Swarat Chaudhuri** [1]

Abstract

We present a reinforcement learning framework, called *Programmatically Interpretable Reinforcement Learning* (PIRL), that is designed to generate interpretable and verifiable agent policies. Unlike the popular Deep Reinforcement Learning (DRL) paradigm, which represents policies by neural networks, PIRL represents policies using a high-level, domain-specific programming language. Such programmatic policies have the benefits of being more easily interpreted than neural networks, and being amenable to verification by symbolic methods. We propose a new method, called *Neurally Directed Program Search* (NDPS), for solving the challenging nonsmooth optimization problem of finding a programmatic policy with maximal reward. NDPS works by first learning a neural policy network using DRL, and then performing a local search over programmatic policies that seeks to minimize a distance from this neural "oracle". We evaluate NDPS on the task of learning to drive a simulated car in the TORCS car-racing environment. We demonstrate that NDPS is able to discover human-readable policies that pass some significant performance bars. We also show that PIRL policies can have smoother trajectories, and can be more easily transferred to environments not encountered during training, than corresponding policies discovered by DRL.

1. Introduction

Deep reinforcement learning (DRL) has had a massive impact on the field of machine learning and has led to remarkable successes in the solution of many challenging tasks (Mnih et al., 2015; Silver et al., 2016; 2017). While neural networks have been shown to be very effective in learning good policies, the expressivity of these models makes them difficult to interpret or to be checked for consistency for some desired properties, and casts a cloud over the use of such representations in safety-critical applications.

Motivated to overcome this problem, we propose a learning framework, called *Programmatically Interpretable Reinforcement Learning* (PIRL)[1], that is based on the idea of learning policies that are represented in a human-readable language. The PIRL framework is parameterized on a high-level *programming language* for policies. A problem instance in PIRL is similar to a one in traditional RL, but also includes a *(policy) sketch* that syntactically defines a set of programmatic policies in this language. The objective is to find a program in this set with maximal long-term reward.

Intuitively, the policy programming language and the sketch characterize what we consider "interpretable". In addition to interpretability, the syntactic restriction on policies has three key benefits. First, the language can be used to implicitly encode the learner's inductive bias that will be used for generalization. Second, the language can allow effective pruning of undesired policies to make the search for a good policy more efficient. Finally, it allows us to use symbolic program verification techniques to formally reason about the learned policies and check consistency with correctness properties. At the same time, policies in PIRL can have rich semantics, for example allowing actions to depend on events far back in history.

A key technical challenge in PIRL is that the space of policies permitted in an instance can be vast and nonsmooth, making optimization extremely challenging. To address this, we propose a new algorithm called *Neurally Directed Program Synthesis* (NDPS). The algorithm first uses DRL to compute a neural policy network that has high performance, but may not be expressible in the policy language. This network is then used to direct a local search over programmatic policies. In each iteration of this search, we maintain a set of "interesting" inputs, and update the program so as to minimize the distance between its outputs and the outputs of the neural policy (an "oracle") on these inputs. The set of interesting inputs is updated as the search progresses. This strategy, inspired by imitation learning (Ross et al., 2011; Schaal, 1999), allows us to perform direct policy search in a highly nonsmooth policy space.

[1]Rice University [2]Google Brain [3]Deepmind. Correspondence to: Abhinav Verma <averma@rice.edu>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

[1]PIRL is pronounced Pi-R-L (as in π-RL)

We evaluate our approach in the task of learning to drive a simulated car in the TORCS car-racing environment (Wymann et al., 2014), as well as three classic control games (we discuss the former in the main paper, and the latter in the Appendix). Experiments demonstrate that NDPS is able to find interpretable policies that, while not as performant as the policies computed by DRL, pass some significant performance bars. Specifically, in TORCS, our policy sketch allows an unbounded set of programs with branches guarded by unknown conditions, each branch representing a Proportional-Integral-Derivative (PID) controller (Åström & Hägglund, 1995) with unknown parameters. The policy we obtain can successfully complete a lap of the race, and the use of the neural oracle is key to doing so. Our results also suggest that a well-designed sketch can serve as a regularizer. Due to constraints imposed by the sketch, the policies for TORCS that NDPS learns lead to smoother trajectories than the corresponding neural policies, and can tolerate greater noise. The policies are also more easily transferred to new domains, in particular race tracks not seen during training. Finally, we show, using several properties, that the programmatic policies that we discover are amenable to verification using off-the-shelf symbolic techniques.

2. Programmatically Interpretable Reinforcement Learning

In this section, we formalize the problem of programmatically interpretable reinforcement learning (PIRL).

We model a reinforcement learning setting as a *Partially Observable Markov Decision Process (*POMDP*)* $M = (\mathcal{S}, \mathcal{A}, \mathcal{O}, T(\cdot|s,a), Z(\cdot|s), r, Init, \gamma)$. Here, $\mathcal{S}$ is the set of (environment) states. $\mathcal{A}$ is the set of *actions* that the learning agent can perform, and $\mathcal{O}$ is the set of *observations* about the current state that the agent can make. An agent action a at the state s causes the environment state to change probabilistically, and the destination state follows the distribution $T(\cdot|s,a)$. The probability that the agent makes an observation o at state s is $Z(o|s)$. The *reward* that the agent receives on performing action a in state s is given by $r(s,a)$. *Init* is the initial distribution over environment states. Finally, $0 < \gamma < 1$ is a real constant that is used to define the agent's aggregate reward over time.

A *history* of M is a sequence $h = o_0, a_0, \ldots, a_{k-1}, o_k$, where o_i and a_i are, respectively, the agent's observation and action at the i-th time step. Let $\mathcal{H}_M$ be the set of histories in M. A *policy* is a function $\pi : \mathcal{H}_M \to \mathcal{A}$ that maps each history as above to an action a_k. For each policy, we can define a set of histories that are possible when the agent follows π. We assume a mechanism to simulate the POMDP and sample histories that are possible under a policy. The policy also induces a distribution over possible rewards R_i that the agent receives at the i-th time step. The agent's *expected aggregate reward* under π is given by $R(\pi) = \mathbf{E}[\sum_{i=0}^{\infty} \gamma^i R_i]$. The goal in reinforcement learning is to discover a policy π^* that maximizes $R(\pi)$.

A Programming Language for Policies. The distinctive feature of PIRL is that policies here are expressed in a high-level, domain-specific programming language. Such a language can be defined in many ways. However, to facilitate search through the space of programs expressible in the language, it is desirable for the language to express computations as compactly and canonically as possible. Because of this, we propose to express parameterized policies using a functional language based on a small number of side-effect-free combinators. It is known from prior work on program synthesis (Feser et al., 2015) that such languages offer natural advantages in program synthesis.

We collectively refer to observations and actions, as well as auxiliary integers and reals generated during computation, as *atoms*. Our language considers two kinds of data: atoms and sequences of atoms (including histories). We assume a finite set of *basic operators* over atoms that is rich enough to capture all common operations on observations and actions.

Figure 1 shows the syntax of this language. The nonterminals E and α represent expressions that evaluate to atoms and histories, respectively. We sketch the semantics of the various language constructs below.

- c ranges over a universe of numerical constants, and $\oplus$ is a basic operator
- $[\,]$ is the empty sequence, $\mathbf{hd}$ returns the element in an input sequence representing the most recent time point, and $\mathbf{tl}$ returns the prefix of the sequence up to (and excluding) this element. "$\mathbf{push}\ e\ a$" evaluates the atom-valued expression e, then puts the result on top of the history to which a evaluates;
- $\mathbf{map}$, $\mathbf{fold}$, $\mathbf{filter}$ are the standard higher-order combinators over sequences with the semantics:
$\mathbf{map}(f, [e_1, \ldots, e_k]) = [f(e_1), \ldots, f(e_k)]$
$\mathbf{fold}(f, [e_1, \ldots, e_k], e) = f(e_k, f(e_{k-1}, ...f(e_1, e)))$
$\mathbf{filter}(f, [e_1, \ldots, e_k]) = [e_{f_1}, \ldots, e_{f_j}]$ where for all $1 \leq i \leq j$, $f(e_{f_i})$ is true;
- x, x_1, x_2 are variables. As usual, unbound variables are assumed to be inputs.

The language comes with a type system that distinguishes between different types of atoms, and ensures that language constructs are used consistently. The type system can catch common errors, such as applying $\mathbf{hd}$ to the empty sequence. This type system identifies a set of expressions whose inputs are histories and outputs are actions. These expressions are known as *programmatic policies*, or simply *programs*.

$$E ::= c \mid x \mid \oplus(E_1, \ldots, E_k) \mid \mathbf{hd}\,(\alpha) \mid \mathbf{fold}\,((\lambda x_1, x_2.\, E_1), \alpha)$$
$$\alpha ::= x \mid [\,] \mid \mathbf{tl}\,(\alpha_1) \mid \mathbf{push}\,(E, \alpha_1) \mid \mathbf{map}\,((\lambda x.\, E), \alpha_1) \mid \mathbf{filter}\,((\lambda x.\, E), \alpha_1)$$

Figure 1. Syntax of the policy language. Here, E and α represent expressions that evaluate to atoms and histories, respectively.

Sketches. Discovering an optimal programmatic policy from the vast space of legitimate programs is typically impractical without some prior on the shape of target policies. PIRL allows the specification of such priors through instance-specific syntactic models called *sketches*.

We define a sketch as a grammar of expressions over atoms and sequences of atoms, obtained by restricting the grammar in Figure 1. The set of programs permitted by a sketch $\mathcal{S}$ is denoted by $[\![\mathcal{S}]\!]$.

PIRL. The PIRL problem can now be stated as follows. Suppose we are given a POMDP M and a sketch $\mathcal{S}$. Our goal is to find a program $e^* \in [\![\mathcal{S}]\!]$ with optimal reward:

$$e^* = \arg\max_{e \in [\![\mathcal{S}]\!]} R(e). \tag{1}$$

Example. Now we consider a concrete example of PIRL, considered in more detail in Section 5.

Suppose our goal is to make a (simulated) car complete laps on a track. We want to do so by learning policies for tasks like steering and acceleration. Suppose we know that we could get well-behaved policies by using PID control — specifically, by switching back and forth between a set of PID controllers. However, we do not know the parameters of these controllers, and neither do we know the conditions under which we should switch from one controller to another. We can express this knowledge using the following sketch:

$$\begin{aligned}
P &::= (\epsilon - \mathbf{hd}(h_i)) \\
I &::= \mathbf{fold}(+, h_i) \\
D &::= \mathbf{hd}(\mathbf{tl}(h_i)) - \mathbf{hd}(h_i) \\
C &::= c_1 + c_2 * P + c_3 * I + c_4 * D \\
B &::= c_0 + c_1 * \mathbf{hd}(h_1) + \cdots + c_k * \mathbf{hd}(h_k) > 0 \mid \\
&\qquad B_1 \textbf{ or } B_2 \mid B_1 \textbf{ and } B_2 \\
E &::= C \mid \textbf{if } B \textbf{ then } E_1 \textbf{ else } E_2.
\end{aligned}$$

Here, E represents programs permitted by the sketch. The program's input is a history h. We assume that this sequence is split into a set of sequences $\{h_1, \ldots, h_k\}$, where h_i is the sequence of observations from the i-th of k sensors. The sensor's most recent reading is given by $\mathbf{hd}(h_i)$, and its second most recent reading is $\mathbf{hd}(\mathbf{tl}(h_i))$. The operators $+$, $-$, $*$, $>$, and if-then-else are as usual. The program (optionally) evaluates a set of boolean conditions (B) over the current sensor readings, then chooses among a set of PID controllers, represented by C. In the definition of C, P is the proportional term, I is the integral term (represented as a **fold**), and D is a finite-difference approximation of the derivative term, ϵ is a known constant and represents the target for the controller. The symbols c_i are real-valued parameters.

The program in Figure 2 shows the body of a policy for acceleration that the NDPS algorithm finds given this sketch in the TORCS car racing environment. The program's input consists of histories for 29 sensors; however, only two of them, `TrackPos` and `RPM`, are actually used in the program. While the sensor `TrackPos` (for the position of the car relative to the track axis) is used to decide which controller to use, only the `RPM` sensor is needed to calculate the acceleration.

3. Neurally Directed Program Search

Imitating a Neural Policy Oracle. The NDPS algorithm is a direct policy search that is guided by a neural "oracle". Searching over policies is a standard approach in reinforcement learning. However, the nonsmoothness of the space of programmatic policies poses a fundamental challenge to the use of such an approach in PIRL. For example, a conceivable way of solving the search problem would be to define a neighborhood relation over programs and perform local search. However, in practice, the objective $R(e)$ of such a search can vary irregularly, leading to poor performance (see Section 5 for experimental results on this).

In contrast, NDPS starts by using DRL to compute a neural policy oracle $e_{\mathcal{N}}$ for the given environment. This policy is an approximation of the programmatic policy that we seek to find. To a first approximation, NDPS is a local search over programmatic policies that seeks to find a program e^* that closely imitates the behavior of $e_{\mathcal{N}}$. The main intuition here is that distance from $e_{\mathcal{N}}$ is a simpler objective than the reward function $R(e)$, which aggregates rewards over a lengthy time horizon. This approach can be seen to be a form of imitation learning (Schaal, 1999).

The *distance* between $e_{\mathcal{N}}$ and the estimate e of e^* in a search iteration is defined as $d(e_{\mathcal{N}}, e) = \sum_{h \in \mathcal{H}} \|e(h) - e_{\mathcal{N}}(h)\|$, where $\mathcal{H}$ is a set of "interesting" inputs (histories) and $\|\cdot\|$ is a suitable norm. During the iteration, we search the neighborhood of e for a program e' that minimizes this distance. At the end of the iteration, e' becomes the new estimate for e^*.

Input Augmentation. One challenge in the algorithm is that under the policy e, the agent may encounter histories

$$\begin{array}{l}\textbf{if } (0.001 - \textbf{hd}(h_{\texttt{TrackPos}}) > 0) \textbf{ and } (0.001 + \textbf{hd}(h_{\texttt{TrackPos}}) > 0) \\ \quad \textbf{then } 1.96 + 4.92 * (0.44 - \textbf{hd}(h_{\texttt{RPM}})) + 0.89 * \textbf{fold}(+, h_{\texttt{RPM}}) + 49.79 * (\textbf{hd}(\textbf{tl}(h_{\texttt{RPM}})) - \textbf{hd}(h_{\texttt{RPM}})) \\ \quad \textbf{else } \ 1.78 + 4.92 * (0.40 - \textbf{hd}(h_{\texttt{RPM}})) + 0.89 * \textbf{fold}(+, h_{\texttt{RPM}}) + 49.79 * (\textbf{hd}(\textbf{tl}(h_{\texttt{RPM}})) - \textbf{hd}(h_{\texttt{RPM}}))\end{array}$$

Figure 2. A programmatic policy for acceleration, automatically discovered by the NDPS algorithm. $h_{\texttt{RPM}}$ and $h_{\texttt{TrackPos}}$ represent histories for the `RPM` and `TrackPos` sensors, respectively.

that are not possible under $e_{\mathcal{N}}$, or any of the programs encountered in previous iterations of the search. For example, while searching for a steering controller, we may arrive at a program that, under certain conditions, steers the car into a wall, an illegal behavior that the neural policy does not exhibit. Such histories would be irrelevant to the distance between $e_{\mathcal{N}}$ and e if the set $\mathcal{H}$ were constructed ahead of time by simulating $e_{\mathcal{N}}$, and never updated. This would be unfortunate as these are precisely the inputs on which the programmatic policy needs guidance.

Our solution to this problem is *input augmentation*, or periodic updates to the set $\mathcal{H}$. More precisely, after a certain number of search steps for a fixed set $\mathcal{H}$, and after choosing the best available synthesized program for this set, we sample a set of additional histories by simulating the current programmatic policy, and add these samples to $\mathcal{H}$.

3.1. Algorithm Details

Algorithm 1 Neurally Directed Program Search

Input: POMDP M, neural policy $e_{\mathcal{N}}$, sketch $\mathcal{S}$
$\mathcal{H} \leftarrow$ `create_histories`$(e_{\mathcal{N}}, M)$
$e \leftarrow$ `initialize`$(e_{\mathcal{N}}, \mathcal{H}, M, \mathcal{S})$
$R \leftarrow$ `collect_reward`(e, M)
repeat
  $(e', R') \leftarrow (e, R)$
  $\mathcal{H} \leftarrow$ `update_histories`$(e, e_{\mathcal{N}}, M, \mathcal{H})$
  $\mathcal{E} \leftarrow$ `neighborhood_pool`(e)
  $e \leftarrow \arg\min_{e' \in \mathcal{E}} \sum_{h \in \mathcal{H}} \|e'(h) - e_{\mathcal{N}}(h)\|$
  $R \leftarrow$ `collect_reward`(e, M)
until $R' \geq R$
Output: e'

We show pseudocode for NDPS in Algorithm 1. The inputs to the algorithm are a POMDP M, a neural policy $e_{\mathcal{N}}$ for M that serves as an oracle, and a sketch $\mathcal{S}$. The algorithm first samples a set of histories of $e_{\mathcal{N}}$ using the procedure `create_histories`. Next it uses the routine `initialize` to generate the program that is the starting point of the policy search. Then the procedure `collect_reward` calculates the expected aggregate reward $R(e)$ (described in Section 2), by simulating the program in the POMDP.

From this point on, NDPS iteratively updates its estimate e of the target program, as well as its estimate $\mathcal{H}$ of the set of interesting inputs used for distance computation. To do the former, NDPS uses the procedure `neighborhood_pool` to generate a space of programs that are structurally similar to e, then finds the program in this space that minimizes distance from $e_{\mathcal{N}}$. The latter task is done by the routine `update_histories`, which heuristically picks interesting inputs in the trajectory of the learned program and then obtains the corresponding actions from the oracle for those inputs. This process goes on until the iterative search fails to improve the estimated reward R of e.

The subroutines used in the above description can be implemented in many ways. Now we elaborate on our implementation of the important subroutines of NDPS.

The optimization step. The search for a program e' at minimal distance from the neural oracle can be implemented in many ways. The approach we use has two steps. First, we enumerate a set of *program templates* — numerically parameterized programs — that are structurally similar to e and are permitted by the sketch $\mathcal{S}$, giving priority to shorter templates. Next, we find optimal parameters for the enumerated templates. Our primary tool for the second step is Bayesian optimization (Snoek et al., 2012), though we also explored a symbolic optimization technique based on Satisfiability Modulo Theories (SMT) solving (Appendix B).

The initialization step. The performance of NDPS turns out to be quite sensitive to the choice of the program that is the starting point of the search. Our initialization routine `initialize` is broadly similar to the optimization step, in that it attempts to find programs that closely imitate the oracle through a combination of template enumeration and parameter optimization. However, rather than settling on a single program, `initialize` generates a pool of programs that are close in behavior to the oracle. After this, it simulates the programs in the POMDP and returns the program that achieves the highest reward.

4. Environments for Experiments

In this section, we describe the environments (modeled by POMDPs) on which we evaluated the NDPS algorithm.

TORCS. We use NDPS to generate controllers for cars in *The Open Racing Car Simulator* (TORCS) (Wymann et al., 2014). TORCS has been used extensively in AI research, for example in (Salem et al., 2017), (Koutník et al., 2013), and (Loiacono et al., 2010) among others. (Lillicrap et al., 2015a) has shown that a Deep Deterministic Policy Gradient (DDPG) network can be used in RL environments with continuous action spaces. The DRL agents for TORCS in this paper implement this approach.

In its full generality TORCS provides a rich environment with input from up to 89 sensors, and optionally the 3D graphic from a chosen camera angle in the race. The controllers have to decide the values of 5 parameters during game play, which correspond to the acceleration, brake, clutch, gear and steering of the car. Apart from the immediate challenge of driving the car on the track, controllers also have to make race-level strategy decisions, like making pit-stops for fuel. A lower level of complexity is provided in the *Practice Mode* setting of TORCS. In this mode all race-level strategies are removed. Currently, so far as we know, state-of-the-art DRL models are capable of racing only in *Practice Mode*, and this is also the environment that use. Here we consider the input from 29 sensors, and decide values for the acceleration and steering actions.

The sketches used in our experiments are as in the example in Section 2, and provide the basic structure of a proportional-integral-derivative (PID) program, with appropriate holes for parameter and observation values. To obtain a practical implementation, we constrain the fold calculation to the five latest observations of the history. This constraint corresponds to the standard strategy of automatic (integral) error reset in discretized PID controllers (Astrom & Hagglund, 1984).

Each track in TORCS can we viewed as a distinct POMDP. In our implementation of NDPS for TORCS we choose one track and synthesize a program for it. Whenever the algorithm needs to interact with the POMDP, we use the program or DRL agent to race on the track. For example, in the procedure `collect_reward` we use the synthesized program to race one lap, and the reward is a function of the speed, angle and position of the car at each time step.

For the `create_histories` procedure we use the DRL agent to complete one lap of the track (an *episode*), recording the sensor values and environment state at each time step. The `update_histories` procedure uses a two step process. First, the synthesized program is used to race one lap and we store the sequence of observations (given by sensor values) $o_1, o_2, \ldots$ provided by TORCS during this lap. Then, we use the DRL agent to generate the corresponding action a_i for each observation o_i, and sample an observation o'_i from the resulting state. Each triple (o_i, a_i, o'_i) is then added to the set of histories.

Classic Control Games. In addition to TORCS, we evaluated our approach in three classic control games, *Acrobot*, *CartPole*, and *MountainCar*. These games provide simpler RL environments, with fewer input sensors than TORCS and only a single discrete action at each time step, compared to two continuous actions in TORCS. These results appear in Appendix A.

5. Experimental Analysis

Now we present an empirical evaluation of the effectiveness of our algorithm in solving the PIRL problem. We synthesize programs for two TORCS tracks, CG-Speedway-1 and Aalborg. These tracks provide varying levels of difficulty, with Aalborg being the more difficult track of the two.

5.1. Evaluating Performance

A controller's performance is measured according to two metrics, *lap time* and *reward*. To calculate the lap time, the programs are allowed to complete a three lap race, and we report the average time taken to complete a lap during this race. The reward function is calculated using the car's velocity, angle with the track axis, and distance from the track axis. The same function is used to train the DRL agent initially. In the experiments we compare the average reward per time step, obtained by the various programs.

We compare among the following RL agents:

A1: DRL. An agent which uses DRL to find a policy represented as a deep neural network. The specific DRL algorithm we use is Deep Deterministic Policy Gradients (Lillicrap et al., 2015b), which has previously been used on TORCS.

A2: *Naive*. Program synthesized without access to a policy oracle.

A3: *NoAug*. Program synthesized without input augmentation.

A4: *NoSketch*. Program synthesized in our policy language without sketch guidance.

A5: *NoIF*. Programs permitted by a restriction of our sketch that does not permit conditional branching.

A6: NDPS. The Program generated by the NDPS algorithm.

In Table 1 we present the performance results of the above list. The lap times in that table are given in minutes and seconds. The TIMEOUT entries indicate that the synthesis process did not return a program that could complete the race, within the specified timeout of twelve hours.

These results justify the various choices that we made in our NDPS algorithm architecture, as discussed in Section 3. In many cases those choices were necessary to be able to synthesize a program that could successfully complete a race. As a consequence of these results, we only consider the DRL agent and the NDPS program for subsequent comparisons.

The *NoAug* and *NoSketch* agents are unable to generate programs that complete a single lap on either track. In the case of *NoSketch* this is because the syntax of the policy language (Figure 1), defines a very large program space. If we randomly sample from this space without any constraints (like those provided by the sketch), then the probability of getting a good program is extremely low and hence we are unable to reliably generate a program that can complete a lap. The *NoAug* agent performs poorly because without input augmentation, the synthesizer has no guidance from the oracle regarding the “correct” behavior once the program deviates even slightly from the oracle’s trajectory.

Table 1. Performance results in TORCS. Lap time is given in Minutes:Seconds. Timeout indicates that the synthesizer did not return a program that completed the race within the specified timeout.

MODEL	CG-SPEEDWAY-1		AALBORG	
	LAP TIME	REWARD	LAP TIME	REWARD
DRL	54.27	118.39	1:49.66	71.23
Naive	2:07.09	58.72	TIMEOUT	–
NoAug	TIMEOUT	–	TIMEOUT	–
NoSketch	TIMEOUT	–	TIMEOUT	–
NoIF	1:01.60	115.25	2:45.13	52.81
NDPS	1:01.56	115.32	2:38.87	54.91

5.2. Qualitative Analysis of the Programmatic Policy

We provide qualitative analysis of the inferred programmatic policy through the lens of interpretability, and its behavior in acting in the environment.

Interpretability. Interpretability is a qualitative metric, and cannot be easily demonstrated via experiments. The DRL policies are considered uninterpretable because their policies are encoded in black box neural networks. In contrast, the PIRL policies are compact and human-readable by construction, as exemplified by the acceleration policy in Figure 2. More examples of our synthesized policies are given in Appendix C.

Behavior of Policy. Our experimental validation showed that the programmatic policy was less aggressive in terms of its use of actions and resulting in smoother steering actions. Numerically, we measure smoothness in Table 2 by comparing the population standard deviation of the set of steering actions taken by the program during the entire race. In Figure 3 we present a scatter plot of the steering actions taken by the DRL agent and the NDPS program during a slice of the CG-Speedway-1 race. As we can see, the NDPS program takes much more conservative actions.

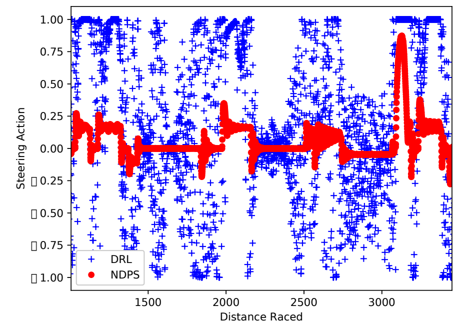

Figure 3. Slice of steering actions taken by the DRL and NDPS agents, during the CG-Speedway-1 race. This figure demonstrates that the NDPS agent drives more smoothly.

Table 2. Smoothness measure of agents in TORCS, given by the standard deviation of the steering actions during a complete race. Lower values indicate smoother steering.

MODEL	CG-SPEEDWAY-1	AALBORG
DRL	0.5981	0.9008
NDPS	0.1312	0.2483

5.3. Robustness to Missing/Noisy Features

To evaluate the robustness of the agents with respect to defective sensors we introduce a *Partial Observability* variant of TORCS. In this variant, a random sample of k sensors are declared defective. During the race, one or more of these defective sensors are blocked with some fixed probability. Hence, during game-play, the sensor either returns the correct reading or a *null* reading. For sufficiently high block probabilities, both agents will fail to complete the race. In Table 3 we show the distances raced for two values of the block probability, and in Figure 4 we plot the distance raced as we increase the block probability on the Aalborg track. In both these experiments, the set of defective sensors was taken to be $\{\texttt{RPM}, \texttt{TrackPos}\}$ because we know that the synthesized programs crucially depend on these sensors.

Table 3. Partial observability results in TORCS blocking sensors $\{\texttt{RPM}, \texttt{TrackPos}\}$. For each track and block probability we give the distance, in meters, raced by the program before crashing.

MODEL	CG-SPEEDWAY-1		AALBORG	
	50%	90%	50%	90%
DRL	21	17	71	20
NDPS	1976	200	1477	287

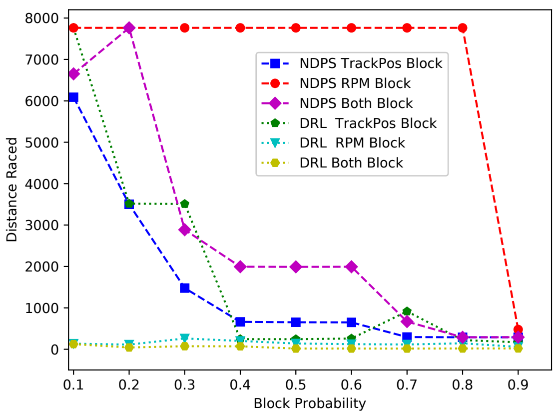

Figure 4. Distance raced by the agents as the block probability increases for a particular sensor(s) on Aalborg. The NDPS agent is more robust to blocked sensors.

5.4. Evaluating Generalization to New Instances

To compare the ability of the agents to perform on unseen tracks, we executed the learned policies on tracks of comparable difficulty. For agents trained on the CG-Speedway-1 track, we chose CG track 2 and E-Road as the transfer tracks, and for Aalborg trained tracks we chose Alpine 2 and Ruudskogen. As can be seen in Tables 4 and 5, the NDPS programmatically synthesized program far outperforms the DRL agent on unseen tracks. The DRL agent is unable to complete the race on any of these transfer tracks. This demonstrates the transferability of the policies NDPS finds.

Table 4. Transfer results with training on CG-Speedway-1. 'Cr' indicates that the agent crashed after racing the specified distance.

MODEL	CG TRACK 2		E-ROAD	
	LAP TIME	REWARD	LAP TIME	REWARD
DRL	CR 1608M	–	CR 1902M	–
NDPS	1:40.57	110.18	1:51.59	98.21

Table 5. Transfer results with training on Aalborg. 'Cr' denotes the agent crashed, after racing the specified distance.

MODEL	ALPINE 2		RUUDSKOGEN	
	LAP TIME	REWARD	LAP TIME	REWARD
DRL	CR 1688M	–	CR 3232M	–
NDPS	3:16.68	67.49	3:19.77	57.69

5.5. Verifiability of Policies

Now we use established symbolic verification techniques to automatically prove two properties of policies generated by NDPS. So far as we know, the current state of the art neural network verifiers cannot verify the DRL network we are using in a reasonable amount of time, due to the size and complexity of the network used to implement the DDPG algorithm. For example, the Reluplex (Katz et al., 2017) algorithm was tested on networks at most 300 nodes wide, whereas our network has three layers with 600 nodes each, and other smaller layers.

Smoothness Property For the program given in Figure 2 we proved, we have $\forall k \ \sum_{i=k-5}^{k} \|\texttt{RPM}_{i+1} - \texttt{RPM}_i\| < 0.006 \implies \|Acceleration_{k+1} - Acceleration_k\| < 0.49$. Intuitively, this means that if the sum of the consecutive differences of the last six RPM sensor values is less than 0.006, then the acceleration actions calculated at the last and penultimate step will not differ by more than 0.49. Similarly, for a policy given in Appendix C, we prove $\forall k \ \sum_{i=k-5}^{k} \|\texttt{TrackPos}_{i+1} - \texttt{TrackPos}_i\| < 0.006 \implies \|Steering_{k+1} - Steering_k\| < 0.11$. This proof gives us a guarantee of the type of smooth steering behavior that we empirically examined earlier in this section.

Universal Bounds We can prove that the program in Figure 2 satisfies the property $\forall i \ (0 \leq \texttt{RPM}_i \leq 1 \wedge -1 \leq \texttt{TrackPos}_i \leq 1) \implies (\|Steering_i\| < 101.08 \wedge -54.53 < Acceleration_i < 53.03)$. Intuitively, this means that we have proved global bounds for the action values in this environment, assuming reasonable bounds on some of the input values. In the TORCS environment these bounds are not very useful, since the simulator clips these actions to certain pre-specified ranges. However, this experiment demonstrates that our framework allows us to prove universal bounds on the actions, and this could be a critical property for other environments.

6. Related Work

Syntax-Guided Synthesis. The original formulation of inductive program synthesis is to search for a program in a hypothesis space (programming language) that is consistent with a specification (such as IO examples). However, this search is often intractable because of the large (potentially infinite) hypothesis space. One of the key ideas to make this search tractable is to provide the synthesizer a sketch of the desired program in addition to the examples, for example in (Solar-Lezama, 2009) and (Feser et al., 2015). The program sketch in addition to providing structure to the search space also allows users to provide additional insights. This approach has been generalized in a framework called Syntax-Guided Synthesis (SYGUS) (Alur et al., 2015). Our PIRL approach is inspired by SYGUS in the sense that we also use a high-level grammar to constrain the shape of the possible learnt policies in a policy language grammar. However, unlike SYGUS and previous sketch-based synthesis approaches that use logical constraints as specification, PIRL searches for policies with quantitative objectives.

Imitation Learning. Imitation learning (Schaal, 1999) has been a successful paradigm for reducing the sample complexity of reinforcement learning algorithms by allowing the agent to leverage the additional supervision provided in terms of expert demonstrations for the desired behaviors. The DAGGER (Dataset Aggregation) algorithm (Ross et al., 2011) is an iterative algorithm for imitation learning that learns stationary deterministic policies, where in each iteration i it uses the current learnt policy π_i to collect new trajectories and adds them to the dataset D of all previously found trajectories. The policy for the next iteration π_{i+1} is a policy that best mimics the expert policy π^* on the whole dataset D. Our Neurally Directed Program Search (NDPS) is inspired by the DAGGER algorithm, where we use the trained DeepRL agent as the expert (oracle), and iteratively perform IO augmentation for unseen input states explored by our synthesized policy with the current best reward. However, one key difference is that NDPS uses the expert trajectories to only guide the local program search in our policy language grammar to find a policy with highest rewards, unlike the imitation learning setting where the goal is to match the expert demonstrations perfectly.

Neural Program Synthesis and Induction. Many recent efforts use neural networks for learning programs. These efforts have two flavors. In *neural program induction*, the goal is to learn a network that encodes the program semantics using internal weights. These architectures typically augment neural networks with differentiable computational substrates such as memory (Neural Turing Machines (Graves et al., 2014)), modules (Neural RAM (Kurach et al., 2015)) or data-structures such as stacks (Joulin & Mikolov, 2015), and formulate the program learning problem in an end-to-end differentiable manner. In *neural program synthesis*, the architectures generate programs directly as outputs using multi-task transfer learning (e.g. ROBUSTFILL (Devlin et al., 2017), DEEPCODER (Balog et al., 2016), BAYOU (Murali et al., 2018)), where the network weights are used to guide the program search in a DSL. There have also been some recent approaches to use RL for learning to search programs in DSLs (Bunel et al., 2018; Abolafia et al., 2018). Our approach falls in the category of program synthesis approaches where we synthesize policies in a policy language. However, we learn richer policy programs with continuous parameters using the NDPS algorithm.

Interpretable Machine Learning. Many recent efforts in deep learning aim to make deep networks more interpretable (Montavon et al., 2017; Lipton, 2016; Garnelo et al., 2016; Zahavy et al., 2016; Shanahan, 2005; Lake et al., 2016). There are three key approaches explored for interpreting DNNs: i) generate input prototypes in the input domain that are representatives of the learned concept in the abstract domain of the top-level of a DNN, ii) explaining DNN decisions by relevance propagation and computing corresponding representative concepts in the input domain, and iii) Using symbolic techniques to explain and interpret a DNN. Our work differs from these approaches in that we are replacing the DRL model with human readable source code, that is programmatically synthesized to mimic the policy found by the neural network. Working at this level of abstraction provides a method to apply existing synthesis techniques to the problem of making DRL models interpretable.

Verification of Deep Neural Networks. Reluplex (Katz et al., 2017) is an SMT solver that supports linear real arithmetic with ReLU constraints, and has been used to verify several properties of DNN-based airborne collision avoidance systems, such as not producing erroneous alerts and uniformity of alert regions. Unlike Reluplex, our framework generates interpretable program source code as output, where we can use traditional symbolic program verification techniques (King, 1976) to prove program properties.

7. Conclusion

We have introduced a framework for interpretable reinforcement learning, called PIRL. Here, policies are represented in a high-level language. The goal is to find a policy that fits a syntactic "sketch" and also has optimal long-term reward. We have given an algorithm inspired by imitation learning, called NDPS, to achieve this goal. Our results show that the method is able to generate interpretable policies that clear reasonable performance goals, are amenable to symbolic verification, and, assuming a well-designed sketch, are robust and easily transferred to unseen environments.

The experiments in this paper only considered environments with symbolic inputs. Handling perceptual inputs may raise additional algorithmic challenges, and is a natural next step. Also, in this paper, we only considered deterministic (if memoryful) policies. Extending our framework to stochastic policies is a goal for future work. Finally, while we explored policies in the context of reinforcement learning, one could define similar frameworks for other learning settings.

Acknowledgements

This research was partially supported by NSF Award CCF-1162076 and DARPA MUSE Award #FA8750-14-2-0270.

References

Abolafia, D. A., Norouzi, M., and Le, Q. V. Neural program synthesis with priority queue training. *CoRR*, abs/1801.03526, 2018.

Alur, R., Bodík, R., Dallal, E., Fisman, D., Garg, P., Juniwal, G., Kress-Gazit, H., Madhusudan, P., Martin, M. M. K., Raghothaman, M., Saha, S., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. Syntax-guided synthesis. In *Dependable Software Systems Engineering*, pp. 1–25. 2015.

Astrom, K. and Hagglund, T. Automatic tuning of simple regulators with specifications on phase and amplitude margins. *Automatica*, 20(5):645 – 651, 1984. ISSN 0005-1098.

Åström, K. J. and Hägglund, T. *PID controllers: Theory, Design, and Tuning*. Instrument society of America, 1995.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. *CoRR*, abs/1611.01989, 2016.

Bunel, R., Hausknecht, M., Devlin, J., Singh, R., and Kohli, P. Leveraging grammar and reinforcement learning for neural program synthesis. In *ICLR*, 2018.

Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A., and Kohli, P. Robustfill: Neural program learning under noisy I/O. *CoRR*, abs/1703.07469, 2017.

Feser, J. K., Chaudhuri, S., and Dillig, I. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pp. 229–239, 2015.

Garnelo, M., Arulkumaran, K., and Shanahan, M. Towards deep symbolic reinforcement learning. *CoRR*, abs/1609.05518, 2016.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

Joulin, A. and Mikolov, T. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pp. 190–198, 2015.

Katz, G., Barrett, C. W., Dill, D. L., Julian, K., and Kochenderfer, M. J. Reluplex: An efficient smt solver for verifying deep neural networks. In *Computer Aided Verification*, pp. 97–117, 2017.

King, J. C. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

Koutník, J., Cuccu, G., Schmidhuber, J., and Gomez, F. J. Evolving large-scale neural networks for vision-based TORCS. In *Proceedings of the 8th International Conference on the Foundations of Digital Games, FDG 2013, Chania, Crete, Greece, May 14-17, 2013.*, pp. 206–212, 2013.

Kurach, K., Andrychowicz, M., and Sutskever, I. Neural random-access machines. *CoRR*, abs/1511.06392, 2015.

Lake, B. M., Ullman, T. D., Tenenbaum, J. B., and Gershman, S. J. Building machines that learn and think like people. *CoRR*, abs/1604.00289, 2016.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015a.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015b.

Lipton, Z. C. The mythos of model interpretability. *CoRR*, abs/1606.03490, 2016.

Loiacono, D., Prete, A., Lanzi, P. L., and Cardamone, L. Learning to overtake in TORCS using simple reinforcement learning. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010, Barcelona, Spain, 18-23 July 2010*, pp. 1–8, 2010.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529, 2015.

Montavon, G., Samek, W., and Müller, K. Methods for interpreting and understanding deep neural networks. *CoRR*, abs/1706.07979, 2017.

Murali, V., Chaudhuri, S., and Jermaine, C. Neural sketch learning for conditional program generation. In *ICLR*, 2018.

Ross, S., Gordon, G. J., and 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, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, pp. 627–635, 2011.

Salem, M., Mora, A. M., Merelo, J. J., and García-Sánchez, P. Driving in TORCS using modular fuzzy controllers. In *Applications of Evolutionary Computation - 20th European Conference, EvoApplications 2017, Amsterdam,* *The Netherlands, April 19-21, 2017, Proceedings, Part I*, pp. 361–376, 2017.

Schaal, S. Is imitation learning the route to humanoid robots? *Trends in cognitive sciences*, 3(6):233–242, 1999.

Shanahan, M. Perception as abduction: Turning sensor data into meaningful representation. *Cognitive Science*, 29(1): 103–134, 2005. ISSN 1551-6709.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T. P., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. Mastering chess and Shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

Snoek, J., Larochelle, H., and Adams, R. P. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, pp. 2951–2959, USA, 2012. Curran Associates Inc.

Solar-Lezama, A. The sketching approach to program synthesis. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, pp. 4–13, 2009.

Wymann, B., Espié, E., Guionneau, C., Dimitrakakis, C., Coulom, R., and Sumner, A. TORCS, The Open Racing Car Simulator. http://www.torcs.org, 2014.

Zahavy, T., Ben-Zrihem, N., and Mannor, S. Graying the black box: Understanding dqns. *CoRR*, abs/1602.02658, 2016.