Appendix: Programmatically Interpretable Reinforcement Learning

A. Evaluation on Classic Control Games

In this section, we provide results of additional experimental evaluation on some classic control games. We use the OpenAI Gym environment implementation of these games. A brief description of these games is given below.

We used the DUEL-DDQN algorithm (Wang et al., 2015) to obtain our neural policy oracle for these games, rather than DDPG, as an implementation of Duel-DDQN already appears on the OpenAI Gym leader-board.

Table 6. Rewards achieved in Classic Control Games. Acrobot does not have threshold at which it is considered solved.

	ACROBOT	CARTPOLE	MOUNTAINCAR
SOLVED	–	195	-110
DRL	-63.17	197.53	-84.73
NDPS-SMT	-84.16	183.15	-108.06
NDPS-BOPT	-127.21	143.21	-143.86
MINIMUM	-200	8	-200

Acrobot. This environment consists of a two link, two joint robot. The joint between the links is actuated. At the start of the episode, the links are hanging downwards. At every timestep the agent chooses an action that correspond to applying a force to move the actuated link to the right, to the left, or to not applying a force. The episode is over once the end of the lower link swings above a certain height. The goal is to end the episode in the fewest possible timesteps.

We use the OpenAI Gym 'Acrobot-v1' environment. This implementation is based on the system presented in (Geramifard et al., 2015). Each observation is a set consisting of readings from six sensors, corresponding to the rotational joint angles and velocities of joints and links. The action space is discrete with three elements, and at each timestep the environment returns the observation and a reward of -1. An episode is terminated after 200 time steps irrespective of the state of the robot. This is an unsolved environment, which means it does not have a specified reward threshold at which it's considered solved.

CartPole. This environment consists of a pole attached by an un-actuated joint to a cart that moves along a frictionless track. At the beginning, the pole is balanced vertically on the cart. The episode ends when the pole is more than 15° from vertical, or the cart moves more than 2.4 units from the center. At every timestep the agent chooses to apply a force to move the cart to the right or to the left, and the goal is to prevent an episode from ending for the maximum possible timesteps.

We use the OpenAI Gym 'CartPole-v0' environment, based on the system presented in (Barto et al., 1983). The sensor values correspond to the cart position, cart velocity, pole angle and pole velocity. The action space is discrete with two elements, and at each timestep the environment returns the observation and a reward of $+1$. An episode is terminated after 200 time steps irrespective of the state of the cart. CartPole-v0 defines "solving" as getting an average reward of at least 195.0 over 100 consecutive trials.

MountainCar. This environment consists of an underpowered car on a one-dimensional track. At the beginning, the car is placed between two 'hills'. The episode ends when the car reaches the top of the hill in front of it. Since the car is underpowered, the agent needs to drive it back and forth to build momentum. At every timestep the agent chooses to apply a force to move the car to the right, to the left, or to not apply a force. The goal is to end the episode in the fewest possible timesteps.

We use the OpenAI Gym 'MountainCar-v0' environment. This implementation is based on the system presented in (Moore, 1991). The sensors provide the position and velocity of the car. The action space is discrete with three elements, and at each timestep the environment returns the observation and a reward of -1. An episode is terminated after 200 time steps irrespective of the state of the robot. MountainCar-v0 is considered "solved" if the average reward over 100 consecutive trials is not less than -110.0.

Results. Table 6 shows rewards obtained by optimal policies found using various methods in these environments. The first row gives numbers for the DRL method. The rows NDPS-SMT and NDPS-BOPT for versions of the NDPS algorithm that respectively use SMT-based optimization and Bayesian optimization to find template parameters (more on this below).

B. Additional Details on Algorithm

Now we elaborate on the optimization techniques we used in the distance computation step $\arg\min_{e'} \sum_{h\in\mathcal{H}} \|e'(h) - e_{\mathcal{N}}(h)\|$, to find a program similar to a given program e, in Algorithm 1.

$$0.97 * (0.0 - \mathbf{hd}(h_{\texttt{TrackPos}})) + 0.05 * \mathbf{fold}(+, h_{\texttt{TrackPos}}) + 49.98 * (\mathbf{hd}(\mathbf{tl}(h_{\texttt{TrackPos}})) - \mathbf{hd}(h_{\texttt{TrackPos}}))$$

Figure 5. A programmatic policy for steering, automatically discovered by the NDPS algorithm with training on Aalborg.

$$\begin{array}{l}\textbf{if } (0.0001 - \mathbf{hd}(h_{\texttt{TrackPos}}) > 0) \textbf{ and } (0.0001 + \mathbf{hd}(h_{\texttt{TrackPos}}) > 0) \\ \quad \textbf{then } 2.02 + 0.95 * (0.64 - \mathbf{hd}(h_{\texttt{RPM}})) + 0.63 * \mathbf{fold}(+, h_{\texttt{RPM}}) + 3.89 * (\mathbf{hd}(\mathbf{tl}(h_{\texttt{RPM}})) - \mathbf{hd}(h_{\texttt{RPM}})) \\ \quad \textbf{else } 1.89 + 0.95 * (0.60 - \mathbf{hd}(h_{\texttt{RPM}})) + 0.63 * \mathbf{fold}(+, h_{\texttt{RPM}}) + 3.89 * (\mathbf{hd}(\mathbf{tl}(h_{\texttt{RPM}})) - \mathbf{hd}(h_{\texttt{RPM}}))\end{array}$$

Figure 6. A programmatic policy for acceleration, automatically discovered by the NDPS algorithm with training on CG-Speedway-1.

$$2.76 * (0.0 - \mathbf{hd}(h_{\texttt{TrackPos}})) + 0.69 * \mathbf{fold}(+, h_{\texttt{TrackPos}}) + 46.51 * (\mathbf{hd}(\mathbf{tl}(h_{\texttt{TrackPos}})) - \mathbf{hd}(h_{\texttt{TrackPos}}))$$

Figure 7. A programmatic policy for steering, automatically discovered by the NDPS algorithm with training on CG-Speedway-1.

As mentioned in the main paper, we start by enumerating a list of *program templates*, or programs with numerical-valued parameters θ. This is done by first replacing the numerical constants in e by parameters, eliding some subexpressions from the resulting parameterized program, and then regenerating the subexpressions using the rules of $\mathcal{S}$ (without instantiating the parameters), giving priority to shorter expressions. The resulting program template e_θ follows the sketch $\mathcal{S}$ and is also structurally close to e. Now we search for values for parameters θ that optimally imitate the neural oracle.

Bayesian optimization. We use Bayesian optimization as our primary tool when searching for such optimal parameter values. This method applies to problems in which actions (program outputs) can be represented as vectors of real numbers. All problems considered in our experiments fall in this category. The distance of individual pairs of outputs of the synthesized program and the policy oracle is then simply the Euclidean distance between them. The sum of these distances is used to define the aggregate cost across all inputs in $\mathcal{H}$. We then use Bayesian optimization to find parameters that minimize this cost.

SMT-based Optimization. We also use a second parameter search technique based on SMT (Satisfiability Modulo Theories) solving. Here, we generate a constraint that stipulates that for each $h \in \mathcal{H}$, the output $e_\theta(h)$ must match $e_\mathcal{N}(h)$ up to a constant error. Here, $e_\mathcal{N}(h)$ is a constant value obtained by executing $e_\mathcal{N}$. The output $e_\theta(h)$ depends on unknown parameters θ; however, constraints over $e_\theta(h)$ can be represented as constraints over θ using techniques for *symbolic execution* of programs (Cadar & Sen, 2013). Because the oracle is only an approximation to the optimal policy in our setting, we do not insist that the generated constraint is satisfied entirely. Instead, we set up a Max-Sat problem which assigns a weight to the constraint for each input h, and then solve this problem with a Max-Sat solver.

$$\begin{array}{l}\textbf{if } (0.1357 + \mathbf{hd}(h_4)) < 0 \\ \quad \textbf{then } 2 \\ \quad \textbf{else } 0\end{array}$$

Figure 8. A programmatic policy for Acrobot, automatically discovered by the NDPS algorithm.

$$\begin{array}{l}\textbf{if } (\mathbf{fold}(+, h_0) - \mathbf{hd}(h_3)) > 0 \\ \quad \textbf{then } 0 \\ \quad \textbf{else } 1\end{array}$$

Figure 9. A programmatic policy for CartPole, automatically discovered by the NDPS algorithm.

$$\begin{array}{l}\textbf{if } (0.2498 - \mathbf{hd}(h_0) > 0) \textbf{ and } (0.0035 - \mathbf{hd}(h_1) < 0) \\ \quad \textbf{then } 0 \\ \quad \textbf{else } 2\end{array}$$

Figure 10. A programmatic policy for MountainCar, automatically discovered by the NDPS algorithm.

Unfortunately, SMT-based optimization does not scale well in environments with continuous actions. Consequently, we exclusively use Bayesian optimization for all TORCS based experiments. SMT-based optimization can be used in the classic control games, however, and Table 6 shows results generated using this technique (in row NDPS-SMT).

The results in Table 6 show that for the classic control games, SMT-based optimization gives better results. This is because the small number of legal actions in these games, limited to at most three values $\{0, 1, 2\}$, are well suited for the SMT setting. The SMT solver is able to efficiently perform parameter optimization, with a small set of histories. Whereas, the limited variability in actions forces the Bayesian optimization method to use a larger set of histories, and makes it harder for the method to avoid getting trapped in local

minimas.

C. Policy Examples

In this section we present more examples of the policies found by the NDPS algorithm.

The program in Figure 5 shows the body of a policy for steering, which together with the acceleration policy given in the paper (Figure 2), was found by the NDPS algorithm by training on the Aalborg track. Figures 6 & 7 likewise show the policies for acceleration and steering respectively, when trained on the CG-Speedway-1 track. Similarly, Figures 8, 9 & 10 show policies found for Acrobot, CartPole, and MountainCar respectively. Here h_i is the sequence of observations from the i-th of k sensors, for example h_0 is the 0-th sensor. The sensor order is determined by the OpenAI simulator.

D. TORCS Video

We provide a video at the following link, which depicts clips of the DRL agent and the NDPS algorithm synthesized program, on the training track and one of the transfer (unseen) tracks, in that order:

`https://goo.gl/Z2X5x6`

On the training track, we can see that the steering actions taken by the DRL agent are very irregular, especially when compared to the smooth steering actions of the NDPS agent in the following clip. For the transfer track, we show the agents driving on the E-Road track. We can see that the DRL agent crashes before completing a full lap, while the NDPS agent does not crash. We have provided only small clips of the car during a race, to keep the video length and size small, but the behavior is representative of the agent for the entire race.