Programmatic policy extraction by iterative local search

Rasmus Larsen and Mikkel N. Schmidt

Department of Applied Mathematics and Computer Science,
Technical University of Denmark.

Abstract

Reinforcement learning policies are often represented by neural networks, but programmatic policies are preferred in some cases because they are more interpretable, amenable to formal verification, or generalize better. While efficient algorithms for learning neural policies exist, learning programmatic policies is challenging. Combining imitation-projection and dataset aggregation with a local search heuristic, we present a simple and direct approach to extracting a programmatic policy from a pretrained neural policy. After examining our local search heuristic on a programming by example problem, we demonstrate our programmatic policy extraction method on a pendulum swing-up problem. Both when trained using a hand crafted expert policy and a learned neural policy, our method discovers simple and interpretable policies that perform almost as well as the original.

1 Introduction

While neural policy representations are by far the most common in modern Reinforcement Learning (RL), other representations are worth considering. Programmatic policies provide a number of potential benefits: For example, a program might be read and understood by a human, something that generally is not possible with a neural network. Programs are also inherently compositional, which allows for not only reuse of policies in new combinations, but also compositional reasoning about their behavior.

However, learning programmatic policies is challenging. The structured, discrete space of programs does not allow for the gradient based optimization that neural policies benefit greatly from. Compared to a more standard inductive synthesis setting, programmatic policies must be evaluated in an environment that, whether simulated or real, is expensive to interact with. Several approaches exist that attempt to handle this interaction issue, such as learning a parametric environment model (Hein et al., 2017), imitating an existing policy (Bastani et al., 2018; Verma, Murali, et al., 2018), or evaluating fewer programs by learning to search more efficiently (Ellis et al., 2018). Furthermore, Verma, Le, et al. (2019) extend the imitation setting by providing a framework for intertwining RL and programmatic policy imitation.

This imitation-projection framework brings us a step closer to programmatic RL, where programs can be learned gradually through interaction with the environment. Essentially, this allows similar sample efficiency when compared to policy gradient methods, since the imitation-projection step is performed offline by scoring programs according to an imitation learning objective. One could even plausibly imagine that the inductive bias in a problem-specific policy language could lead to improved learning. The framework leaves many choices open in terms of how the policy update and programmatic policy projection steps are performed, as well as in terms of defining the

program space. Verma, Le, et al. (2019) perform experiments with a specific choice of update and projection, using two tailored program spaces based on PID controllers with either decision tree regression or Bayesian optimisation over some parameters as the projection operator.

In this paper we experiment with a more general program space based on Domain Specific Languages (DSLs) implemented in a typed lambda calculus. We demonstrate a method for re-using projections by local search around a previous projection, potentially reducing the required computational effort while allowing much longer programmatic policies to be found. Since imitation-projection greatly reduces environment interaction, the presented method takes advantage of this and performs relatively large searches in program space. Demonstrating the method on the pendulum swing-up task, we show that a simple and effective programmatic policy can be found by imitating a learned neural policy.

2 Methods

Our framework is based on previous work on imitation-projected and programmatically interpretable reinforcement learning (Verma, Murali, et al., 2018; Verma, Le, et al., 2019). We extend these methods to DSLs defined in a general-purpose programming language, namely the lambda calculus with Hindley-Milner type system. Using the building block of depth-limited type-directed program search, we construct an algorithm for finding a programmatic imitation of a given control policy. In order to discover programs much larger than what the depth limit of a single search allows, the algorithm performs multiple iterations of local search. Before describing the full algorithm, it is useful to consider the type-directed search itself.

2.1 Program synthesis by type-directed search

By choosing the lambda calculus with Hindley-Milner type system, we obtain an expressive program space, in which exhaustive search for programs of a specified type is straightforwardly defined. An especially useful feature of this program representation is that the type system can be used to reduce the search space, by filtering candidate programs that do not type check. Further filtering is possible, such as the filtering of semantically similar programs as done in MagicHaskeller (Katayama, 2008). In this program representation, DSLs can be defined as sets of typed functions and constants, which together represent the space of possible programs to be searched.

Our starting point for program synthesis is a simple version of depth-limited type-directed search. This choice is not a given; other, more advanced program search algorithms can be used. Here, the space of all programs in a DSL is viewed a tree, with the empty program at the root. Internal nodes are partial programs, with each branch being a candidate substitution for a hole in a partial program. Enumerating through this search tree results in generating all valid programs, according to the DSL, as the leaves of the tree.

We take advantage of the typed language to reduce the search space. Instead of yielding all syntactically valid programs, as explained above, we want to yield only well-typed programs. This type-directed search algorithm is the same as used by e.g. Ellis et al., 2018 to sample programs from a prior distribution, but instead of sampling, all programs are enumerated. To expand a node in the search tree, a typed hole (an empty program with a type annotation) in the corresponding partial program is selected for synthesis. Then, valid candidates are selected from the set of all DSL candidates by unification; the resulting context of the unification is propagated to the further expansion of child nodes, ensuring that any constraints are satisfied. All candidates that can produce the correct type are considered, even if they would need arguments applied to them first.

Algorithm 1 Depth-limited local search (typed neighborhood)

```
1    input: domain specific language 𝒟
2    input: imitation dataset Γ
3    input: initial program P
4    output: best program in typed neighborhood p*
5    function N_n^d(𝒟, P, l)    // generates the neighborhood for location l in P
6      return ∅ if d = 0
7      T = type(P, l)    // type of expression at l in P
8      C = {e | e : t ∈ 𝒟 ∧ T can unify with yield(t)}    // everything valid from DSL
9      P' = {edit(P, l, c) | c ∈ C}
10     // return all complete programs, and recursively generate remaining partial ones
11     return {p ∈ P' | p is complete} ∪ N_n^{d-1}(𝒟, p', l')   ∀p' ∈ {p' ∈ P' | p' is partial}
12       where l' is the location of the first hole in p'
13   end
14   p* ← ∅, v* ← ∞    // best program and imitation loss
15   foreach l ∈ set of all paths in P
16     E_l = expression(P, l)    // expression at location
17     𝒟' = 𝒟 ∪ (E_l, type(E_l))    // locally extend DSL with E_l
18     foreach p ∈ N_n^d(𝒟', P, l)
19       evaluate p on Γ and update p* and v*
20     end
21   end
```

Algorithm 2 Iterative local programmatic policy imitation

```
1    input: oracle policy f
2    optional input: initial program p_init = ∅
3    output: imitation program p_K
4    collect N on-policy trajectories using f:
5      τ_0 = ((s_0^0, f(s_0^0), s_1^0, f(s_1^0), …), …, (s_0^N, f(s_0^N), s_1^N, f(s_1^N), …))
6    create supervised dataset Γ_0 = {(s, f(s)) | s ∈ τ_0}
7    derive p_0 from Γ_0 by local search from p_init    // algorithm 1
8    for k = 1, …, K
9      collect M on-policy trajectories using p_{k-1}:
10       τ_k = ((s_0^0, p_{k-1}(s_0^0), s_1^0, p_{k-1}(s_1^0), …), …, (s_0^M, p_{k-1}(s_0^M), s_1^M, p_{k-1}(s_1^M), …))
11     create supervised dataset Γ' = {(s, f(s)) | s ∈ τ_k}
12     aggregate datasets:
13       Γ_k = Γ_{k-1} ∪ Γ'    // or Γ_k = Γ_0 ∪ Γ', which is cheaper
14     derive p_k from Γ_k by local search from p_{k-1}    // algorithm 1
15   end
```

2.2 Typed neighborhood

In alg. 1, the type-directed depth-limited synthesis algorithm is used to generate what we call the typed neighborhood of a given program. This construction applies the basic synthesis algorithm in multiple places of an existing program, resulting in a iterative local search method that can both add and remove subprograms in each iteration. Because of this, the algorithm can synthesize larger programs, while also benefiting from work performed in previous iterations of the search.

In more detail, we use a tree edit operation to define the programs contained within a typed neighborhood. Define the edit operation $\texttt{edit}(P, l, P')$ as the program obtained by replacing the subprogram at location l in program P with the program P'. Given a typed DSL $\mathcal{D}$, containing functions, constants, and their (polymorphic) types, the neighborhood of the program P at location l is the set of programs obtained by generating all well-typed expressions P' contained in $\mathcal{D}$, written $N_n^d(\mathcal{D}, P, l)$. The definition of a location is the root-to-expression path in the abstract syntax tree (AST) of the program. Here, we use the concept of a location in a generalized manner that can encompass multiple simultaneous locations, that is, a location l can represent

Algorithm 3 Imitation-Projected Programmatic Reinforcement Learning with Local Synthesis

```
1   input: initial policy π₀
2   optional input: initial program p₀ = ∅
3   output: trained policy π_J, program p_J
4   for j = 1, ..., J
5     π_j ← Update(π_{j-1})   // reinforcement learning, e.g. policy gradient
6     p_j ← Project(π_j, p_init = p_{j-1})   // program synthesis by algorithm 2
7   end
```

multiple paths in the AST that are to be simultaneously replaced using independent edit operations. The neighborhood of a program P is thus the union of the neighborhoods at all locations, $N_n^d(\mathcal{D}, P) = \bigcup_{l \in L(P)} N_n^d(\mathcal{D}, P, l)$, where the neighborhood is parameterized with a maximum depth d of the expressions generated by edit operations, and with a maximum number of simultaneous edits n.

Furthermore, the expression being edited is dynamically added as a candidate to the DSL, and for the depth evaluation this candidate counts as having a depth of 1. This allows an edit not just to replace an expression, but to also extend an expression by using it as part of the new expression, despite the result otherwise being too large (compared to the depth limit). The size of the neighborhood $|N_n^d(\mathcal{D}, P)|$ is quite sensitive to all involved parameters $\mathcal{D}$, P, n, and d, but these can be flexibly chosen based on the problem and available computational resources.

2.3 Policy extraction by local synthesis

We use the typed neighborhood to discover programs that imitate reinforcement learning policies. In this setting, input/output examples are obtained by executing an existing policy and storing the state observations together with the corresponding actions chosen by the policy in each state. Thus, the policy synthesis problem is framed as imitation learning. Like in previous policy imitation methods, an interactive dataset aggregation method such as DAgger (Ross et al., 2010) is used: Instead of imitating only on states that the expert experiences, which is called behavioral cloning, some experience from the imitation policy is periodically added to the set of states considered. This allows subsequent imitation iterations to correct mistakes that otherwise wouldn't be observed, since the expert policy never experiences these mistakes. However, unless the expert is a global optimum, it is possible that it also makes mistakes on states which are not usually observed, and for this reason it is not always a clear benefit. The iterative imitation approach using the typed neighborhood is described in alg. 2.

One purpose of this search algorithm is to fit into the full imitation-projection framework from Verma, Le, et al. (2019). A simple, modified version of this framework is shown in alg. 3. The difference consists of the projection step, which now also depends on the previous projection, as enabled by alg. 1. Compared to alg. 2, the main difference is that each iteration also contains an Update step, which optimizes the expert policy.

3 Experiments

We present three different program synthesis experiments: The first is a programming by example (PBE) task with sampled ground truth programs, demonstrating the efficacy of the local search heuristic in alg. 1. The second is a policy extraction task, testing alg. 2, where the ground truth is a hand-coded policy. In these two first experiments, the DSL used for the search contains the true program used to generate observational data. Finally, in our third experiment we examine if we
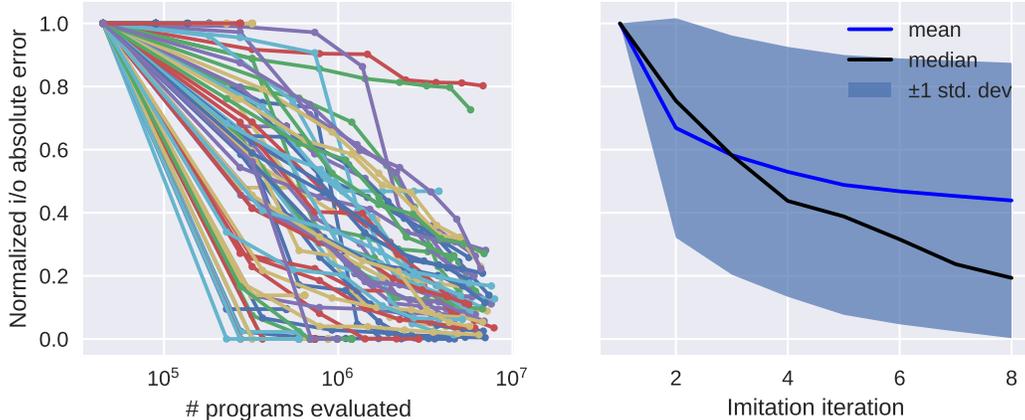
Figure 1: Programming by example: Learn a program from input-output examples. Experiment on 100 sampled programs. Left: For each program, the absolute error (normalized wrt. first iteration) by number of programs evaluated. Right: Mean, median, and standard deviation by search iteration.

can learn a simple yet effective policy by imitation from a more complicated neural network policy which is trained using an existing reinforcement learning algorithm.

3.1 Programming by example with local program search

As a first evaluation of the method, we used a straightforward PBE task, whose purpose is to show that the described iterative local search is capable of synthesizing nontrivial programs from input-output specifications.

DSL: We defined a language containing the set of constants $\{-1, 0, 0.5, 0.8, 1, 3, 5, 6, \texttt{true}\}$, which are all `Floating` point numbers except `true` which is `Boolean`, and the functions with associated type signatures $\{\texttt{if} : \texttt{Bool} \Rightarrow \texttt{T0} \Rightarrow \texttt{T0} \Rightarrow \texttt{T0}, > : \texttt{Float} \Rightarrow \texttt{Float} \Rightarrow \texttt{Bool}, \wedge : \texttt{Bool} \Rightarrow \texttt{Bool} \Rightarrow \texttt{Bool}, \oplus : \texttt{Bool} \Rightarrow \texttt{Bool} \Rightarrow \texttt{Bool}, - : \texttt{Float} \Rightarrow \texttt{Float} \Rightarrow \texttt{Float}), * : \texttt{Float} \Rightarrow \texttt{Float} \Rightarrow \texttt{Float}, \cdot^2 : \texttt{Float} \Rightarrow \texttt{Float}\}$.

Data: The observation space (i.e. input) to these programs consists of three `Float`s, which are distinct variables that can be used just like constants. 10 sets of these numbers were randomly sampled as inputs to be used during synthesis. Ground truth programs of some length, as a simple proxy for complexity, were sampled from a weighted distribution over the DSL. In order to obtain samples that have a reasonable length, we designed a distribution on the abstract syntax of our DSL that puts more probability on higher-arity functions. Further, the probability of sampling `true` was weighted significantly down, while the probability of sampling an input variable was weighted higher. Since program length is not the best measure of complexity, samples were rejected on other criteria as well. Programs were discarded if: the length of the program (number of tokens) was less than 8, program output was constant, or an input-output equivalent program, on some randomly chosen inputs, existed within a depth 4 search of the DSL.

Results and discussion: Results from $d = 4$ local search on 100 sampled ground truth programs can be seen in fig. 1, which shows that for many of the programs an exact fit is found on the given inputs. Even for programs where an exact solution is not found, most of the searches show significant progress through the iterations, although a few make no progress at all. Since the search is deterministic, if no improvement is made in an iteration, further iterations will not lead to better results. It should also be noted that a single iteration of search with $d = 5$ in this setting

5

corresponds to evaluating about as many programs as 20 iterations with $d = 4$.

3.2 Imitation of a programmatic pendulum swing-up policy

Next we examined if we were able to discover a ground truth programmatic policy.

Task: We based the experiment on a simple, classical control problem, the pendulum swing-up task. The state space consists of the angle and angular velocity of the pendulum, and the action space is the torque applied to the base of the pendulum, normalized to the interval [-1, 1]. This two-dimensional state space allows us to easily display and visually compare policies. The simulator is discretized with a time step of 0.05s, and an episode is 200 steps long. The reward function is $r(\theta, \dot{\theta}, a) = ((\theta + \pi \pmod{2\pi}) - \pi)^2 + 0.1\dot{\theta} + 0.001a^2$, which results in a reward of 0 if the pendulum is perfectly balanced with no torque applied, and a reward of $-\pi^2$ when the pendulum is pointing straight down while not moving. While the state space of the pendulum task is $(\theta, \dot{\theta})$, the observation space supplied to the policies is $(\texttt{x1}, \texttt{x2}, \texttt{x3}) = (\sin\theta, \cos\theta, \dot{\theta})$.

DSL: We used a simple, pure DSL with primitives suitable for solving the RL task, containing the constants $\{-6, -1, 1, 0.5, 0.6, 8, 10\}$, and the functions $\{\texttt{if} : \texttt{Bool} \Rightarrow \texttt{T0} \Rightarrow \texttt{T0} \Rightarrow \texttt{T0}, \texttt{gt} : \texttt{Float} \Rightarrow \texttt{Float} \Rightarrow \texttt{Bool}, \texttt{sub} : \texttt{Float} \Rightarrow \texttt{Float} \Rightarrow \texttt{Float}, \texttt{add} : \texttt{Float} \Rightarrow \texttt{Float} \Rightarrow \texttt{Float}, \texttt{mul} : \texttt{Float} \Rightarrow \texttt{Float} \Rightarrow \texttt{Float}, \texttt{sign} : \texttt{Float} \Rightarrow \texttt{Float}, \texttt{sqr} : \texttt{Float} \Rightarrow \texttt{Float}\}$.

Ground truth: We hand crafted a ground truth policy in the DSL, capable of swinging up the pendulum from any starting state. The policy achieves an average reward of approximately -211 and a maximum of -113. This handcrafted policy is given by the program `((if ((gt x1) 0.6)) ((sub ((mul x2) -6)) x3)) (sign ((mul ((sub ((add ((mul 0.5) (sqr x3))) ((mul ten) ((sub x1) 1)))) 8)) ((mul -1) x3)))`.

Synthesis: At each iteration of the local search we used a search depth of $d = 4$ which was found to be enough to discover the `if` expression that switches between swing-up and balancing. As training data we used $N = 5$ state trajectories from the ground truth policy and $M = 2$ trajectories from the latest programmatic imitation policy. All training states were expert labelled with actions from the ground truth policy. For evaluation we simulated 100 rollouts from uniformly random states in the range $\frac{\pi}{2} \le \theta \le \frac{3\pi}{2}$, $-1 \le \dot{\theta} \le 1$, which is the pendulum below horizontal with relatively low velocity.

Results and discussion: The results of the experiment is shown in fig. 2 (left column). After four iterations of imitation learning a simple policy was found, capable of balancing the pendulum and swinging up from some states. After approximately ten iterations the policy could effectively swing up and balance the pendulum from any state. The imitation learning did not find the ground truth programmatic policy by iteration 10, likely due to the small number of observations in certain areas of the state space. Nonetheless, it managed to synthesize an effective policy which is quite similar to the ground truth.

3.3 Imitation of a neural pendulum swing-up policy

Finally, we examined if we were able to discover a simple, interpretable policy in a more realistic setting, where we synthesized by imitation learning from a neural policy. The task, DSL, and synthesis procedure were as described in the previous experiment, with the ground truth policy as the only difference.

Ground truth: The neural ground truth policy was found by TD3 (Fujimoto et al., 2018), using feed-forward neural networks with 2 hidden layers of 24 neurons for both the actor and critic. Training was run for 5 million steps with a learning rate of 10^{-4} to ensure relatively good convergence.
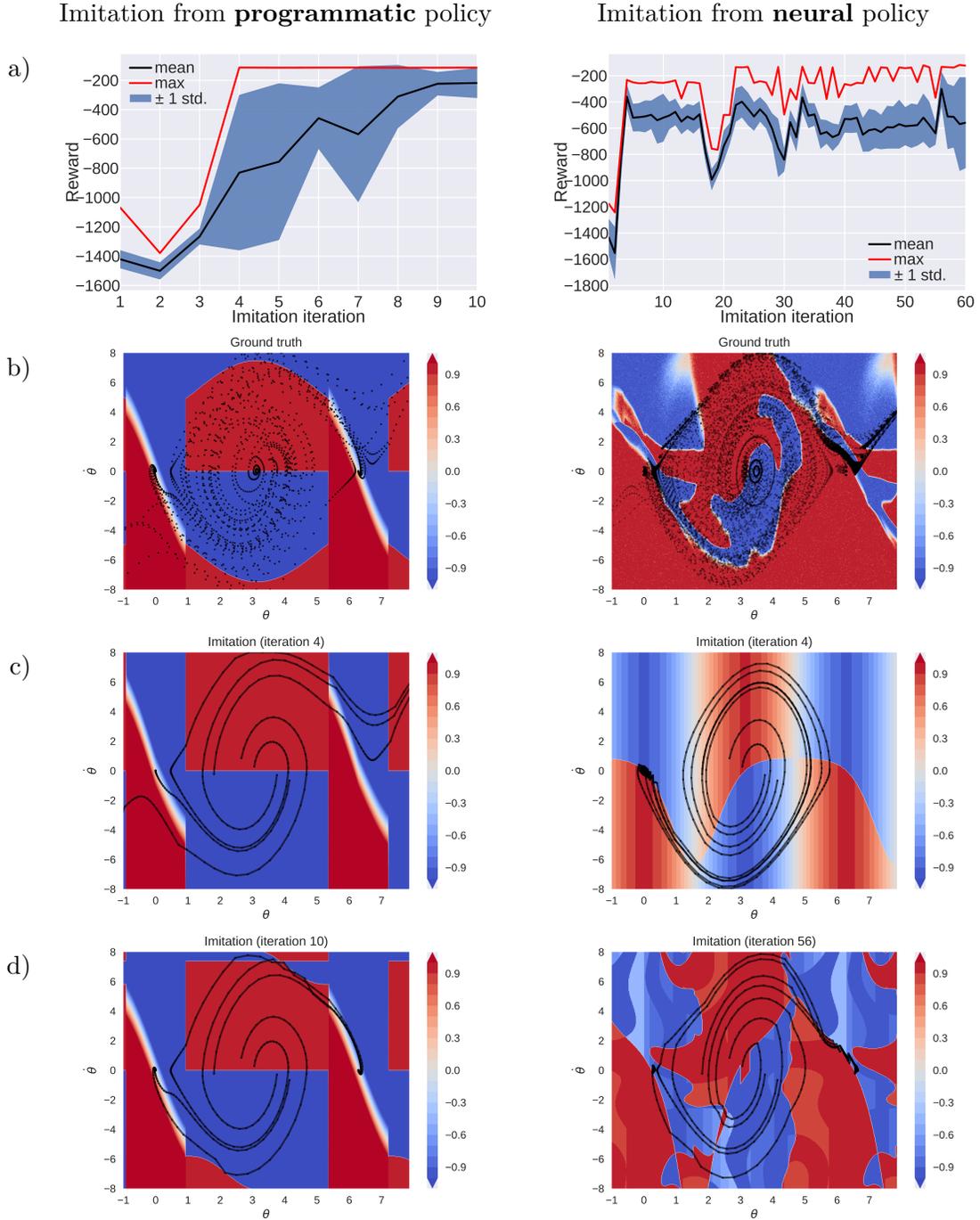
Figure 2: Pendulum swing-up imitation learning of a programmatic policy from a ground truth programmatic (left) or neural (right) policy. Policies are visualized as a heat map; the state space is pendulum angle, θ, and angular velocity, $\dot{\theta}$, and the action is pendulum torque. The goal state is $\theta = 0 \pmod{2\pi}$, $\dot{\theta} = 0$. a) Cumulative reward of test trajectories. b) Ground truth programmatic/neural policy. Points indicate all states seen during training. c) Programmatic policy found after four iterations of imitation learning, with five test trajectories shown. d) Programmatic policy found after several more iterations, with five test trajectories shown.

7

Results and discussion: The results of the experiment is shown in fig. 2 (right column). After four iterations of imitation learning, a simple imitating policy capable of swinging up and balancing the pendulum is found. This imitation policy is `(mul x1) (cos (exp (sign ((add x3) ((add -1) (sqr (exp x2)))))))`. After several more iterations, at iteration 56, a significantly more complicated programmatic policy was found which resembles the neural policy more closely but yields only a minor performance improvement, while being significantly less interpretable. This imitation policy has a length of 121 tokens, i.e. function calls plus arguments.

4 Discussion

We have presented and evaluated our method with simple experiments, and much remains to be done. As mentioned, one goal is to integrate the local search with reinforcement learning as described by alg. 3. While simple, we believe that the presented results show potential, especially through the programs that were discovered in only a few iterations. In particular, it would be interesting to evaluate this approach on more structured tasks, where neural networks might struggle with generalization while a program could be found that immediately generalizes. In such a setting, we could also take better advantage of type-directed search, with more complicated DSLs containing e.g. logic, matrix or computer vision functions potentially still remaining computationally tractable.

It should also be mentioned that local, iterated synthesis as a concept remains orthogonal to several other improvements in program synthesis; for example, enumerating or sampling programs according to a learned probability distribution as in e.g. Ellis et al., 2018 is possible, as is better filtering as in Katayama (2008). Instead of depth-limited search, it would be possible to limit the search to programs above a certain likelihood. However, it seems unclear how this distribution would be effectively learned for policies.

4.1 Related work

Previous work on synthesizing programmatic policies at the intersection of RL and genetic programming (GP) include GPRL (Hein et al., 2017) which is based on offline GP, performed in a previously learned parametric model of the system of interest. They include a comparison with behavioral cloning, i.e., direct imitation of the actions of a trained policy. Their method performs better on the actual (simulated, but not learned) system. It is well known that behavioral cloning can lead to poor performance, e.g. Ross et al. (2010), which could explain the observed performance gap. It seems likely that interaction with the model can overcome some of the distributional issues arising from behavioral cloning. In RL, it might be preferable to not learn a parametric model if it is used for credit assignment (i.e. policy learning) (Hasselt et al., 2019). Gupta et al. (2020) proposed a method for using program repair in neural program synthesis. After neural synthesis, the resulting program might not be correct or even satisfy the input-output relation. The authors propose to learn a neural debugger that outputs so-called edits which correct potential errors in the program. The relation to this work is apparent in how we use an edit operator to define the neighborhood of a program. Kamio et al. (2003) describe a way to integrate GP, RL and simulated systems. By first synthesizing a policy using GP in the simulated system, it can later be adapted and fine-tuned through RL, allowing the policy to function on a real robot. Inala et al. (2020) describe an imitation learning method that improves the inductive generalization by adapting the teacher distribution according to the imitating policy. The presented neighborhood search method can be considered an instance of the (Very) Large-Scale Neighborhood Search framework (Pisinger and Ropke, 2010). Deterministic versions of genetic algorithms have been considered before, such as in Salomon (2003).

References

Bastani, Osbert, Yewen Pu, and Armando Solar-Lezama (May 2018). "Verifiable Reinforcement Learning via Policy Extraction". URL: http://arxiv.org/abs/1805.08328.

Ellis, Kevin, Lucas Morales, Mathias Sablé Meyer, Armando Solar-Lezama, and Joshua B. Tenenbaum (2018). "Dreamcoder: Bootstrapping domain-specific languages for neurally-guided bayesian program learning". In: *Neural Abstract Machines & Program Induction Workshop at NIPS*.

Fujimoto, Scott, Herke van Hoof, and David Meger (Feb. 2018). "Addressing Function Approximation Error in Actor-Critic Methods". URL: http://arxiv.org/abs/1802.09477.

Gupta, Kavi, Peter Ebert Christensen, Xinyun Chen, and Dawn Song (July 2020). "Synthesize, Execute and Debug: Learning to Repair for Neural Program Synthesis". In: *Advances in Neural Information Processing Systems*. URL: http://arxiv.org/abs/2007.08095.

Hasselt, Hado P. van, Matteo Hessel, and John Aslanides (2019). "When to use parametric models in reinforcement learning?" In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc.

Hein, Daniel, Steffen Udluft, and Thomas A. Runkler (Dec. 2017). "Interpretable Policies for Reinforcement Learning by Genetic Programming". URL: http://arxiv.org/abs/1712.04170.

Inala, Jeevana Priya, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama (2020). "Synthesizing Programmatic Policies That Inductively Generalize". In: *International Conference on Learning Representations, (ICLR)*.

Kamio, Shotaro, Hideyuki Mitsuhashi, and Hitoshi Iba (2003). "Integration of Genetic Programming and Reinforcement Learning for Real Robots". In: *Genetic and Evolutionary Computation — GECCO 2003*. Springer Berlin Heidelberg, pp. 470–482.

Katayama, Susumu (2008). "Efficient exhaustive generation of functional programs using Monte-Carlo search with iterative deepening". In: *PRICAI 2008: Trends in Artificial Intelligence*. Lecture notes in computer science. Springer Berlin Heidelberg, pp. 199–210. ISBN: 9783540891963.

Pisinger, David and Stefan Ropke (2010). "Large Neighborhood Search". In: *Handbook of Metaheuristics*. Ed. by Michel Gendreau and Jean-Yves Potvin. Springer US, pp. 399–419. ISBN: 9781441916655.

Ross, Stephane, Geoffrey J Gordon, and J Andrew Bagnell (Nov. 2010). "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning". In: arXiv: 1011.0686.

Salomon, R. (2003). "The deterministic genetic algorithm: implementation details and some results". In: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*. IEEE. ISBN: 9780780355361. DOI: 10.1109/cec.1999.782001.

Verma, Abhinav, Hoang Le, Yisong Yue, and Swarat Chaudhuri (2019). "Imitation-Projected Programmatic Reinforcement Learning". In: *Advances in Neural Information Processing Systems 32*. Ed. by H Wallach, H Larochelle, A Beygelzimer, F dÁlché-Buc, E Fox, and R Garnett. Curran Associates, Inc., pp. 15752–15763.

Verma, Abhinav, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri (Apr. 2018). "Programmatically Interpretable Reinforcement Learning". URL: http://arxiv.org/abs/1804.02477.