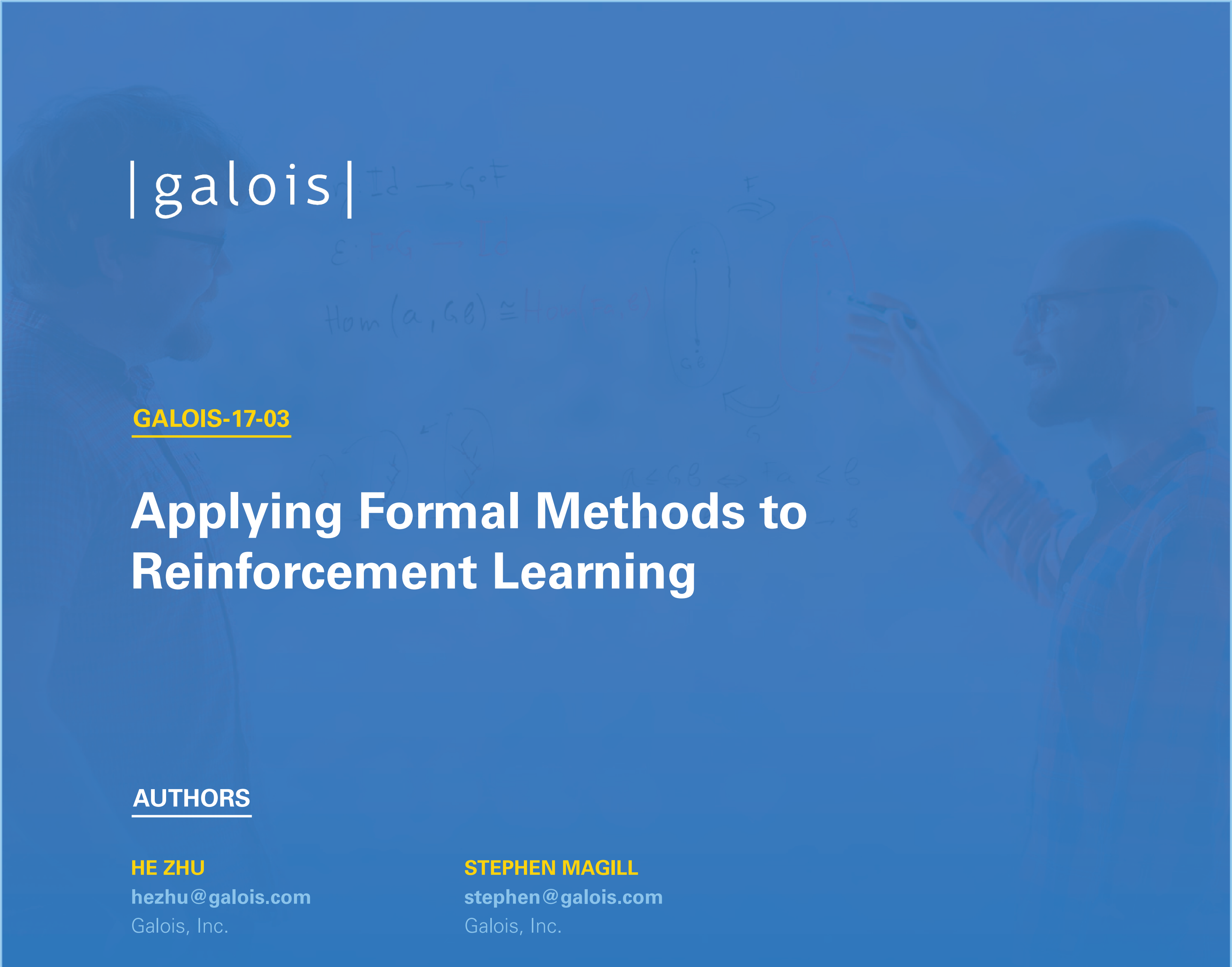


GALOIS-17-03

Applying Formal Methods to Reinforcement Learning

AUTHORS

HE ZHU
hezhu@galois.com
Galois, Inc.

STEPHEN MAGILL
stephen@galois.com
Galois, Inc.

Abstract

We report our research on formal methods guided testing of autonomous systems. In particular, we looked at closed-loop control systems that incorporate neural network based reinforcement learning components. Typically, reinforcement learning approaches train good control policies only after millions of experiments due to the sparsity of high reward actions. This may be unacceptable in an online learning setting. Our solution to this problem is inspired by Imitation Learning, a learning from demonstrations framework, in which an agent learns a control policy by directly mimicking demonstrations provided by an expert.

Our approach automatically generates expert trajectories using a variant of the Monte Carlo tree search method. Unlike traditional methods, which eagerly expand a Monte Carlo search tree and draw samples on tree nodes, our technique is lazy and counterexample driven. This gives us the ability to directly and more efficiently get states and actions that are weighted highly. For each violation of safety properties found by the simulator, we traverse the Monte Carlo search tree induced by the failed trajectory. This search procedure allows us to collect expert trajectories that satisfy our safety properties by forcing the agent to take certain actions in key states to avoid property violation. Our technique proceeds by aggregating an expert trajectory dataset at each training iteration. The intuition is that, over iterations, it builds up the set of inputs that the policy is likely to encounter during its execution based on previous experience, which allows us to use supervised learning algorithms for efficient reinforcement learning. As our solution is counterexample-driven, the available expert trajectories may still be sparse. To combat this challenge, we use a constraint encoding of the neural network policy model and an SMT solver to compute states that are nearby states visited by an expert trajectory, but for which the policy chooses significantly different actions. In a robust policy, such nearby states should behave similarly, and so these ”edge cases” are likely to be interesting from a safety property perspective. We then use the above tree based search procedure again to test and improve the policy model using these newly generated trajectories.

In our experiments, we found that our algorithm enabled policy training with several orders of magnitude less data. We show the effectiveness of our approach via a case study on the game Pong.

1 Introduction

Machine Learning has recently made a significant leap forward as the rapid adoption of new Deep Learning techniques [1, 2] have enabled advances in many domains, such as computer vision, speech recognition, and text translation. Reinforcement Learning, which aims to train a policy that maximizes the reward obtained by an agent in a given environment, combines well with deep learning approaches and has resulted in performance surpassing that of humans in various Atari 2600 games [3] and the game of Go [4].

For safety-critical autonomous systems, analysis of the performance and safety of deep learning models is of great interest. This paper contributes to our understanding of the intersection of Formal Methods and Reinforcement Learning by focusing on two important aspects of the problem space: *Formal Methods in Policy Training* and *Formal Methods in Policy Verification.*

Formal Methods in Policy Training. Operating deep reinforcement learning successfully in the real world is much harder than playing Go or Space Invaders with it. Challenges arise from the fact that classic reinforcement machine learning approaches are training a competitive or acceptable policy from scratch, out of millions of samples in a sufficiently long cycle. Oppositely humans do not learn in this way as they often Learn From Demonstrations (a.k.a examples) given by teachers or experts. As opposed to classic Reinforcement Learning, Imitation Learning and Apprenticeship Learning are *Learning From Demonstrations* frameworks in which an agent learns a control policy from a dynamic environment by observing demonstrations delivered by an expert agent [5].

In this paper, we explore how to combine learning from demonstrations with classic reinforcement learning. We would like to ensure that trained agents behave in a way that best mimics expert trajectories. In our context, expert trajectories focus on safety only as we are interested in obtaining an export trajectory to fix an agent's behavior when it is observed violating safety constraints. We still rely on reinforcement learning to balance safety and optimality by maximizing long-term rewards.

Two possible challenges are: (a) a naive approach to apply learning from demonstrations to reinforcement learning might be training a policy to predict an expert's behavior given training data of the encountered states as input and expert actions as output. However since the learner's prediction affects future input states during execution, this violates the *i.i.d.* assumption made by most statistical learning approaches. As soon as the learner makes a mistake, it may encounter completely different states than those under expert demonstrations, leading to a compounding of errors. (b) It is very hard to *automatically* generate expert demonstrations in an online reinforcement learning framework with low cost let alone precisely synthesize a reward function from the expert demonstrations that could explain expert behaviors.

Our solution to challenge (a) is using a DAGGER (Dataset Aggregation) [6] based technique to improve reinforcement learning in online learning setting. This technique proceeds by collecting an expert trajectory dataset at each training iteration. The intuition is that, over iterations, it builds up the set of inputs that the trained policy is likely to encounter during its execution based on previous experience. Our solution to challenge (b) is reward function synthesis by applying formal methods to systematically diagnose the execution path of a trained agent, which leads to a safety violation, to determine which actions must take at certain states to avoid the violation, *resp.* automatic expert trajectories generation, and then assign high rewards to those (state, action) pairs for reinforcement learning to train an improved policy, *resp.* extracting a reward function given observed expert behaviors.

Formal Methods in Policy Verification Safety of AI systems is receiving increasing attention due to their potential to cause harm in safety-critical autonomous systems. In many cases the safety of a decision can be reduced to ensuring the correct behavior of a machine learning component. However safety assurance and verification methodologies for machine learning are still very challenging.

Traditionally, the correctness of a neural network based classifier is expressed in terms of risk, defined as the probability of misclassification of a given, say image, weighted with respect to the input distribution. Despite having high accuracy, neural networks have been shown to be susceptible to small perturbations to inputs that can cause it to become mislabeled. We are interested in two directions of applying formal methods, especially formal verification approaches, to neural network verification: (1) the safety of an individual decision, and to this end focus on the key property of the classifier being invariant to perturbations, which is also known as pointwise robustness, especially on expert trajectories discovered from our Formal Methods in Policy Training. (2) the ability to additionally improve neural network safety with verification engines. For both of the two directions, we symbolically encode neural networks into first order arithmetic constraints and leverage modern linear programming solvers and SMT solvers for efficient verification.

2 DAGGER based Policy Training

As discussed in Sec. 1, a naive adoption of learning from demonstrations to reinforcement learning, where future observations depend on previous predictions (actions), might lead to poor performance as it violates the common *i.i.d.* assumptions made in statistical learning. Our solution is applying a DAGGER-like (Dataset Aggregation [6]) algorithm for reinforcement learning which learns a stationary deterministic policy guaranteed to perform well under the distribution of states the policy itself induces. Such a reduction based approach allows us reusing existing supervised learning algorithms for efficient reinforcement learning.

Our DAGGER based training approach is depicted in Algorithm 1. Although it initializes the first policy π_1 with an online learner, it can be started with any policy in space. At the first iteration of the training loop, it uses the expert's policy to gather a dataset of expert trajectories $\mathcal{D}$ (obtained from an application of formal methods for which we explain in the next section) and train a policy π_2 that best mimics the expert on those trajectories. Then at iteration n, it uses π^n to collect more expert trajectories and adds those trajectories to the dataset $\mathcal{D}$. The next policy π^{n+1} is the policy that best mimics the expert on the whole dataset $\mathcal{D}$ in hindsight, i.e. under all trajectories seen so far over the iterations.

Algorithm 1: Automated Expert Trajectories Aggregation.

```
Initialize $\mathcal{D} \leftarrow \emptyset$;
Initialize $\pi_1$ to a policy trained via an online learner;
for $i$ = 1 to $N$ do
    Sample $T$-step trajectories using $\pi_i$;
    Get dataset $\mathcal{D}_i = \{(s, \pi^*(s))\}$ of visited states by $\pi_i$ and actions $\pi^*(s)$ given by best trials
      from Algorithm 2;
    Aggregate datasets: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_i$;
    Use the online learner to re-train a classifier $\pi_{i+1}$ on $\mathcal{D}$;
end
return best $\pi$ on validation
```

The original DAGGER approach [6] explicitly asks an expert to provide expert trajectories as demonstrations, which is a very high-cost step in realistic setting. As opposed to it, Algorithm 1 automatically synthesize expert trajectories.

3 Policy Diagnosis

The goals of policy diagnosis are: (*i*) Automatically synthesize expert trajectories; (*ii*) Automatically synthesize a reward function to explain expert trajectories.

Our formal methods based approach to policy diagnosis is essentially a variant of Monte Carlo method, which considers multiple trajectories of a game to find expert trajectories. Monte Carlo Tree Search (MCTS) is a method for making optimal and expert decisions in artificial intelligence problems, typically move planning in combinatorial games. Intuitively, our idea is inspired by the hypothesis that slow but good traditional game players can train much faster deep networks in supervised fashion to outperform similar deep networks trained by more general reinforcement learning methods [7]. (Deep networks trained on such results allow to retrieve results within orders of magnitude faster time.)

There are several points worth to be mentioned as they form the basis of the design of our algorithms. *(i)* The basic MCTS algorithm is simplicity itself. It builds a search tree, expands it node by node according to the outcomes of simulated playouts. While MCTS performs extremely well on Atari 2600 games, it is also too computationally expensive. Our approach must be able to combat such a limitation to be used in a realistic setting. *(ii)* Typically, decision making in safety-critical autonomous systems is either solely based on machine learning, through end-to-end controllers, or involves some combination of logic-based reasoning and machine learning components, where an image classifier may produce a classification, say speed limit or a stop sign, that serves as input to a controller. Our search procedure needs to take logic-based constraints over machine learning components into consideration.

Algorithm 2: Policy diagnosis on discrete control program P.

```
Initialize $T$ to the number of steps of a trajectory $t$ failed on $\mathcal{P}$;
Symbolically re-execute $\mathcal{P}$ on $t$ generating symbolic constraints on each state $s$ of $t$;
for $i = T$ to $1$ do
    $\mathcal{D} = \emptyset$;
    for $j = i$ to $T$ do
        foreach Action $a \in \mathcal{A}$ do
            Conjoin the symbolic constraint at $s_j$ with the logic constraint to choose $a$ at $s_j$;
            if there exists an input satisfying the current symbolic constraint then
                $\mathcal{D} = \mathcal{D} \cup \{(s_j, a)\}$;
                Transit to $s_{j+1}$ by choosing action $a$ at $s_j$;
                Sample a $(T-j)$ steps trajectory $t'$ on $\mathcal{P}$ using current policy from $s_{j+1}$;
                if $t'$ does not fail on $\mathcal{P}$ then
                    return $\mathcal{D}$;
        end
    end
end
return $\emptyset$
```

Based on the above design factors, instead of building an expensive Monte Carlo search tree a priori and then training a deep policy network [7], our technique is counterexample driven. The basic algorithm is depicted in Algorithm 2. We systematically expand the given trajectory t while collecting symbolic constraints that enable direct generation of new trajectories allowed by P via constraint solving. This search procedure allows us to collect expert trajectories $\mathcal{D}$ in which the agent must take certain actions to avoid the failure in t again. We send $\mathcal{D}$ to Algorithm 1 for dataset aggregation. We believe Algorithm 1 and Algorithm 2 together can improve training convergence and performance by emphasizing more meaningful information (a.k.a explanations of expert behaviors). In our experiments, we found that our algorithms enabled learning component training with several orders of magnitude less data.

Because we search from a tree like data structure, we can develop multiple efficient tree search algorithms. Indeed, since the search tree can grow extremely fast in a continuous action space, in the future work, we will develop a backward goal-directed symbolic exploration technique that enables only traversing actions that satisfy safety properties of interest.

4 Constraint Solving based Policy Verification

We apply automated verification to discharge safety of classification decisions made by policy based on feed-forward deep neural networks. The core idea is that we encode a neural net policy $f(x) = a$, which chooses action a when given input state x, as constraints $C_f(x, a)$. In the next section, we show how to express constraints $C_f(x, a)$. Our work is based on recent advance on constraint-based encoding of neural networks that enables proofs of neural networks safety in our desire [8, 9, 10, 11].

The starting point of our verification algorithm is based on $\mathcal{D}$, obtained from expert trajectories generation by Algorithm 2. For a given state x (a point in a vector space) from $\mathcal{D}$, we assume that there is a (possibly infinite) region η around that point that incontrovertibly supports the decision, in the sense that all points in this region must have the same class. This region is specified by the user and can be given as a small diameter. We define a policy decision to be safe for input x and region η if all the points in η produce a same result as x. Recall that the search procedure Algorithm 2 discovers states in which the policy must take certain actions to avoid failures. Our intuition is that, because the action taken at x is very important to avoid policy crash, we anticipate that a small enough perturbation made to x should not affect policy predication result otherwise we may encounter a policy failure again.

Safety Verification. Formally, we want to prove that the following formula is unsatisfiable:

$$\forall (x^*, a^*) \in \mathcal{D}.\ C_f(x, a) \wedge ||x^* - x|| < \eta \wedge a \neq a^*$$

Neural Network Encoding. We now show how to encode $C_f(x, a)$ so that the above formula is solvable via off-the-shelf linear programming solvers and SMT solvers. We assume f has form $f(x) = argmax_{a \in \mathcal{A}}\{[f^k(f^{k-1}(...(f^1(x))...))]_a\}$ where the i^{th} layer of the network is a function $f^i : \mathbb{R}^{n_{i-1}} \rightarrow \mathbb{R}^{n_i}$, with $n_0 = n$ and $n_k = |\mathcal{A}|$ with n being the number of input neurons and $\mathcal{A}$ represents all the possible actions. We introduce the variables $x^{(0)}, \cdots, x^{(k)}$ into our constraints, with the interpretation that $x^{(i)}$ represents the output vector of layer i: $x^{(i)} = f^i(x^{(i-1)})$.

We describe the encoding of input, fully-connected, ReLU and output layers. The constraint $C_{in}(x) = (x^{(0)} = x)$ encodes the input layer. For each layer f^i, we encode the computation of $x^{(i)}$ given $x^{(i-1)}$ as a constraint C_i. In a fully-connected layer, $x^{(i)} = W^{(i)}x^{(i-1)} + b^{(i)}$, which can be encoded using constraints $C_i = \bigwedge_{j=1}^{n_i} \{x_j^{(i)} = W_j^{(i)}x^{(i-1)} + b_j^{(i)}\}$, where $W_j^{(i)}$ corresponds to the j^{th} row of $W^{(i)}$. In a ReLU layer, $x_j^{(i)} = max\{x_j^{(i-1)}, 0\}$ for each $1 \leq j \leq n_i$, which can be encoded using constraints $C_i = \bigwedge_{j=1}^{n_i} C_{ij}$ where $C_{ij} = ((x_j^{(i-1)} < 0 \wedge x_j^{(i)} = 0) \vee (x_j^{(i-1)} \geq 0 \vee x_j^{(i)} = x_j^{(i-1)}))$. In the output layer, the constraints $C_{out}(a) = \bigwedge_{a' \neq a} \{x_a^{(k)} \geq x_{a'}^{(k)}\}$ ensure that the action a is selected by the policy f. Finally, the constraints $C_f(x, a) = C_{in}(x) \wedge \left(\bigwedge_{i=1}^{k} C_i\right) \wedge C_{out}(a)$ faithfully encodes the computation of policy f. Notice that the above encoding can be extended to convolutional layers and max-pooling layers as convolutional layers can be encoded similarly to fully-connected layers and max-pooling layers can be encoded similarly to ReLU layers.

Counterexample Guided Policy Safety Enhancement. The above sections provide a tool of formally verifying safety of neural network based policy with respect to pointwise robustness on expert trajectories $\mathcal{D}$ but does not address how we can improve a policy with verification counterexamples. Algorithm 3 combines formal verification with testing to iteratively exploit counterexamples to enhance the safety of a policy. Intuitively we use testing (sampling) to validate whether a verification counterexample is spurious. A spurious counterexample is removed from expert trajectories $\mathcal{D}$ as they might not be as important as the other samples in $\mathcal{D}$. A counterexample that is not spurious is diagnosed using Algorithm 2 as a part of dataset aggregation. Therefore Algorithm 3 can be considered as another variant of DAGGER as we train an improved policy on aggregated expert trajectories dataset.

Algorithm 3: Using Policy Verification to improve safety of a policy f.

```
Obtain expert trajectories $\mathcal{D}$ from calling Algorithm 1;
Initialize $VerifySucc$ = True;
repeat
    $VerifySucc$ = True;
    foreach State and action pair $(s^*, a^*) \in \mathcal{D}$ do
        foreach Action $a \in \mathcal{A}$ do
            if $\exists\ (s^*, a^*) \in \mathcal{D}.\ C_f(s, a) \wedge ||s^* - s|| < \eta \wedge a \neq a^*$ then
                Sample a trajectory $t$ using policy $f$ from $s$;
                if $t$ does fail then
                    Call Algorithm 2 to obtain expert trajectories $D_i$ to fix $t$;
                    if $D_i \neq \emptyset$ then
                        $\mathcal{D} = \mathcal{D} \cup D_i$;
                        $VerifySucc$ = False;
                else
                    $\mathcal{D} = \mathcal{D} - \{(s^*, a^*)\}$;
        end
    end
    Train policy $f$ on $\mathcal{D}$;
until VerifySucc is True;
return $f$ as an improved policy
```

We have incorporated this verification and testing mixed approach to formal policy verification that enables more effective testing of systems that use neural networks for decision-making and control. We found that it allowed us to find safety violations that were not identified by randomized testing.

5 Limitations and Future Work

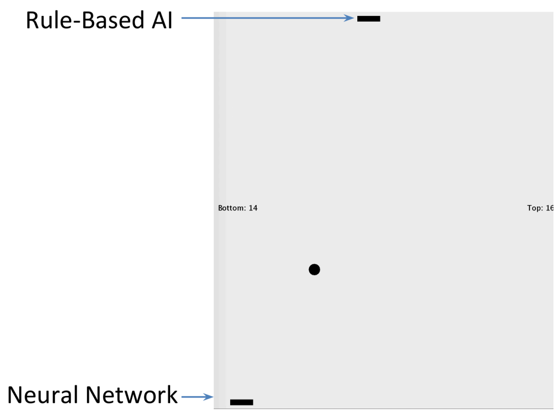
So far we have focused on pointwise robustness verification for a neural policy. We will need to expand our ability to system-wide verification, formally verifying a policy satisfying a safety requirement given as a reachability property.

First, we would like to study the safety and robustness of a whole neural network classifier in the sense that we desire to compute the average minimum distance to a misclassification and are independent of the data point. We plan to parallelize the queries to SMT solvers to avoid a belowup in search space. Second, directly discharging a proof on systems with heavy dependencies on neural policies might not be scalable due to hight complexity of neural networks. We hence would exploit an assume-guarantee style verification paradigm. The goal is to synthesize an *interpretable* model from a policy and then (*i*) verify that the synthesized model is conditionally equivalent to the neural policy under certain resilient (error) bound using SMT or linear programming solvers and (*ii*) perform system-wide bounded model checking agains safety properties replacing complex neural policies with their lightweight interpretable models a priori.

6 Case Study: The Pong Game

Pong is a classic two-dimensional sports game that simulates table tennis. In this demo, the policy trained from reinforcement learning controls an in-game paddle by moving it horizontally across the bottom of the screen, and can compete against a computer-controlled opponent, another player controlling a second paddle on the opposing side. Players use the paddles to hit a ball back and forth. For exposition purpose, we simplify training task to learning a neural network policy that never fails to return the ball from its opponent.

We train our neural controller using policy gradient, which is a type of policy optimization approach. It relies upon optimizing parametrized policies with respect to the expected return (long-term cumulative reward) by gradient descent. They do not suffer from many of the problems that have been marring dynamic programming based reinforcement learning approaches such as the lack of guarantees of a value function and the intractability problem resulting from uncertain state information.



Policy Training Experiment. In our experiment, we first use standard policy gradient implementation to train a policy for the neural controller. Intuitively, we take all decisions the controller made in the winning games and do a positive update, filling in a +1.0 in the gradient for the sampled action (assuming log probability), doing back propagation and parameter update encouraging the actions we picked in all those states. And we take the other decisions we made in the losing games and do a negative update, discouraging whatever the controller did. After training, hopefully the network can become slightly more likely to repeat actions that worked, and slightly less likely to repeat actions that failed. We evaluate the policy after training with more than 500,000,000 samples (state and action pairs). However, the performance of the controller is still very poor, not able to return the ball from its opponent constantly. The possible reasons include: (a) we do not follow any demonstrations from experts; (b) we train with a very coarse reward function without a meaningful explanation which parts of trajectories lead to a good strategy and which parts lead to a policy failure.

We then apply Algorithm 1 and Algorithm 2 on this poor policy. We found that, after only collecting approximately 5000 samples in $\mathcal{D}$ using expert trajectories generation, policy training

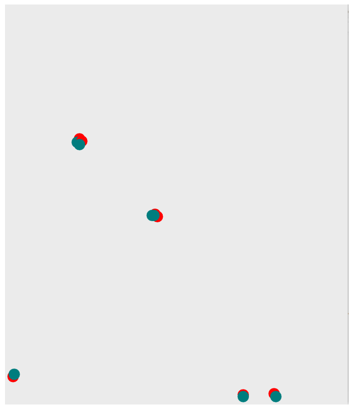


Figure 1: Safety Verification of a Policy

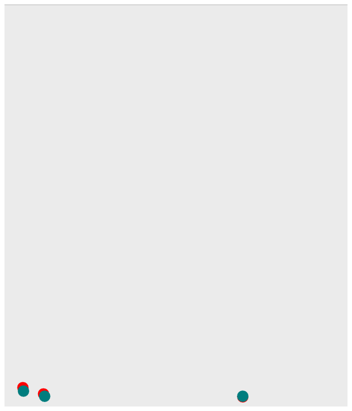


Figure 2: Use Algorithm 3 to improve it

becomes very performant and we do not observe any ball missing from the policy controller after a tremendous number of test runs. Our algorithms therefore enables policy training with five orders of magnitude less data in the Pong game.

Policy Verification Experiment. We apply Algorithm 3 to verify safety of the synthesized policy. The result is depicted in Fig. 1, which is obtained by just performing our safety verification with expert trajectories generated from the above policy training experiment. Each pair of circles in the picture represents a ball location x from expert trajectories and a tiny perturbation to it, say x', where $||x^* - x|| < \eta$ where η is small parameter as reflected in the picture, found by the verification solver, such that the policy chooses different actions on x and x', while the policy controller can only return one of the balls from x and x' in the end. Recall that we have tested the policy, which is obtained from running Algorithm 1 and Algorithm 2, in a large number of test runs and cannot observe a single failure. Formal verification is proved more effective than simple random testing.

Fig. 2 shows the result after applying Algorithm 3 to the policy. It can be observed that the two unsafe locations in the middle part of the UI from Fig. 1 get fixed. For the remaining three locations, Algorithm 3 cannot offer additional policy improvements because the balls are too close to the boarder of the UI and seem impossible to be returned according to the locations of the paddle controllers (which we do not draw for simplifying the demonstration).

References

[1] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[2] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[4] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[5] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robot. Auton. Syst.*, 57(5), May 2009.

[6] Stéphane Ross, Geoffrey J. Gordon, and Drew Bagnell. 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*, pages 627–635, 2011.

[7] Xiaoxiao Guo, Satinder P. Singh, Honglak Lee, Richard L. Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3338–3346, 2014.

[8] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2613–2621, 2016.

[9] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, pages 3–29, 2017.

[10] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, pages 97–117, 2017.

[11] Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. Maximum resilience of artificial neural networks. In *Automated Technology for Verification and Analysis, ATVA 2017, Oct 3-6, 2017, Pune, India*, 2017.