An Inductive Synthesis Framework for Verifiable Machine Learning

He Zhu
Galois, Inc., USA
hezhu@galois.com

Zikang Xikong
Purdue University, USA
xiong84@cs.purdue.edu

Stephen Magill
Galois, Inc., USA
stephen@galois.com

Suresh Jagannathan
Purdue University, USA
suresh@cs.purdue.edu

Abstract

Despite the tremendous advances that have been made in the last decade on developing useful machine-learning applications, their wider adoption has been hindered by the lack of strong assurance guarantees that can be made about their behavior. In this paper, we consider how formal verification techniques developed for traditional software systems can be repurposed for verification of reinforcement learning-enabled ones, a particularly important class of machine learning systems. Rather than enforcing safety by examining and altering the structure of a complex neural network implementation, our technique uses blackbox methods to synthesizes deterministic programs, simpler, more interpretable, approximations of the network that can nonetheless guarantee desired safety properties are preserved, even when the network is deployed in unanticipated or previously unobserved environments. Our methodology frames the problem of neural network verification in terms of a counterexample and syntax-guided inductive synthesis procedure over these programs. The synthesis procedure searches for both a deterministic program and an inductive invariant over an infinite state transition system that represents a specification of an application's control logic. Additional specifications defining environment-based constraints can also be provided to further refine the search space. Synthesized programs deployed in conjunction with a neural network implementation dynamically enforce safety conditions by monitoring and preventing potentially unsafe actions proposed by neural policies. Experimental results over a wide range of cyber-physical applications support our claims that software-inspired formal verification techniques can be used to realize trustworthy machine learning systems with low overhead.

1 Introduction

Neural networks have proven to be a promising software architecture for expressing a variety of machine learning applications. However, non-linearity and stochasticity inherent in their design greatly complicate reasoning about their behavior. Many existing approaches to verifying [14, 19, 24, 30] and testing [36, 38, 41] these systems typically attempt to tackle implementations head-on, reasoning directly over the structure of activation functions, hidden layers, weights, biases, and other kinds of low-level artifacts that are far-removed from the specifications they are intended to satisfy. Moreover, the notion of safety verification that is typically considered in these efforts ignore effects induced by the actual environment in which the network is deployed, significantly weakening the utility of any safety claims that are actually proven. Consequently, effective verification methodologies in this important domain still remains very much an open problem.

To overcome these difficulties, we define a new verification toolchain that reasons about correctness extensionally, using a syntax-guided synthesis framework [4] that generates a simpler and more malleable deterministic program guaranteed to represent a safe control policy of a reinforcement learning (RL)-based neural network, an important class of machine learning systems, commonly used to govern cyber-physical systems such as autonomous vehicles, where high assurance is particularly important. Like other contemporary efforts [7, 40], our synthesis procedure treats the neural network as an oracle, extracting a deterministic program $\mathcal{P}$ intended to approximate the policy actions implemented by the network. Unlike these other efforts, however, which are primarily focussed on interpretability and guided by quantitative objectives, our synthesis procedure is designed with verification in mind, and is thus structured to incorporate formal safety constraints drawn from a logical specification of the control system the network purports to implement, along with additional salient environment properties relevant to the deployment context. To this end, we realize our synthesis procedure via a *counterexample guided inductive synthesis* (CEGIS) loop [4] that eliminates any counterexamples to safety in the synthesized program $\mathcal{P}$. Now, rather than repairing the network directly to satisfy the constraints governing $\mathcal{P}$, we instead treat $\mathcal{P}$ as a safety shield that operates in tandem with the network, overriding network-proposed actions whenever such actions can be shown to lead to a potentially

Draft, January 01–03, 2019, New York, NY, USA
2019.

unsafe state. Unlike existing shielding techniques [3] that inherently only handle finite state systems, our approach naturally generalizes to *infinite state* systems with a *continuous* underlying action space. Taken together, these properties enable safety enforcement of RL-based neural networks without having to suffer a loss in performance to achieve high assurance. We show that over a range of cyber-physical applications defining various kinds of control systems, the overhead of runtime assurance is nominal, less than a few percent, compared to running an unproven, and thus potentially unsafe, network with no shield support.

This paper makes the following contributions:

1. We present a verification toolchain for ensuring that the control policies learned by an RL-based neural network are safe. Our notion of safety is defined in terms of a specification of an infinite state transition system that captures, for example, the system dynamics of a cyber-physical controller.
2. We develop a counterexample-guided inductive synthesis framework that treats the neural control policy as an oracle to guide the search for a simpler deterministic program that approximates the behavior of the network but which is more amenable for verification. The synthesis procedure differs from prior efforts [7, 40] because the search procedure is bounded by safety constraints defined by the specification (aka state transition system) as well as a characterization of specific environment conditions defining the application's deployment context.
3. We use a verification procedure that guarantees actions proposed by the synthesized program always lead to a state consistent with an inductive invariant of the original specification and deployed environment context. This invariant defines an inductive property that separates all reachable (safe) and unreachable (unsafe) states expressible in the transition system.
4. We develop a runtime monitoring framework that treats the synthesized program as a safety shield [3], overriding proposed actions of the network whenever such actions can cause the system to enter into an unsafe region.

We present a detailed experimental study over a wide range of cyber-physical control systems that justify the utility of our approach. These results indicate that the cost of ensuring verification is low, typically on the order of a few percent. The remainder of the paper is structured as follows. In the next section, we present a detailed overview of our approach. Sec. 3 formalizes the problem and the context. Details about the synthesis and verification procedure are given in Sec. 4. A detailed evaluation study is provided in Sec. 5. Related work and conclusions are given in Secs. 6 and 7, resp.

2 Motivation and Overview

To motivate the problem and to provide an overview of our approach, consider the definition of a learning-enabled controller that operates an inverted pendulum. While the specification of this system is simple, it is nonetheless representative of a number of practical control systems, such as Segway transporters and autonomous drones, that have thus far proven difficult to verify, but for which high assurance is very desirable.

2.1 State Transition System

We model an inverted pendulum system as an *infinite* state transition system with *continuous* actions in Fig. 1. Here, the pendulum has mass m and length l. A system state is $s = [\eta, \omega]^T$ where η is the pendulum's angle and ω is its angular velocity. A controller can use a 1-dimensional continuous control action a to maintain the pendulum upright.

Since modern controllers are typically implemented digitally (using digital-to-analog converters for interfacing between the analog system and a digital controller), we assume that the pendulum is controlled in discrete time instants kt where $k = 0, 1, 2, \cdots$, *i.e.*, the controller uses the system dynamics, the change of rate of s, denoted as $\dot{s}$, to transition every t time period, with the conjecture that the control action a is a constant during each discrete time interval. Using Euler's method, for example, a transition from state $s_k = s(kt)$ at time kt to time $kt + t$ is approximated as $s(kt + t) = s(kt) + \dot{s}(kt) \times t$. We specify the change of rate $\dot{s}$ using the differential equation shown in Fig. 1.[1] Intuitively, the control action a is allowed to affect the change of rate of η and ω to balance a pendulum. Thus, small values of a result in small swing and velocity changes of the pendulum, actions that are useful when the pendulum is upright (or nearly so), while large values of a contribute to larger changes in swing and velocity, actions that are necessary when the pendulum enters a state where it may risk losing balance. In case the system dynamics are unknown, we can use known algorithms to infer dynamics from online experiments [1].

Assume the state transition system of the inverted pendulum starts from a set of initial states $\mathcal{S}_0$:

$$\mathcal{S}_0 : \{(\eta, \omega) \mid -20^\circ \leq \eta \leq 20^\circ \wedge -20^\circ \leq \omega \leq 20^\circ\}$$

The global safety property we wish to preserve is that the pendulum never falls down. We define a set of unsafe states of the transition system (colored in yellow in Fig. 1):

$$\mathcal{S}_u : \{(\eta, \omega) \mid \neg(-90^\circ < \eta < 90^\circ \wedge -90^\circ < \omega < 90^\circ)\}$$

We assume the existence of a neural network control policy $\pi_{\mathbf{w}} : \mathbb{R}^2 \to \mathbb{R}$ that executes actions over the pendulum, whose weight values of $\mathbf{w}$ are learned from training episodes. This policy is a state-dependent function, mapping

[1] We derive the control dynamics equations assuming that an inverted pendulum satisfies general Lagrangian mechanics and approximate non-polynomial expressions with their Taylor expansions.

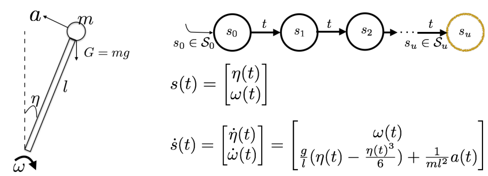


Figure 1. Inverted Pendulum State Transition System.

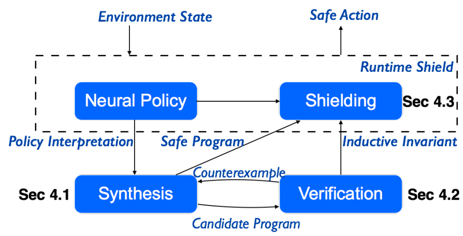


Figure 2. The Framework of Our Approach.

a 2-dimensional state s (η and ω) to a control action a. At each transition, the policy mitigates uncertainty through feedback over state variables in s.

Environment Context. An environment context $C[\cdot]$ defines the behavior of the application, where $[\cdot]$ is left open to deploy a reasonable neural controller π_{w}. The actions of the controller are dictated by constraints imposed by the environment. In its simplest form, the environment is simply a state transition system. In the pendulum example, this would be the equations given in Fig. 1, parameterized by pendulum mass and length. In general, however, the environment may include additional constraints (*e.g.*, a constraining bounding box that restricts the motion of the pendulum beyond the specification given by the transition system in Fig. 1).

2.2 Synthesis, Verification and Shielding

In this paper, we assume that a neural network is trained using a state-of-the-art reinforcement learning strategy [33, 37]. Even though the resulting neural policy may appear to work well in practice, the complexity of its implementation makes it difficult to assert any strong and provable claims about its correctness.

Framework. We construct a *policy interpretation* mechanism to enable verification, inspired by prior work on imitation learning [29, 31] and interpretable machine learning [7, 40]. Fig. 2 depicts the high-level framework of our approach. Our idea is to synthesize a *deterministic policy program* from a neural policy π_{w}, approximating π_{w} (which we call an *oracle*) with a simpler structural program $\mathcal{P}$. Like π_{w}, $\mathcal{P}$ takes as input a system state and generates a control action a. To this end, $\mathcal{P}$ is simulated in the environment used to train the neural policy π_{w}, to collect feasible states. Guided by π_{w}'s actions on such collected states, $\mathcal{P}$ is further improved to resemble π_{w}.

The goal of the synthesis procedure is to search for a deterministic program $\mathcal{P}^*$ satisfying both (1) a quantitative specification such that it bears reasonably close resemblance to its oracle so that allowing it to serve as a potential substitute is a sensible notion, and (2) a desired logical safety property such that when in operation the unsafe states defined in the environment C cannot be reached. Formally,

$$\mathcal{P}^* = \underset{\mathcal{P} \in \text{Safe}(C, [\![\mathcal{H}]\!])}{\arg\max} \; d(\pi_{\text{w}}, \mathcal{P}, C) \tag{1}$$

where $d(\pi_{\text{w}}, \mathcal{P}, C)$ measures proximity of $\mathcal{P}$ with its neural oracle in an environment C; $[\![\mathcal{H}]\!]$ defines a search space for $\mathcal{P}$ with prior knowledge on the shape of target deterministic programs; and, $\text{Safe}(C, [\![\mathcal{H}]\!])$ restricts the solution space to a set of safe programs. A program $\mathcal{P}$ is safe if the safety of the transition system $C[\mathcal{P}]$, the deployment of $\mathcal{P}$ in the environment C, can be *formally verified*.

The novelty of our approach against prior work on neural policy interpretation [7, 40] is thus two-fold:

1. We bake in the concept of safety and formal safety verification into the synthesis of a deterministic program from a neural policy as depicted in Fig. 2. If a candidate program is not safe, we rely on a counterexample-guided inductive synthesis loop to improve our synthesis outcome to enforce the safety conditions imposed by the environment.
2. We allow $\mathcal{P}$ to operate in tandem with the high-performing neural policy. $\mathcal{P}$ can be viewed as capturing an *inductive* invariant of the state transition system, which can be used as a shield to describe a boundary of safe states within which the neural policy is free to make optimal control decisions. If the system is about to be driven out of this safety boundary, the synthesized program is used to take an action that is guaranteed to stay within the space subsumed by the invariant. By allowing the synthesis procedure to treat the neural policy as an oracle, we constrain the search space of feasible programs to be those whose actions reside within a proximate neighborhood of actions undertaken by the neural policy.

Synthesis. Reducing a complex neural policy to a simpler yet safe deterministic program is possible because we do not require other properties from the oracle; specifically, we do not require that the deterministic program precisely mirror the performance of the neural policy. For example, experiments described in [28] show that while a linear-policy controlled robot can effectively stand up, it is unable to learn an efficient walking gait, unlike a sufficiently-trained neural policy. However, if we just care about the safety of the neural network, we posit that a linear reduction can be sufficiently expressive to describe necessary safety constraints. Based on this hypothesis, for our inverted pendulum example, we can explore a *linear* program space from which a deterministic program $\mathcal{P}_\theta$ can be drawn expressed in terms of the following program sketch:

```
def P[θ1, θ2](η, ω): return θ1η + θ2ω
```

Here, $\theta = [\theta_1, \theta_2]$ are unknown parameters that need to be synthesized. Intuitively, the program weights the importance of η and ω at a particular state to provide a feedback control action to mitigate the deviation of the inverted pendulum from ($\eta = 0°, \omega = 0°$).

Our search-based synthesis sets θ to $\mathbf{0}$ initially. It runs the deterministic program $\mathcal{P}_\theta$ instead of the oracle neural policy π_w within the environment C defined in Fig. 1 (in this case, the state transition system represents the differential equation specification of the controller) to collect a batch of trajectories. A run of the state transition system of $C[\mathcal{P}_\theta]$ produces a finite trajectory $s_0, s_1, \cdots, s_T$. We find θ from the following optimization task that realizes (1):

$$\max_{\theta \in \mathbb{R}^2} \mathbb{E}[\Sigma_{t=0}^{T} d(\pi_w, \mathcal{P}_\theta, s_t)] \tag{2}$$

where $d(\pi_w, \mathcal{P}_\theta, s_t) \equiv \begin{cases} -(\mathcal{P}_\theta(s_t) - \pi_w(s_t))^2 & s_t \notin \mathcal{S}_u \\ -MAX & s_t \in \mathcal{S}_u \end{cases}$. This equation aims to search for a program $\mathcal{P}_\theta$ at minimal distance from the neural oracle π_w along sampled trajectories, while simultaneously maximizing the likelihood that $\mathcal{P}_\theta$ is safe.

Our synthesis procedure described in Sec. 4.1 is a random search-based optimization algorithm [23]. We sample a new position of θ iteratively from its hypersphere of a given small radius surrounding the current position of θ and move to the new position (*w.r.t.* a learning rate) as dictated by Equation (2). For the running example, our search synthesizes:

```
def P(η, ω): return −12.05η + −5.87ω
```

The synthesized program can be used to intuitively interpret how the neural oracle works. For example, if a pendulum with a positive angle $\eta > 0$ leans towards the right ($\omega > 0$), the controller will need to generate a large negative control action to force the pendulum to move left.

Verification. Since our method synthesizes a deterministic program $\mathcal{P}$, we can leverage off-the-shelf formal verification algorithms to verify its safety with respect to the state transition system C defined in Fig. 1. To ensure that $\mathcal{P}$ is safe, we must ascertain that it can never transition to an unsafe state, *i.e.*, a state that causes the pendulum to fall. When framed as a formal verification problem, answering such a question is tantamount to discovering an inductive invariant φ that represents all safe states over the state transition system:

1. *Safe:* φ is disjoint with all unsafe states $\mathcal{S}_u$,
2. *Init:* φ includes all initial states $\mathcal{S}_0$,
3. *Induction:* Any state in φ transitions to another state in φ and hence cannot reach an unsafe state.

Inspired by template-based constraint solving approaches on inductive invariant inference [15, 16, 20, 27], the verification algorithm described in Sec. 4.2 uses a constraint solver to look for an inductive invariant in the form of a *convex barrier certificate* [27] $E(s) \leq 0$ that maps all the states in the (safe) reachable set to non-positive reals and all the states in the unsafe set to positive reals. The basic idea is to identify a polynomial function $E : \mathbb{R}^n \rightarrow \mathbb{R}$ such that 1) $E(s) > 0$ for any state $s \in \mathcal{S}_u$, 2) $E(s) \leq 0$ for any state $s \in \mathcal{S}_0$, and 3) $E(s') - E(s) \leq 0$ for any state s that transitions to s' in the state transition system $C[\mathcal{P}]$. The second and third condition collectively guarantee that $E(s) \leq 0$ for any state s in the reachable set, thus implying that an unsafe state in $\mathcal{S}_u$ can never be reached.

Fig. 3(a) draws the discovered invariant in blue for $C[\mathcal{P}]$ given the initial and unsafe states where $\mathcal{P}$ is the synthesized program for the inverted pendulum system. We can conclude that the safety property is satisfied by the $\mathcal{P}$ controlled system as all reachable states do not overlap with unsafe states. In case verification fails, our approach conducts a counterexample guided loop (Sec. 4.2) to iteratively synthesize safe deterministic programs until verification succeeds.

Shielding. Keep in mind that a safety proof of a reduced deterministic program of a neural network does not automatically lift to a safety argument of the neural network from which it was derived since the network may exhibit behaviors not fully captured by the simpler deterministic program. To bridge this divide, we propose to recover soundness at runtime by monitoring system behaviors of a neural network in its actual environment (deployment) context.

Fig. 4 depicts our runtime shielding approach with more details given in Sec. 4.3. The inductive invariant φ learnt for a deterministic program $\mathcal{P}$ of a neural network π_w can serve as a *shield* to protect π_w at runtime under the environment context and safety property used to synthesize $\mathcal{P}$. An observation about a current state is sent to both π_w and the shield. A high-performing neural network is allowed to take any actions it feels are optimal as long as the next state it proposes is still within the safety boundary formally characterized by the inductive invariant of $C[\mathcal{P}]$. Whenever a neural policy proposes an action that steers its controlled system out of the state space defined by the inductive invariant we have learned as part of deterministic program synthesis, our shield can instead take a safe action proposed by the deterministic program $\mathcal{P}$. The action given by $\mathcal{P}$ is guaranteed safe because φ defines an inductive invariant of $C[\mathcal{P}]$; taking the action allows the system to stay within the provably safe region identified by φ. Our shielding mechanism is sound due to formal verification. Because the deterministic program was synthesized using π_w as an oracle, it is expected that the shield derived from the program will not frequently interrupt the neural network's decision, allowing the combined system to perform (close-to) optimally.

In the inverted pendulum example, since the 90° bound given as a safety constraint is rather conservative, we do not expect a well-trained neural network to violate this boundary. Indeed, in Fig. 3(a), though the inductive invariant of the synthesized program defines a substantially smaller state space than what is permissible, in our simulation results,

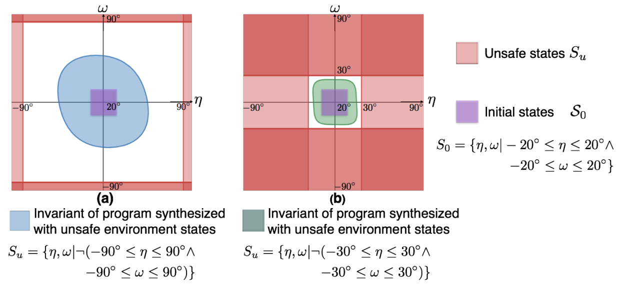


Figure 3. Invariant Inference on Inverted Pendulum.

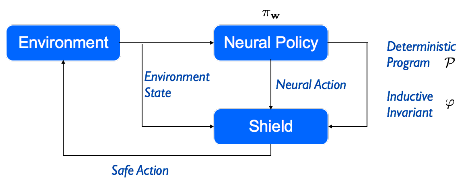


Figure 4. The Framework of Neural Network Shielding.

we find that the neural policy is never interrupted by the deterministic program when governing the pendulum. Note that the non-shaded areas in Fig. 3(a), while appearing safe, presumably define states from which the trajectory of the system can be led to an unsafe state, and would thus not be inductive.

The idea of synthesizing safety shields was previously considered in [3]. Because their verification approach is not symbolic, it can only work over finite discrete state and action systems. Since states in infinite state and continuous action systems are not enumerable, using these methods requires the end user to provide a finite abstraction of a complex environment dynamics; such abstractions are typically too coarse to be useful (because they result in too much over-approximation), or otherwise have too many states to be analyzable [12]. Indeed, automated environment abstraction tools such as [13] often take hours even for simple 4-dimensional systems. We embrace the nature of infinity in control systems by learning a *symbolic* shield from an inductive invariant for a synthesized program that includes an infinite number of environment states under which the program can be guaranteed to be safe. We therefore believe our framework provides a more promising verification pathway for machine learning in high-dimensional control systems.

Handling Different Environment Contexts. Our synthesis approach is critical to ensuring safety when the neural policy is used to predict actions in an environment different from the one used during training. Consider a neural network suitably protected by a shield that now operates safely. The effectiveness of this shield would be greatly diminished if the network had to be completely retrained from scratch whenever it was deployed in a new environment which imposes different safety constraints.

In our running example, suppose we wish to operate the inverted pendulum in an environment such as a Segway transporter in which the model is prohibited from swinging significantly and whose angular velocity must be suitably restricted. We might specify the following new constraints on state parameters to enforce these conditions:

$$\mathcal{S}_u : \{(\eta, \omega) \mid \neg(-30^\circ < \eta < 30^\circ \wedge -30^\circ < \omega < 30^\circ)\}$$

Because of the dependence of a neural network to the quality of training data used to build it, environment changes that deviate from assumptions made at training-time could result in a costly retraining exercise because the network must learn a new way to penalize unsafe actions that were previously safe. However, training a neural network from scratch requires substantial non-trivial effort, involving fine-grained tuning of training parameters or even network structures.

In our framework, the existing network provides a reasonable approximation to the desired behavior. To incorporate the additional constraints defined by the new environment C', we attempt to synthesize a new deterministic program $\mathcal{P}'$ for C', a task based on our experience is substantially easier to achieve than training a new neural network policy from scratch. This new program can be used to protect the original network provided that we can use the aforementioned verification approach to formally verify that $C'[\mathcal{P}']$ is safe by identifying a new inductive invariant φ'. As depicted in Fig. 4, we simply build a new shield $\mathcal{S}'$ that is composed of the program $\mathcal{P}'$ and the safety boundary captured by φ'. The shield $\mathcal{S}'$ can ensure the safety of the neural network in the environment context C' with a strengthened safety condition, despite the fact that the neural network was trained in a different environment context C.

Fig. 3(b) depicts the new unsafe states in C' (colored in red). It extends the unsafe range draw in Fig. 3(a) so the inductive invariant learned there is unsafe for the new one. Our approach synthesizes a new deterministic program for which we learn a more restricted inductive invariant depicted in green in Fig. 3(b). While we provide a detailed experimental study in Sec. 5, we briefly characterize the effectiveness of our shielding approach. Our experiment examined 1000 potential trajectories, each of which is comprised of 5000 simulation steps of the inverted pendulum system with the safety constraints defined by C'. Without the shield $\mathcal{S}'$, the pendulum entered the unsafe region $\mathcal{S}_u$ 41 times. All of these violations were prevented by $\mathcal{S}'$. Notably, the intervention rate of $\mathcal{S}'$ to interrupt the neural network's decision was extremely low. Out of a total of 5000×1000 decisions, we only observed 414 instances (0.00828%) where the shield interfered with (i.e., overrode) the decision made by the network.

Comparison to Directly Learning Deterministic Programs. One may be curious as to why not directly learning a deterministic program to control the device (without appeal to the neural policy), using the same algorithm to train the neural policy. Even for an example as simple as the inverted pendulum, however, we found directly generating a linear control program challenging. For example, a pendulum control action may be required to be within the range $[-1, 1]$ if it needs to operate in an environment with low power constraints. We found that despite many experiments on tuning learning rates and rewards, directly training a linear control program to conform to this restriction was unsuccessful because of undesirable overfitting. In contrast, neural network control policies learn such constraints more effectively because they are better regularized (*i.e.*, are designed to avoid overfitting). Indeed, by treating the neural policy as an oracle, we were able to quickly discover a linear deterministic program that in fact satisfies this additional motion constraint.

3 Problem Setup

We model the context C of a control policy as an environment state transition system $C[\cdot] = (X, \mathcal{A}, \mathcal{S}, \mathcal{S}_0, \mathcal{S}_u, \mathcal{T}_t[\cdot], f, r)$. Note that $\cdot$ is intentionally left open to deploy neural control policies. Here, X is a finite set of variables interpreted over the reals $\mathbb{R}$ and $\mathcal{S} = \mathbb{R}^X$ is the set of all valuations of the variables X. We denote $s \in \mathcal{S}$ to be an n-dimensional environment state and $a \in \mathcal{A}$ to be a control action where $\mathcal{A}$ is an infinite set of m-dimensional continuous actions that a learning-enabled controller can perform. We use $\mathcal{S}_0 \in \mathcal{S}$ to specify a set of initial environment states and $\mathcal{S}_u \in \mathcal{S}$ to specify a set of unsafe environment states where a safe controller should avoid. The transition relation $\mathcal{T}_t[\cdot]$ defines how one state transitions to another given an action by an unknown policy. We assume that $\mathcal{T}_t[\cdot]$ is governed by a standard *differential equation* f defining the relationship between a continuously varying state $s(t)$ and action $a(t)$ and its rate of change $\dot{s}(t)$ over time t:

$$\dot{s}(t) = f(s(t), a(t))$$

We assume f is defined by equations of the form: $\mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ such as the example in Fig. 1. In the following, we often omit the time variable t for simplicity. A deterministic neural network control policy $\pi_{\mathbf{w}} : \mathbb{R}^n \to \mathbb{R}^m$ parameterized by a set of weight values $\mathbf{w}$ is a function mapping an environment state s to a control action a where

$$a = \pi_{\mathbf{w}}(s)$$

By providing a feedback action, a policy can alter the rate of change of state variables to realize optimal system control.

The transition relation $\mathcal{T}_t[\cdot]$ is parameterized by a control policy π that is deployed in C. We explicitly model this deployment as $\mathcal{T}_t[\pi]$. Given a control policy $\pi_{\mathbf{w}}$, we use $\mathcal{T}_t[\pi_{\mathbf{w}}] : \mathcal{S} \times \mathcal{S}$ to specify all possible state transitions allowed by the policy. We assume that a system transitions in discrete time instants kt where $k = 0, 1, 2, \cdots$ and t is a fixed time step ($t > 0$). A state s transitions to a next state s' after time t with the assumption that a control action $a(\tau)$ at time τ is a constant between the time period $\tau \in [0, t)$. Using Euler's method[2], we discretize the continuous dynamics f with finite difference approximation so it can be used in the discretized transition relation $\mathcal{T}_t[\pi_{\mathbf{w}}]$. Then $\mathcal{T}_t[\pi_{\mathbf{w}}]$ can compute these estimates by the following *difference equation*:

$$\mathcal{T}_t[\pi_{\mathbf{w}}] := \{(s, s') \mid s' = s + f(s, \pi_{\mathbf{w}}(s)) \times t\}$$

Environment Disturbance. Our model allows bounded external properties (*e.g.*, additional environment-imposed constraints) by extending the definition of change of rate: $\dot{s} = f(s, a) + d$ where d is a vector of random disturbances. We use d to encode environment disturbances in terms of bounded nondeterministic values. We assume that tight upper and lower bounds of d can be accurately estimated at runtime using multivariate normal distribution fitting methods.

Trajectory. A trajectory h of a state transition system $C[\pi_{\mathbf{w}}]$ which we denote as $h \in C[\pi_{\mathbf{w}}]$ is a sequence of states $s_0, \cdots, s_i, s_{i+1}, \cdots$ where $s_0 \in \mathcal{S}_0$ and $(s_i, s_{i+1}) \in \mathcal{T}_t[\pi_{\mathbf{w}}]$ for all $i \geq 0$. We use $\mathcal{C} \subseteq C[\pi_{\mathbf{w}}]$ to denote a set of trajectories. The reward that a control policy receives on performing an action a in a state s is given by the reward function $r(s, a)$.

3.1 Reinforcement Learning

The goal of reinforcement learning is to maximize the reward that can be collected by a neural control policy in a given environment context C. Such problems can be abstractly formulated as

$$\begin{aligned} &\max_{\mathbf{w} \in \mathbb{R}^n} J(\mathbf{w}) \\ &J(\mathbf{w}) = \mathbb{E}[r(\pi_{\mathbf{w}})] \end{aligned} \tag{3}$$

where $s_0, s_1, \ldots, s_T$ is a trajectory of length T of the state transition system and $r(\pi_{\mathbf{w}}) = \lim_{T \to \infty} \Sigma_{k=0}^{T} r(s_k, \pi_{\mathbf{w}}(s_k))$ is the cumulative reward achieved by the policy $\pi_{\mathbf{w}}$ from this trajectory. Thus this formulation uses simulations of the transition system with finite length rollouts to estimate the expected cumulative reward collected over T time steps and aim to maximize this reward.

Policy Gradient. Control actions are used to mitigate uncertainty through feedback on states, rather than optimizing over deterministic sequences of controls actions. Reinforcement learning thus optimizes over policies, resulting in the following update rule in a policy gradient approach:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \tag{4}$$

where α is a user-specified learning rate and $\nabla_\theta J(\mathbf{w})$ is a policy gradient [33]. Theoretically, policy gradient algorithms converge to at least a local minimum.

[2]Euler's method may sometimes poorly approximate the true system transition function when f is highly nonlinear. More precise higher-order approaches such as *Runge-Kutta* methods exist to compensate for loss of precision in this case.

$$E ::= v \mid x \mid \oplus(E_1, \ldots, E_k)$$
$$\varphi ::= E \leq 0$$
$$\mathcal{P} ::= \textbf{return } E \mid \textbf{if } \varphi \textbf{ then return } E \textbf{ else } \mathcal{P}$$

Figure 5. Syntax of the Policy Programming Language.

3.2 Safety Verification in Reinforcement Learning

Since reinforcement learning only considers finite length rollouts, we are wish to determine if a control policy is safe to use under an infinite time horizon. Given a state transition system, the safety verification problem is concerned with verifying that no trajectories contained in $\mathcal{S}$ starting from an initial state in $\mathcal{S}_0$ reach an unsafe state in $\mathcal{S}_u$. However, a neural network is a representative of a class of deep and sophisticated models that challenges the capability of the state-of-the-art verification techniques. This level of complexity is exacerbated in our work because we consider the long term safety of a neural policy deployed within a nontrivial environment context C that in turn is described by a complex infinite-state transition system.

4 Verification Procedure

To verify a neural network control policy π_w with respect to an environment context C, we first synthesize a *deterministic policy program* $\mathcal{P}$ from the neural policy. We require that $\mathcal{P}$ both (a) broadly resemble its neural oracle and (b) additionally satisfies a desired safety property when it is deployed in C. We conjecture that a safety proof of $C[\mathcal{P}]$ is easier to construct than that of $C[\pi_w]$. More importantly, we leverage the safety proof of $C[\mathcal{P}]$ to ensure the safety of $C[\pi_w]$.

4.1 Synthesis

Fig. 5 defines a search space for a deterministic policy program $\mathcal{P}$ where E and φ represent the basic syntax of (polynomial) program expressions and inductive invariants, respectively. Here v ranges over a universe of numerical constants, x represents variables, and $\oplus$ is a basic operator including $+$ and $\times$. A deterministic program $\mathcal{P}$ also features conditional statements using φ as branching predicates.

We allow the user to define a sketch [34, 35] to describe the shape of target policy programs using the grammar in Fig. 5. We use $\mathcal{P}[\theta]$ to represent a sketch where θ represents unknowns that need to be filled-in by the synthesis procedure. We use $\mathcal{P}_\theta$ to represent a synthesized program with known values of θ. Similarly, the user can define a sketch of an inductive invariant $\varphi[\cdot]$ that is used (in Sec. 4.2) to learn a safety proof to verify a synthesized program $\mathcal{P}_\theta$.

We do not require the user to explicitly define conditional statements in a program sketch. Our synthesis algorithm uses verification counterexamples to lazily add branch predicates φ under which a program performs different computations

Algorithm 1: Synthesize $(\pi_w, \mathcal{P}[\theta], C[\cdot])$

$\theta \leftarrow \mathbf{0}$;
do
  Sample δ from a zero mean Gaussian vector;
  Sample a set of trajectories C_1 using $C[\mathcal{P}_{\theta+\nu\delta}]$;
  Sample a set of trajectories C_2 using $C[\mathcal{P}_{\theta-\nu\delta}]$;
  $\theta \leftarrow \theta + \alpha[\frac{d(\pi_w, \mathcal{P}_{\theta+\nu\delta}, C_1) - d(\pi_w, \mathcal{P}_{\theta-\nu\delta}, C_2)}{\nu}]\delta$;
while θ *is not converged*;
return $\mathcal{P}_\theta$

depending on whether φ evaluates to true or false. The end user simply writes a sketch over basic expressions. For example, a sketch that defines a family of linear function over a collection of variables can be expressed as:

$$\mathcal{P}[\theta](X) ::= \textbf{return } \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n + \theta_{n+1} \quad (5)$$

Here $X = (x_1, x_2, \cdots)$ are system variables in which the coefficient constants $\theta = (\theta_1, \theta_2, \cdots)$ are unknown.

The goal of our synthesis algorithm is to find unknown values of θ that maximize the likelihood that $\mathcal{P}_\theta$ resembles the neural oracle π_w while still being a safe program with respect to the environment context C:

$$\theta^* = \arg\max_{\theta \in \mathbb{R}^n} d(\pi_w, \mathcal{P}_\theta, C) \quad (6)$$

where d measures the *distance* between the outputs of the estimate program $\mathcal{P}_\theta$ and the neural policy π_w subject to safety constraints. To avoid the computational complexity that arises if we consider a solution to this equation analytically as an optimization problem, we instead approximate $d(\pi_w, \mathcal{P}_\theta, C)$ by randomly sampling a set of trajectories C that are encountered by $\mathcal{P}_\theta$ in the environment state transition system $C[\mathcal{P}_\theta]$:

$$d(\pi_w, \mathcal{P}_\theta, C) \approx d(\pi_w, \mathcal{P}_\theta, C) \ s.t.\ C \subseteq C[\mathcal{P}_\theta]$$

We estimate θ^* using these sampled trajectories C and define:

$$d(\pi_w, \mathcal{P}_\theta, C) = \sum_{h \in C} d(\pi_w, \mathcal{P}_\theta, h)$$

Since each trajectory $h \in C$ is a finite rollout $s_0, \ldots, s_T$ of length T, we have:

$$d(\pi_w, \mathcal{P}_\theta, h) = \sum_{t=0}^{T} \begin{cases} -\,\|(\mathcal{P}_\theta(s_t) - \pi_w(s_t))\| & s_t \notin \mathcal{S}_u \\ -MAX & s_t \in \mathcal{S}_u \end{cases}$$

where $\|\cdot\|$ is a suitable norm. As illustrated in Sec. 2.2, we aim to minimize the distance between a synthesized program $\mathcal{P}_\theta$ from a sketch space and its neural oracle along sampled trajectories encountered by $\mathcal{P}_\theta$ in the environment context but put a large penalty on states that are unsafe.

Random Search. We implement the idea encoded in equation (6) in Algorithm 1 that depicts the pseudocode of our policy interpretation approach. It take as inputs a neural policy, a policy program sketch parameterized by θ, and an environment state transition system and outputs a synthesized policy program $\mathcal{P}_\theta$.

An efficient way to solve equation (6) is to directly perturb θ in the search space by adding random noise and then update θ based on the effect on this perturbation [23]. We choose a direction uniformly at random on the sphere in parameter space, and then optimize the goal function along this direction. To this end, in line 3 of Algorithm 1, we sample Gaussian noise δ to be added to policy parameters θ in both directions where ν is a small positive real number. In line 4 and line 5, we sample trajectories C_1 and C_2 from the state transition systems obtained by running the perturbed policy program $\mathcal{P}_{\theta+\nu\delta}$ and $\mathcal{P}_{\theta-\nu\delta}$ in the environment context C.

We evaluate the proximity of these two policy programs to the neural oracle and, in line 6 of Algorithm 1, to improve the policy program, we also optimize equation (6) by updating θ with a finite difference approximation along the direction:

$$\theta \leftarrow \theta + \alpha\left[\frac{d(\mathcal{P}_{\theta+\nu\delta}, \pi_{\mathrm{w}}, C_1) - d(\mathcal{P}_{\theta-\nu\delta}, \pi_{\mathrm{w}}, C_2)}{\nu}\right]\delta \qquad (7)$$

where α is a predefined learning rate. Such an update increment corresponds to an unbiased estimator of the gradient of θ [26] for equation (6). The algorithm iteratively updates θ until convergence.

4.2 Verification

A synthesized policy program $\mathcal{P}$ is verified with respect to an environment context given as an infinite state transition system defined in Sec. 3: $C[\mathcal{P}] = (X, \mathcal{A}, \mathcal{S}, \mathcal{S}_0, \mathcal{S}_u, \mathcal{T}_t[\mathcal{P}], f, r)$. Our verification algorithm learns an inductive invariant φ over the transition relation $\mathcal{T}_t[\mathcal{P}]$ formally proving that all possible system behaviors are encapsulated in φ and φ is required to be disjoint with all unsafe states $\mathcal{S}_u$.

We follow template-based constraint solving approaches for inductive invariant inference [15, 16, 20, 27] to discover this invariant. The basic idea is to identify a function $E : \mathbb{R}^n \to \mathbb{R}$ that serves as a "barrier" [16, 20, 27] between reachable system states (evaluated to be nonpositive by E), and unsafe states (evaluated positive by E). Using the invariant syntax in Fig. 5, the user can define an invariant sketch

$$\varphi[c](X) ::= E[c](X) \leq 0 \qquad (8)$$

over variables X and c unknown coefficients intended to be synthesized. Fig. 5 carefully restricts that an invariant sketch $E[c]$ can only be postulated as a polynomial function as there exist efficient SMT solvers [11] and constraint solvers [22] for nonlinear polynomial reals. Formally, assume real coefficients $c = (c_1, \cdots, c_m)$ are used to parameterize $E[c]$ in an affine manner:

$$E[c](X) = \Sigma_{i=1}^{p} c_i b_i(X)$$

where the $b_i(X)$'s are some monomials in variables X. As a heuristic, the user can simply determine an upper bound on the degree of $E[c]$, and then include all monomials whose degrees are no greater than the bound in the sketch. Large values of the bound enable verification of more complex safety conditions, but impose greater demands on the constraint solver; small values capture coarser safety properties, but are easier to solve.

Example 4.1. Consider the inverted pendulum system in Sec. 2.2. To discover an inductive invariant for the transition system, the user might choose to define an upper bound of 4, which results in the following polynomial invariant sketch: $\varphi[c](\eta, \omega) ::= E[c](\eta, \omega) \leq 0$ where

$$E[c](\eta, \omega) = c_0\eta^4 + c_1\eta^3\omega + c_2\eta^2\omega^2 + c_3\eta\omega^3 + c_4\omega^4 + c_5\eta^3 + \cdots + c_n$$

The sketch includes all monomials over η and ω, whose degrees are no greater than 4. The coefficients $c = [c_0, \cdots, c_n]$ are unknown and need to be synthesized.

To synthesize these unknowns, we require that $E[c]$ must satisfy the following verification conditions:

$$\forall (s) \in \mathcal{S}_u \qquad E[c](s) > 0 \qquad (9)$$

$$\forall (s) \in \mathcal{S}_0 \qquad E[c](s) \leq 0 \qquad (10)$$

$$\forall (s, s') \in \mathcal{T}_t[\mathcal{P}] \qquad E[c](s') - E[c](s) \leq 0. \qquad (11)$$

We claim that $\varphi ::= E[c](x) \leq 0$ defines an inductive invariant because verification condition (10) ensures that any initial state $s_0 \in S_0$ satisfies φ since $E[c](s_0) \leq 0$; verification condition (11) asserts that along the transition from a state $s \in \varphi$ (so $E[c](s) \leq 0$) to a resulting state s', $E[c]$ cannot become positive so s' satisfies φ as well. Finally, according to verification condition (9), φ does not include any unsafe state $s_u \in \mathcal{S}_u$ as $E[c](s_u)$ is positive.

Verification conditions (9) (10) (11) are polynomial over reals. Synthesizing unknown coefficients can be left to an SMT solver [11] after universal quantifiers are eliminated using a variant of Farkas Lemma as in [16]. However, observe that $E[c]$ is convex.[3] We can gain efficiency by finding unknown coefficients c using off-the-shelf convex constraint solvers following [27]. Encoding verification conditions (9) (10) (11) as polynomial inequalities, we search c that can prove non-negativity of these constraints via an efficient and convex sum of squares programming solver [22]. Additional technical details are provided in the supplementary material.

Counterexample-guided Inductive Synthesis (CEGIS). Given the environment state transition system $C[\mathcal{P}]$ deployed with a synthesized program $\mathcal{P}$, the verification approach above can compute an inductive invariant over the state transition system or tell if there is no feasible solution in the given set of candidates. Note however that the latter does not necessarily imply that the system is unsafe.

Since our goal is to learn a safe deterministic program from a neural network, we develop a counterexample guided inductive program synthesis approach. A CEGIS algorithm in our context is challenging because safety verification is

[3] For arbitrary $E_1(x)$ and $E_2(x)$ satisfying the verification conditions and $\alpha \in [0, 1]$, $E(x) = \alpha E_1(x) + (1 - \alpha)E_2(x)$ satisfies the conditions as well.

Algorithm 2: CEGIS (π_{w}, $\mathcal{P}[\theta]$, $C[\cdot]$)

```
1  policies ← ∅;
2  covers ← ∅;
3  while C.S0 ⊈ covers do
4      search s0 such that s0 ∈ C.S0 ∧ s0 ∉ covers;
5      r* ← Diameter(C.S0);
6      do
7          φ_bound ← {s | s ∈ {s0 − r*, s0 + r*}};
8          C̃ ← C where C̃.S0 = (C.S0 ∩ φ_bound);
9          θ ← Synthesize(π_w, P[θ], C̃[·]);
10         φ ← Verify(C̃[P_θ]);
11         if φ is False then
12             r* ← r*/2;
13         else
14             covers ← covers ∪ {s | φ(s)};
15             policies ← policies ∪ (P_θ, φ);
16             break;
17     while True;
18 end
19 return policies
```

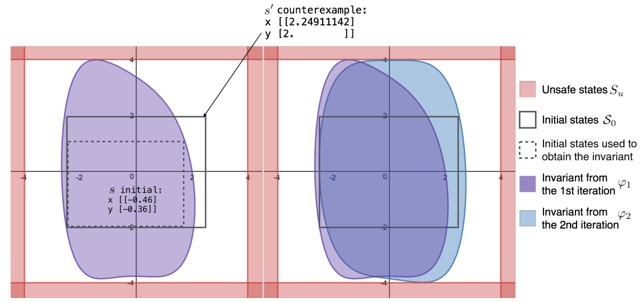
necessarily incomplete, and may not be able to produce a counterexample that serves as an explanation for why a verification attempt is unsuccessful.

We solve the incompleteness challenge by leveraging the fact that we can simultaneously synthesize and verify a program. Our CEGIS approach is given in Algorithm 2. A counterexample is an initial state on which our synthesized program is not yet proved safe. Driven by these counterexamples, our algorithm synthesizes a set of programs from a sketch along with the conditions under which we can switch from from one synthesized program to another.

Algorithm 2 takes as input a neural policy π_{w}, a program sketch $\mathcal{P}[\theta]$ and an environment context C. It maintains synthesized policy programs in policies in line 1, each of which is inductively verified safe in a partition of the universe state space that is maintained in covers in line 2. For soundness, the state space covered by such partitions must be able to include all initial states, checked in line 3 of Algorithm 2 by an SMT solver. In the algorithm, we use $C.\mathcal{S}_0$ to access a field of C such as its initial state space.

The algorithm iteratively samples a counterexample initial state s_0 that is currently not covered by covers in line 4. Since covers is empty at the beginning, this choice is uniformly random initially; we synthesize a presumably safe policy program in line 9 of Algorithm 2 that resembles the neural policy π_{w} considering all possible initial states $\mathcal{S}_0$ of the given environment model C, using Algorithm 1.

If verification fails in line 11, our approach simply reduces the initial state space, hoping that a safe policy program is easier to synthesize if only a subset of initial states are considered. The algorithm in line 12 gradually shrinks the radius



(a)The synthesized program from the 1st CEGIS iteration has a counterexample.
(b)The synthesized program from the 2nd CEGIS iteration is safe.

Figure 6. CEGIS for Verifiable Machine Learning.

r^* of the initial state space around the sampled initial state s_0. The synthesis algorithm in the next iteration synthesizes and verifies a candidate using the reduced initial state space. The main idea is that if the initial state space is shrunk to a restricted area around s_0 but a safety policy program still cannot be found, it is quite possible that either s_0 points to an unsafe initial state of the neural oracle or the sketch is not sufficiently expressive.

If verification at this stage succeeds with an inductive invariant φ, a new policy program $\mathcal{P}_\theta$ is synthesized that can be verified safe in the state space covered by φ. We add the inductive invariant and the policy program into covers and policies in line 14 and 15 respectively and then move to another iteration of counterexample-guided synthesis. This iterative *synthesize-and-verify* process continues until the entire initial state space is covered (line 3 to 18). The output of Algorithm 2 is $[(\mathcal{P}_{\theta 1}, \varphi_1), (\mathcal{P}_{\theta 2}, \varphi_2), \cdots]$ that essentially defines conditional statements in a synthesized program performing different actions depending on whether a specified invariant condition evaluates to true or false.

Theorem 4.2. *If* $CEGIS(\pi_w, \mathcal{P}[\theta], C[\cdot]) = [(\mathcal{P}_{\theta 1}, \varphi_1), (\mathcal{P}_{\theta 2}, \varphi_2), \cdots]$ *(as defined in Algorithm 2), then the deterministic program* $\mathcal{P}$*:*

```
λX.if φ1(X): return Pθ1(X) else if φ2(X): return Pθ2(X) ···
```

is safe in the environment C *meaning that* $\varphi_1 \vee \varphi_2 \vee \cdots$ *is an inductive invariant of* $C[\mathcal{P}]$ *proving that* $C.\mathcal{S}_u$ *is unreachable.*

Example 4.3. We illustrate the proposed counterexample guided inductive synthesis method by means of a simple example, the Duffing oscillator [18], a nonlinear second-order environment. The transition relation of the environment system C is described with the differential equation:

$$\dot{x} = y$$
$$\dot{y} = -0.6y - x - x^3 + a$$

where x, y are the state variables and a the continuous control action given by a well-trained neural feedback control policy π such that $a = \pi(x, y)$. The control objective is to regulate the state to the origin from a set of initial states $C.\mathcal{S}_0 : \{x, y \mid -2.5 \leq x \leq 2.5 \wedge -2 \leq y \leq 2\}$. To be safe, the controller must be able to avoid a set of unsafe states $C.\mathcal{S}_u : \{x, y \mid \neg(-5 \leq x \leq 5 \wedge -5 \leq y \leq 5)\}$. Given a program sketch as in the pendulum example, that is $\mathcal{P}[\theta](x, y) ::= \theta_1 x + \theta_2 y$, the user can ask the constraint solver to reason over a small (say 4) order polynomial invariant sketch for a synthesized program as in Example 4.1.

Algorithm 2 initially samples an initial state s as $\{x = -0.46, y = -0.36\}$ from $\mathcal{S}_0$. The inner do-while loop of the algorithm can discover a sub-region of initial states in the dotted box of Fig. 6(a) that can be leveraged to synthesize a verified deterministic policy $\mathcal{P}_1(x, y) ::= 0.39x - 1.41y$ from the sketch. We also obtain an inductive invariant showing that the synthesized policy can always maintain the controller within the invariant set drawn in purple in Fig. 6(a): $\varphi_1 \equiv 20.9x^4 + 2.9x^3y + 1.4x^2y^2 + 0.4xy^3 + 29.6x^3 + 20.1x^2y + 11.3xy^2 + 1.6y^3 + 25.2x^2 + 39.2xy + 53.7y^2 - 680 \leq 0$.

This invariant explains why the initial state space used for verification does not include the entire $C.\mathcal{S}_0$: a counterexample initial state $s' = \{x = 2.249, y = 2\}$ is not covered by the invariant for which the synthesized policy program above is not verified safe. The CEGIS loop in line 3 of Algorithm 2 uses s' to synthesize another deterministic policy $\mathcal{P}_2(x, y) ::= 0.88x - 2.34y$ from the sketch whose learned inductive invariant is depicted in blue in Fig. 2(b): $\varphi_2 \equiv 12.8x^4 + 0.9x^3y - 0.2x^2y^2 - 5.9x^3 - 1.5xy^2 - 0.3y^3 + 2.2x^2 + 4.7xy + 40.4y^2 - 619 \leq 0$. Algorithm 2 then terminates because $\varphi_1 \vee \varphi_2$ covers $C.\mathcal{S}_0$. Our system interprets the two synthesized deterministic policies as the following deterministic program $\mathcal{P}_{oscillator}$ using the syntax in Fig. 5:

```
def P_oscillator (x, y):
  if 20.9x^4 + 2.9x^3y + 1.4x^2y^2 + ··· + 53.7y^2 − 680 ≤ 0:  # φ1
    return 0.39x − 1.41y
  else if 12.8x^4 + 0.9x^3y − 0.2x^2y^2 + ··· + 40.4y^2 − 619 ≤ 0:  # φ2
    return 0.88x − 2.34y
  else abort # unsafe
```

Neither of the two deterministic policies enforce safety by themselves on all initial states but do guarantee safety when combined together because by construction, $\varphi = \varphi_1 \vee \varphi_2$ is an inductive invariant of $\mathcal{P}_{oscillator}$ in the environment C.

4.3 Shielding

A safety proof of a synthesized deterministic program of a neural network does not automatically lift to a safety argument of the neural network from which it was derived since the network may exhibit behaviors not captured by the simpler deterministic program. To bridge this divide, we recover soundness at runtime by monitoring system behaviors of a neural network in its environment context by using the synthesized policy program and its inductive invariant as a shield. The pseudo-code for using a shield is given in Algorithm 3. In line 1 of Algorithm 3, for a current state s, we use the state transition system of our environment context model to predict the next state s'. If s' is not within φ, we are unsure whether entering s' would inevitably make the system unsafe as we lack a proof that the neural oracle is safe. However, we do have a guarantee that if we follow the synthesized program $\mathcal{P}$, the system would stay within the safety boundary defined by φ that Algorithm 2 has formally proved. We do so in line 3 of Algorithm 3, using $\mathcal{P}$ and φ as shields, intervening only if necessary so as to restrict the shield from unnecessarily intervening the neural policy. [4]

Algorithm 3: Shield (s, $C[\pi_w]$, $\mathcal{P}$, φ)

1 Predict s' such that $(s, s') \in C[\pi_w].\mathcal{T}_t(s)$;
2 **if** $\varphi(s')$ **then return** $\pi_w(s)$;
3 **else return** $\mathcal{P}(s)$;

5 Experimental Results

We have applied our framework on a number of challenging control- and cyberphysical-system benchmarks. We use the deep policy gradient algorithm [33] for neural network training, the Z3 SMT solver [11] to check convergence of the CEGIS loop, and the Mosek constraint solver [5] to generate inductive invariants of synthesized programs from a sketch. All of our benchmarks are verified using the program sketch defined in equation (5) and the invariant sketch defined in equation (8). We report simulation results on our benchmarks over 1000 runs (each run consists of 1000 simulation steps). Each simulation time step is fixed 0.01 second. Our experiments were conducted on a standard desktop machine consisting of Intel(R) Core(TM) i7-8700 CPU cores and 64GB memory.

Safety Verification. We first consider the utility of our approach for verifying the safety of trained neural network controllers. Our results are given in Table 1. In the table, **Vars** represents the number of variables in a control system - this number serves as a proxy for application complexity; **Size** the number of neurons in hidden layers of the network; the **Training** time for the network; and, its **Failures**, the number of times the network failed to satisfy the safety property in simulation. The table also gives the **Size** of a synthesized program in term of the number of polices found by Algorithm 2 (used to generate conditional statements in the program); its **Synthesis** time; the **Overhead** of our approach in terms of the additional cost (compared to the non-shielded variant) in running time to use a shield; and, the number of **Shield Interventions**, the number of times the shield was invoked across all simulation runs.

The first five benchmarks are linear time-invariant control systems adapted from [12]. The safety property is that the

[4] We also extend our approach to synthesize deterministic programs which can guarantee stability. The details are given in the supplementary material.

Table 1. Experimental Results on Deterministic Program Synthesis, Verification and Shielding.

Benchmarks	Vars	Neural Network			Deterministic Program as Shield			
		Size	Training	Failures	Size	Synthesis	Overhead	Shield Interventions
Satellite	2	240×200	957s	0	1	160s	3.37%	0
DCMotor	3	240×200	944s	0	1	68s	2.03%	0
Tape	3	240×200	980s	0	1	42s	2.63%	0
Magnetic Pointer	3	240×200	992s	0	1	85s	2.92%	0
Suspension	4	240×200	960s	0	1	41s	8.71%	0
Biology	3	240×200	978s	0	1	168s	5.23%	0
Data Center Cooling	3	240×200	968s	0	1	168s	4.69%	0
Quadcopter	2	300×200	990s	182	2	67s	6.41%	185
Pendulum	2	240×200	962s	60	3	1107s	9.65%	65
CartPole	4	300×200	990s	47	4	998s	5.62%	1799
Self-Driving	4	300×200	990s	61	1	185s	4.66%	236
Lane Keeping	4	240×200	895s	36	1	183s	8.65%	64
4-Car platoon	8	$500 \times 400 \times 300$	1160s	8	4	609s	3.17%	8
8-Car platoon	16	$500 \times 400 \times 300$	1165s	40	1	1217s	6.05%	1080
Oscillator	18	240×200	1023s	371	1	618s	21.31%	93703

Table 2. Experimental Results on Handling Environment Changes.

Benchmarks	Environment Change	Neural Network		Deterministic Program as Shield			
		Size	Failures	Size	Synthesis	Overhead	Shield Interventions
Cartpole	Increased Pole length by 0.15m	1200×900	3	1	239s	2.91%	8
Pendulum	Increased Pendulum mass by 0.3kg	1200×900	77	1	581s	8.11%	8748
Pendulum	Increased Pendulum length by 0.15m	1200×900	76	1	483s	6.53%	7060
Self-driving	Added an obstacle that must be avoided	1200×900	106	1	392s	8.91%	9255

reach set has to be within a safe rectangle. Benchmark Biology defines a minimal model of glycemic control in diabetic patients such that the dynamics of glucose and insulin interaction in the blood system are defined by polynomials [8]. For safety, we verify that the neural controller ensures that the level of plasma glucose concentration is above a certain threshold. Benchmark Data Center Cooling is a model of a collection of three server racks each with their own cooling devices and they also shed heat to their neighbors. The safety property is that a learned controller must keep the data center below a certain temperature. In these benchmarks, the cost to query the network oracle constitutes the dominant time for generating the safety shield in these benchmarks. Given the simplicity of these benchmarks, the neural network controllers did not violate the safety condition in our trials, and moreover there were no interventions from the safety shields that affected performance.

The next three benchmarks Quadcopter, (Inverted) Pendulum and Cartpole are selected from classic control applications and have more sophisticated safety conditions. We have discussed the inverted pendulum example at length earlier. The Quadcopter environment tests whether a controlled quadcopter can realize stable flight. The environment of Cartpole consists of a pole attached to an unactuated joint connected to a cart that moves along a frictionless track. The system is unsafe when the pole's angle is more than 30° from being upright or the cart moves by more than 0.3 meters from the origin. We observed safety violations in each of these benchmarks that were eliminated using our verification methodology. Notably, the number of interventions is remarkably low, as a percentage of the overall number of simulation steps.

Benchmark Self-driving defines a single car navigation problem. The neural controller is responsible for preventing the car from veering into canals found on either side of the road. Benchmark Lane Keeping models another safety-related car-driving problem. The neural controller aims to maintain a vehicle between lane markers and keep it centered in a possibly curved lane. The curvature of the road is considered as a disturbance input to the model. Benchmarks n-Car platoon model multiple (n) vehicles forming a platoon, maintaining a safe relative distance among one another [32]. Each of these benchmarks exhibited some number of violations that were remediated by our verification methodology. Benchmark Oscillator consists of a two-dimensional switched oscillator plus a 16-order filter. The filter smoothens the input signals and has a single output signal. We verify that the output signal is below a safe threshold. Because of the model complexity of this benchmark, it exhibited significantly more violations than the others. Indeed, the neural-network controlled system often oscillated between the safe and unsafe boundary in many runs. Consequently, the overhead in this benchmark is high because a large number of shield interventions was required to ensure safety. In other words, the synthesized shield trades performance for safety to guarantee that the threshold boundary is never violated.

For all benchmarks, our tool successfully generated safe interpretable deterministic programs and inductive invariants as shields. When a neural controller takes an unsafe action, the synthesized shield correctly prevents this action from executing by providing an alternative provable safe action proposed by the verified deterministic program.

Handling Environment Changes. We consider the effectiveness of our tool when previously trained neural network controllers are deployed in environment contexts different from the environment used for training. Here we consider neural network controllers of larger size (two hidden layers with 1200×900 neurons) than in the above experiments. This is because we want to ensure that a neural policy is trained to be near optimal in the environment context used for training. These larger networks were in general more difficult to train, requiring at least 1500 seconds to converge.

Our results are summarized in Table 2. For Cartpole, we simulated the trained controller in a new environment by increasing the length of the pole by 0.15 meters. The neural controller failed 3 times in our 1000 episode simulation; the shield interfered with the network operation only 8 times to prevent these unsafe behaviors. The new shield was synthesized in 239s significantly faster than retraining a new neural network for the new environment. For (Inverted) Pendulum, we deployed the trained neural network in an environment in which the pendulum's mass is increased by 0.3kg. The neural controller exhibits noticeably higher failure rates than in the previous experiment; we were able to synthesize a safety shield adapted to this new environment in 581 seconds that prevented these violations. The shield intervened with the operation of the network only 8.7 number of times per episode. Similar results were observed when we increased the pendulum's length by 0.15m. For Self-driving, we additionally required the car to avoid an obstacle. The synthesized shield provided safe actions to ensure collision-free motion.

6 Related Work

Verification of Neural Networks. While neural networks have historically found only limited application in safety- and security-related contexts, recent advances have defined suitable verification methodologies that are capable of providing stronger assurance guarantees. For example, Reluplex [19] is an SMT solver that supports linear real arithmetic with ReLU constraints and has been used to verify safety properties in a collision avoidance system. AI^2 [14, 24] is an abstract interpretation tool that can reason about robustness specifications for deep convolutional networks. Robustness verification is also considered in [6, 17]. Systematic testing techniques such as [36, 38, 41] are designed to automatically generate test cases to increase a coverage metric, *i.e.*, explore different parts of neural network architecture by generating test inputs that maximize the number of activated neurons. These approaches ignore effects induced by the actual environment in which the network is deployed. Unlike these whitebox efforts that reason over the architecture of the network, our verification framework is fully blackbox, using a network only as an oracle to learn a simpler deterministic program. This gives us the ability to reason about safety entirely in terms of the synthesized program, using a shielding mechanism derived from the program to ensure that the neural policy can only explore safe regions.

Safe Reinforcement Learning. The machine learning community has explored various techniques to develop safe reinforcement machine learning algorithms in contexts similar to ours, *e.g.*, [25, 39]. In some cases, verification is used as a safety validation oracle [2, 9]. In contrast to these efforts, our inductive synthesis algorithm can be freely incorporated into any existing reinforcement learning framework and applied to any legacy machine learning model. More importantly, our design allows us to synthesize new shields from existing ones, without requiring retraining of the network, under reasonable changes to the environment.

Syntax-Guided Synthesis for Machine Learning. An important reason underlying the success of program synthesis is that sketches of the desired program [34, 35], often provided along with example data, can be used to effectively restrict a search space and allow users to provide additional insight about the desired output [4]. Our sketch based inductive synthesis approach is a realization of this idea applicable to continuous and infinite state control systems central to many machine learning tasks. A high-level policy language grammar is used in our approach to constrain the shape of possible synthesized deterministic policy programs.

In a similar vein, recent efforts on interpretable machine learning generate interpretable models such as program code [40] or decision trees [7] as output, after which traditional symbolic verification techniques can be leveraged to prove program properties. Our approach novelly ensures that only safe programs are synthesized via a CEGIS procedure and can provide safety guarantees on the original high-performing neural networks via invariant inference. A detailed comparison was provided on page 3.

Runtime Shielding. Shield synthesis was widely used in runtime enforcement for reactive systems [10, 21]. Safety shielding in reinforcement learning was firstly introduced in [3], and a comparison of this technique with ours is given on page 5. In general, these early efforts focused on shielding of discrete and finite systems with no obvious generalization to effectively deal with continuous and infinite systems that our approach can monitor and protect.

7 Conclusions

This paper presents an inductive synthesis-based toolchain that can verify neural network control policies within an environment formalized as an infinite-state transition system. The key idea is a novel synthesis framework capable of synthesizing a deterministic policy program based on an user-given sketch that resembles a neural policy oracle and simultaneously satisfies a safety specification using a counterexample-guided synthesis loop. Verification soundness is achieved at runtime by using the learned deterministic program along with its learned inductive invariant as a shield to protect the neural policy, correcting the neural policy's action only if the chosen action can cause violation of the inductive invariant. Experimental results demonstrate that our approach can be used to realize fully trustworthy machine learning systems with low overhead.

References

[1] M. Ahmadi, C. Rowley, and U. Topcu. 2018. Control-Oriented Learning of Lagrangian and Hamiltonian Systems. In *American Control Conference*.

[2] Anayo K. Akametalu, Shahab Kaynama, Jaime F. Fisac, Melanie Nicole Zeilinger, Jeremy H. Gillula, and Claire J. Tomlin. 2014. Reachability-based safe learning with Gaussian processes. In *53rd IEEE Conference on Decision and Control, CDC 2014*. 1424–1431.

[3] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. 2018. Safe Reinforcement Learning via Shielding. *AAAI* (2018).

[4] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*. NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 40. IOS Press, 1–25.

[5] MOSEK ApS. 2018. *The mosek optimization software*. http://www.mosek.com

[6] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya V. Nori, and Antonio Criminisi. 2016. Measuring Neural Net Robustness with Constraints. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. 2621–2629.

[7] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. 2018. Verifiable Reinforcement Learning via Policy Extraction. *CoRR* abs/1805.08328 (2018).

[8] Richard N Bergman, Diane T Finegood, and Marilyn Ader. 1985. Assessment of insulin sensitivity in vivo. *Endocrine reviews* 6, 1 (1985), 45–86.

[9] Felix Berkenkamp, Matteo Turchetta, Angela P. Schoellig, and Andreas Krause. 2017. Safe Model-based Reinforcement Learning with Stability Guarantees. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*. 908–919.

[10] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. 2015. Shield Synthesis: - Runtime Enforcement for Reactive Systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015*. 533–548.

[11] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. 337–340.

[12] Chuchu Fan, Umang Mathur, Sayan Mitra, and Mahesh Viswanathan. 2018. Controller Synthesis Made Real: Reach-Avoid Specifications and Linear Dynamics. In *Computer Aided Verification - 30th International Conference, CAV 2018*. 347–366.

[13] Ioannis Filippidis, Sumanth Dathathri, Scott C. Livingston, Necmiye Ozay, and Richard M. Murray. 2016. Control design for hybrid systems with TuLiP: The Temporal Logic Planning toolbox. In *2016 IEEE Conference on Control Applications, CCA 2016*. 1030–1041.

[14] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy, SP 2018*. 3–18.

[15] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2008. Program Analysis As Constraint Solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. 281–292.

[16] Sumit Gulwani and Ashish Tiwari. 2008. Constraint-based approach for analysis of hybrid systems. In *International Conference on Computer Aided Verification*. 190–203.

[17] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *CAV (1) (Lecture Notes in Computer Science)*, Vol. 10426. 3–29.

[18] Dominic W Jordan and Peter Smith. 1987. Nonlinear ordinary differential equations. (1987).

[19] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Computer Aided Verification - 29th International Conference, CAV 2017*. 97–117.

[20] Hui Kong, Fei He, Xiaoyu Song, William N. N. Hung, and Ming Gu. 2013. Exponential-Condition-Based Barrier Certificate Generation for Safety Verification of Hybrid Systems. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'13)*. 242–257.

[21] Bettina Könighofer, Mohammed Alshiekh, Roderick Bloem, Laura Humphrey, Robert Könighofer, Ufuk Topcu, and Chao Wang. 2017. Shield Synthesis. *Form. Methods Syst. Des.* 51, 2 (Nov. 2017), 332–361.

[22] Benoît Legat, Chris Coey, Robin Deits, Joey Huchette, and Amelia Perry. 2017 (accessed Nov 16, 2018). *Sum-of-squares optimization in Julia*. https://www.juliaopt.org/meetings/mit2017/legat.pdf

[23] J Matyas. 1965. Random optimization. *Automation and Remote Control* 26, 2 (1965), 246–253.

[24] Matthew Mirman, Timon Gehr, and Martin T. Vechev. 2018. Differentiable Abstract Interpretation for Provably Robust Neural Networks. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*. 3575–3583.

[25] Teodor Mihai Moldovan and Pieter Abbeel. 2012. Safe Exploration in Markov Decision Processes. In *Proceedings of the 29th International Coference on International Conference on Machine Learning (ICML'12)*. 1451–1458.

[26] Yurii Nesterov and Vladimir Spokoiny. 2017. Random Gradient-Free Minimization of Convex Functions. *Found. Comput. Math.* 17, 2 (2017), 527–566.

[27] Stephen Prajna and Ali Jadbabaie. 2004. Safety Verification of Hybrid Systems Using Barrier Certificates. In *Hybrid Systems: Computation and Control, 7th International Workshop, HSCC 2004*. 477–492.

[28] Aravind Rajeswaran, Kendall Lowrey, Emanuel Todorov, and Sham M. Kakade. 2017. Towards Generalization and Simplicity in Continuous Control. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*. 6553–6564.

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

[30] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. 2018. Reachability Analysis of Deep Neural Networks with Provable Guarantees. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018*. 2651–2659.

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

[32] Bastian Schürmann and Matthias Althoff. 2017. Optimal control of sets of solutions to formally guarantee constraints of disturbed linear systems. In *2017 American Control Conference, ACC 2017*. 2522–2529.

[33] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. Deterministic Policy Gradient Algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*. I–387–I–395.

[34] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. Berkeley, CA, USA. Advisor(s) Bodik, Rastislav.

[35] Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems (APLAS '09)*. 4–13.

[36] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic Testing for Deep Neural Networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. 109–119.

[37] Richard Sutton and Andrew Barto. 1998. *Reinforcement Learning*. MIT Press.

[38] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-neural-network-driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. 303–314.

[39] Matteo Turchetta, Felix Berkenkamp, and Andreas Krause. 2016. Safe Exploration in Finite Markov Decision Processes with Gaussian Processes. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. 4312–4320.

[40] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. 2018. Programmatically Interpretable Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*. 5052–5061.

[41] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. 2018. Feature-Guided Black-Box Safety Testing of Deep Neural Networks. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018*. 408–426.