Deep Reinforcement Learning Verification: A Survey

MATTHEW LANDERS and AFSANEH DORYAB, University of Virginia

Deep reinforcement learning (DRL) has proven capable of superhuman performance on many complex tasks. To achieve this success, DRL algorithms train a decision-making agent to select the actions that maximize some long-term performance measure. In many consequential real-world domains, however, optimal performance is not enough to justify an algorithm's use—for example, sometimes a system's robustness, stability, or safety must be rigorously ensured. Thus, methods for verifying DRL systems have emerged. These algorithms can guarantee a system's properties over an infinite set of inputs, but the task is not trivial. DRL relies on deep neural networks (DNNs). DNNs are often referred to as "black boxes" because examining their respective structures does not elucidate their decision-making processes. Moreover, the sequential nature of the problems DRL is used to solve promotes significant scalability challenges. Finally, because DRL environments are often stochastic, verification methods must account for probabilistic behavior. To address these complications, a new subfield has emerged. In this survey, we establish the foundations of DRL and DRL verification, define a taxonomy for DRL verification methods, describe approaches for dealing with stochasticity, characterize considerations related to writing specifications, enumerate common testing tasks/environments, and detail opportunities for future research.

CCS Concepts: • **Computing methodologies → Reinforcement learning**; • **General and reference → Surveys and overviews;**

Additional Key Words and Phrases: Deep reinforcement learning, neural network verification, trustworthy reinforcement learning

ACM Reference format:
Matthew Landers and Afsaneh Doryab. 2023. Deep Reinforcement Learning Verification: A Survey. *ACM Comput. Surv.* 55, 14s, Article 330 (July 2023), 31 pages.
https://doi.org/10.1145/3596444

1 INTRODUCTION

Reinforcement Learning (RL) is a machine learning subfield in which methods are designed to solve sequential decision-making problems. Recently, the efficacy of these algorithms has been demonstrated in a variety of complex domains such as robotic control [88], autonomous driving [47, 50, 79], game playing [63], and dialogue generation [57]. As confidence in RL has amassed, these methods have been increasingly relied upon to make decisions in domains with high human costs. For example, RL has been used in healthcare to design dynamic treatment regimes, automate medical diagnosis, and manage disease [41, 108].

In RL, a decision-making agent selects actions based upon information provided by the environment in which it operates. One of these data is a numerical reward that provides a signal to

Authors' address: M. Landers and A. Doryab, Department of Computer Science, University of Virginia, 151 Engineer's Way, Charlottesville, VA 22903; emails: {mlanders, ad4ks}@virginia.edu.

0360-0300/2023/07-ART330 $15.00
https://doi.org/10.1145/3596444

the agent about its performance. These rewards are earned as a direct consequence of the agent's actions. The agent's goal is to learn how to act such that its cumulative reward is maximized; this task can be considered an optimal control problem. Unlike traditional control methods that require a well-specified model of the environment, RL algorithms learn through trial-and-error interaction with their environment [74]. This makes RL suitable for settings with unknown or time-varying dynamics, infinite state and action spaces, or changing performance requirements.

Although RL has proven remarkably capable of solving challenging decision-making problems, its fundamental objective to maximize long-term rewards has limited its use in real-world applications for which optimal performance is not the sole property of interest [70]. For example, robotic systems often need stability assurances [2, 14, 53], air traffic controllers must be robust [44], and many applications require safety guarantees [33]. In response, a class of algorithms from formal methods research—called *verification* methods—have been adapted for RL. Defined generally, verification techniques either guarantee a system's properties over an infinite set of inputs or find counterexamples that demonstrate violations of the properties.

A diverse set of RL verification techniques have been developed. In this survey, we provide an overview of a particularly exciting subset of RL verification methods—those designed to verify **Deep Reinforcement Learning (DRL)** algorithms. The article is structured as follows. In Section 2, we introduce the technical foundations of DRL. Next, in Section 3, we describe the challenges associated with verifying DRL systems. In Section 4, we define a taxonomy for DRL verification methods and describe representative methods therein. Section 5 examines methods for contending with the probabilistic nature of DRL systems. In Section 6, we characterize considerations related to writing specifications, enumerate common testing tasks/environments, and define the metrics used to evaluate DRL verification methods. Finally, in Section 7, we detail opportunities for future research.

In this article, we use lowercase italic font for scalars (x), lowercase bold font for vectors ($\mathbf{x}$), uppercase bold font for matrices ($\mathbf{X}$), and calligraphic uppercase font for sets ($\mathcal{X}$).

2 FOUNDATIONS OF DRL

RL is both a learning problem and a class of methods designed to solve that problem. The learning problem is formalized as an optimal control task in which an agent determines how to act in an environment such that the value of a numerical objective is maximized [86]. Traditional RL methods have been successful in many applications but are inherently limited to low-dimensional settings. DRL methods, by contrast, can scale to infinitely large input spaces by using **Deep Neural Networks (DNNs)** to find low-dimensional representations of high-dimensional data [8]. In this section, we introduce the RL problem setting and foundational RL/DRL techniques.

2.1 Reinforcement Learning

Broadly defined, sequential decision making proceeds as follows. Given some information, an agent makes a decision. The consequences of that decision are then observed and used to make the next decision. The agent's goal is to maximize some performance metric over a defined time horizon. In RL, this procedure is typically formalized as the optimal control of an incompletely known **Markov Decision Process (MDP)** [86]. An MDP can be represented as a collection of objects $\mathcal{M} = \langle \mathcal{T}, \mathcal{S}, \mathcal{A}, \mathbf{P}, \mathbf{R}, \gamma \rangle$, where $\mathcal{T}$ is a set of timesteps in which decisions are made, $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $\mathbf{P}$ is a transition probability matrix, $\mathbf{R}$ is a reward function, and γ is a discount factor [73].

At each timestep $t \in \mathcal{T}$, the agent has some information about the environment in which it operates. This information is referred to as the agent state $s_t \in \mathcal{S}$. Based on the information in s_t, the agent selects an action $a_t \in \mathcal{A}$. The environment receives action a_t, after which it emits a new state $s_{t+1} \in \mathcal{S}$ and a signal to the agent about its performance in the form of a reward $r_{t+1} \in \mathbb{R}$.

The probability that the environment emits a particular state $s' \in \mathcal{S}$ given action $a \in \mathcal{A}$ is performed in $s \in \mathcal{S}$ is governed by the state transition probability matrix $\mathbf{P}(s, a, s') = \mathbb{P}[s_{t+1} = s' \mid s_t = s, a_t = a]$. These transitions are typically modeled as a probability distribution because most systems are subject to some amount of stochasticity.

The agent's reward at each timestep is determined by the reward function $\mathbf{R}(s, a) = \mathbb{E}[r_{t+1} \mid s_t = s, a_t = a]$, which gives the expected immediate reward after $a \in \mathcal{A}$ is performed in $s \in \mathcal{S}$ [87].

An agent's performance—or its *return*—is determined by summing its discounted rewards: $g_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$, where $\gamma \in [0, 1]$.

2.1.1 *Value Functions and Policies.*

Given that an agent's objective is to maximize the sum of its discounted rewards and given rewards are received after transitioning from s to s', it is sensible for the agent to develop some notion about the quality of each state. The *value function* is used for this purpose; it ascribes a value to a state based on the returns the agent can expect from that state.

Because transitioning from s to s' requires an action a, the value function is defined with respect to a sequence of actions called a *policy*. A policy maps states to distributions over action spaces $\pi(a \mid s) = \mathbb{P}[a_t = a \mid s_t = s]$. In other words, the policy dictates the agent's actions.

The *state-value function* defines the value of a particular state with respect to a policy $v_\pi(s) = \mathbb{E}_\pi[g_t \mid s_t = s]$, whereas the *action-value function* defines the value of taking a particular action in a particular state with respect to a policy $q_\pi(s, a) = \mathbb{E}_\pi[g_t \mid s_t = s, a_t = a]$. Informally, the action-value function is the expected return when taking a in s and following π thereafter. These functions allow us to directly compare the quality of policies and are thus fundamental to finding the *optimal policy* π_*; the policy with expected returns greater than or equal to all other policies. Specifically, if an MDP is finite—that is, it has a finite number of states, actions, and rewards—then $\pi \geq \pi'$ iff $q_\pi(s, a) \geq q_{\pi'}(s, a) \;\; \forall s \in \mathcal{S}, a \in \mathcal{A}$ [86].

2.1.2 *Finding the Optimal Policy.*

The Bellman equation is a useful representation for finding the value functions; it expresses a state's value in terms of its immediate reward and the discounted value of its successor state $v_\pi(s) = \mathbb{E}_\pi[r_{t+1} + \gamma v_\pi(s_{t+1}) \mid s_t = s]$. The *action-value Bellman expectation equation* is defined as $q_\pi(s, a) = \mathbf{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathbf{P}(s, a, s') \sum_{a' \in \mathcal{A}} \pi(a' \mid s') q_\pi(s', a')$.

Solving the Bellman expectation equation amounts to finding the value of a policy π; however, we are typically not strictly interested in finding the value of a particular policy. Rather, we want to find π_*. We can find the optimal policy using the Bellman equation and greedy action selection.

A greedy action with respect to q_π in state s maximizes $q_\pi(s, a)$. Naturally, a greedy policy with respect to q_π selects the greedy action in all states. It therefore follows that the optimal policy is the greedy policy with respect to the optimal value function. We can find the optimal value function by solving the *optimal action-value function* $q_*(s, a) = \mathbf{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathbf{P}(s, a, s') v_*(s')$, where $v_*(s) = \max_a q_*(s, a)$.

Many methods have been designed to solve this equation (we refer interested readers to the work of Sutton and Barto [86] for a complete introduction to fundamental RL algorithms). Q-learning, for example, continually updates a state's estimated value using the estimated value of other states. Specifically, the value of a state-action pair $q(s_t, a_t)$ is updated immediately upon transitioning from s_t to s_{t+1}: $\hat{q}(s_t, a_t) \leftarrow \hat{q}(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_{a_{t+1}} \hat{q}(s_{t+1}, a_{t+1}) - \hat{q}(s_t, a_t))$, where $\alpha \in (0, 1]$ is a learning rate parameter.

2.2 Deep Reinforcement Learning

Traditional RL methods require storing the value of each state-action pair. Q-learning's update rule, for example, needs the current value $q(s, a)$ and the value of $\max_{a'} q(s', a')$. Thus, the value of $q(s, a)$ for all $s \in \mathcal{S}, a \in \mathcal{A}$ must be stored (e.g., in a table). Storing all of these values, however, is often not feasible—modeling chess takes $\sim 10^{120}$ states [80] ($\gg \sim 10^{82}$ atoms in the observable universe), computer Go needs $\sim 10^{170}$ states [96], and most robotic applications require a continuous state space. Thus, we need methods that can generalize from an agent's experience with a limited subset of the state and/or action space. DRL methods rely on a DNN to approximate the optimal value function. One approach, deep Q-learning, is fundamentally the same as Q-learning with the notable exception that it introduces a DNN. Specifically, a **Deep Q Network (DQN)** takes a state as an input and estimates the Q value for every action possible in that state $f : \mathbf{s}_t \rightarrow (q_{\theta}(\mathbf{s}_t, a_0), \ldots, q_{\theta}(\mathbf{s}_t, a_n))$. To learn, deep Q-learning uses gradient descent to minimize the squared difference between the current estimate of the Q value and the updated estimate derived using the Bellman equation: $[(r + \gamma \max_{a'} q_{\theta}(\mathbf{s}', a')) - q_{\theta}(\mathbf{s}, a)]^2$.

Note that the notation for a state has changed from s to $\mathbf{s}$ for two reasons. First, inputs to a DQN are typically vectors instead of scalars. Second, it is more consistent with the notation used in the subsequent DRL verification sections of this article.

3 DRL VERIFICATION CHALLENGES

The efficacy of DNNs has been explored in a diverse set of applications beyond DRL. As these networks have proliferated, there has been an increasing need to rigorously ensure their properties (e.g., safety). In response, approaches from formal methods research have been adapted for DNNs. One such approach—called *verification*—guarantees a system's properties over an infinite set of inputs. In principle, this makes verification appropriate for assuring the quality of DNNs as these networks learn a function that fits a training dataset while generalizing to (potentially infinite) unseen inputs. Despite this compatibility, the fundamental characteristics of DNNs make verification efforts difficult.

Traditionally, software verification methods have been used to prove *functional correctness*—a guarantee that a program is faithful to the mathematical function it implements [3, 29]. Although important for many applications, functional correctness is incongruent with a DNN's purpose. DNNs are used to approximate complex functions that would otherwise be impractical or impossible to define. Moreover, a DNN is a "black box," meaning that examining a DNN's internal structure does not elucidate its decision-making process. As a consequence, we cannot precisely characterize the mathematical function to which a DNN should adhere. We can, however, define a DNN's desirable properties.

The properties to be verified are use case dependent. For DRL, verification typically concerns proving safety (informally, nothing "bad" ever happens) and liveness (informally, something "good" eventually happens) constraints. As we will describe, however, DRL verification techniques are not inherently restricted to proving these two properties.

Many neural network verification algorithms have been proposed [44, 45, 68, 81, 92, 102, 104]. Because DRL methods rely on DNNs, these techniques form the foundation of DRL verification approaches. Thus, in this section, we will broadly describe the fundamentals of DNN verification before focusing on DRL verification. Like our discussion of DRL, this section is not exhaustive; however, it does give sufficient background for a thorough consideration of the DRL verification methods described in Section 4.

3.1 Defining Specifications

Consider an n-layer neural network f with input $\mathbf{x} \in \mathcal{D}_{\mathbf{x}} \subseteq \mathbb{R}^{k_0}$ and output $\mathbf{y} \in \mathcal{D}_{\mathbf{y}} \subseteq \mathbb{R}^{k_n}$, where k_0 is the input dimension and k_n is the output dimension. Verifying f requires properties ϕ that govern the desired inputâĂŞoutput behavior of f [59]. Informally, ϕ is constructed: "for any input $\mathbf{x} \in \mathcal{X}$ the neural network f produces outputs $\mathbf{y} \in \mathcal{Y}$" [3], where $\mathcal{X} \subseteq \mathcal{D}_{\mathbf{x}}$ is a constraint on the network's inputs and $\mathcal{Y} \subseteq \mathcal{D}_y$ is a constraint on the network's outputs. More succinctly, solving a verification problem requires demonstrating $\mathbf{x} \in \mathcal{X} \Rightarrow \mathbf{y} = f(\mathbf{x}) \in \mathcal{Y}$, where $\Rightarrow$ denotes implication.

For example, to verify that an image classification network is robust, we must ensure that small perturbations of the input (e.g., slightly altering the brightness of the pixels) do not affect the network's output. Assume we have some input image $\mathbf{i}$ that has been labeled $c^* \in \{1, \ldots, k_n\}$. Our goal is to establish that $y_{c^*} > y_j$ for all $j \neq c^*$. Thus, we define the input and output constraints as

$$\mathcal{X} = \{\mathbf{x} : \|\mathbf{x} - \mathbf{x}_0\|_p \leq \epsilon\}$$
$$\mathcal{Y} = \{\mathbf{y} : y_{c^*} > y_j, \forall j \neq c^*\},$$

where ϵ is the maximum amount the input can be perturbed (e.g., how much the brightness of each pixel can be adjusted) and p denotes the norm used to measure the perturbation size [59].

Given a network and a property, DNN verification algorithms generate one of three types of results. An *adversarial result* is the minimum perturbation ϵ^* required to produce an undesirable result; if $\epsilon^* < \epsilon$, the specification is violated. A *counter-example result* provides an example $\mathbf{x}^* \in \mathcal{X}$ that violates the specification $f(\mathbf{x}^*) \notin \mathcal{Y}$; if such a result is found, the specification is violated. A *reachability result* produces the network's output reachable set $\mathcal{R}(\mathcal{X}, f) := \{\mathbf{y} : \mathbf{y} = f(\mathbf{x}), \forall \mathbf{x} \in \mathcal{X}\}$; if the reachable set does not belong to the output set, $\mathcal{R}(\mathcal{X}, f) \not\subseteq \mathcal{Y}$, the specification is violated [59].

3.2 Constraint-Based Verification

Specifications, denoted as ϕ, are commonly encoded as constraints with first-order logic. With propositional logic, the simplest form of first-order logic, arguments can be formed with Boolean variables. Compound propositions can be constructed by connecting propositions with the logical connectives AND ($\wedge$), OR ($\vee$), and not ($\neg$).

When defining properties for neural networks, an extension of propositional logic called *arithmetic theories* is often used. A particular arithmetic theory, linear real arithmetic, allows us to represent a Boolean expression as a linear inequality:

$$\sum_{i=1}^{n} c_i x_i + b \leq 0 \qquad \text{or} \qquad \sum_{i=1}^{n} c_i x_i + b < 0,$$

where $c_i, b \in \mathbb{R}$ and x_i is a fixed set of variables [3].

Consider a simple network with two inputs x and y, one neuron $t = 2x + 8y$, and a ReLU activation function. To verify the network returns a value greater than or equal to 0 for all inputs between 0 and 1, we encode the network $\psi \triangleq t = 2x + 8y \wedge (t > 0 \Rightarrow r = t) \wedge (t \leq 0 \Rightarrow r = 0)$ and the correctness property $\forall x, y \in [0, 1]\ \phi \Rightarrow r \geq 0$. Because ψ and ϕ are just conjunctions and disjunctions of linear inequalities, constraint-based verification methods use a **Satisfiability Modulo Theories (SMT)** solver [66] or a **Mixed-Integer Linear Programming (MILP)** solver [92] to either prove or disprove ϕ [4].

Constraint-based verification's reliance on SMT and MILP solvers introduces a fundamental scalability challenge. Specifically, the nonlinearity introduced by a ReLU ($f(x) = 0$ **if** $x \leq 0$ **else** $f(x) = x$) must be partitioned into two linear constraints and checked separately by a solver. Consequently, the total number of linear pieces grows exponentially with the number of nodes in the network [65, 71], which makes complete verification NP-hard [44].

3.3 Abstraction-Based Verification

In contrast to constraint-based techniques, which are exact but inefficient, abstraction-based verification methods are efficient but not complete, meaning not everything that is true has a proof

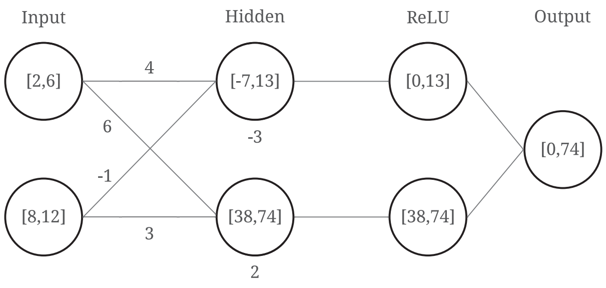

Fig. 1. A simple neural network with a ReLU activation layer through which abstract intervals are propagated. The numbers next to edges denote weights, whereas the numbers beneath the nodes denote biases.

(i.e., the verifier does not always give a definitive "yes" or "no" answer). Despite not being complete, abstraction-based verification is sound, meaning everything that is provable is genuinely true.

Abstraction-based verifiers achieve their efficiency by over-approximating the neural network's semantics. Specifically, these techniques employ abstract domains [25] to evaluate the neural network on sets of inputs. In other words, $f(\mathcal{X})$ gives the network's predictions for all inputs in the domain defined by $\mathcal{X}$. For example, proving $f(\mathcal{X}) \subseteq \{\mathbf{y} \mid \text{class}(\mathbf{y}) = \text{“dog”}\}$ verifies that a network classifies all inputs $\mathbf{x} \in \mathcal{X}$ as a "dog" [3].

The interval domain is the simplest abstract domain. Consider an elementary function for which we want to evaluate a set of inputs $g(\mathcal{X}) = \{x + 2 \mid x \in \mathcal{X}\}$. Because $\mathcal{X}$ is potentially infinite, it is infeasible to assess g in the concrete domain. Abstract interpretation simplifies this problem by evaluating an interval of real values $[l, u]$, where $l, u \in \mathbb{R}$, $l \leq u$, and $\{x \mid l \leq x \leq u\}$. To do so, we rewrite g as $g^a([l, u]) = [l + 2, u + 2]$. Like g, g^a takes a set of real numbers and returns a set of real numbers. However, unlike g, the input and output sets for g^a are intervals [3].

The principles illustrated by our simple function $g(x) = x + 2$ easily extend to the activation and affine functions that constitute neural networks. Specifically, for any monotonically increasing activation function, we define the function $f^a([l, u]) = [f(l), f(u)]$. For an affine function $f(x_1, \ldots, x_n) \sum_i c_i x_i$, where $c_i \in \mathbb{R}$, we define the function

$$f^a([l_1, u_1], \ldots, [l_n, u_n]) = \left[\sum_i l'_i, \sum_i u'_i \right],$$

where $l'_i = \min(c_i \cdot l_i, c_i \cdot u_i)$, $u'_i = \max(c_i \cdot l_i, c_i \cdot u_i)$, and $([l_1, u_1], \ldots, [l_n, u_n])$ denotes a hyperrectangular region in $\mathbb{R}^n$—that is, the set of all vectors of size n in the region defined by l_i and u_i $\{\mathbf{x} \in \mathbb{R}^n \mid l_i \leq x_i \leq u_i\}$ [3].

A neural network can then be defined as a composition of abstract activation and abstract affine functions. The properties of that network can be verified once the intervals have fully propagated. For example, a property might specify that for inputs $x_1 \in [2, 8]$ and $x_2 \in [6, 12]$, the output $y \in [0, 100]$. Given the network in Figure 1, a verifier could guarantee that this property holds.

Although the interval domain has been used to verify DNNs employed in complex applications such as natural language processing [37] and image classification [35], this domain is non-relational, meaning it does not account for dependencies between inputs. As a consequence, the interval domain can dramatically over-approximate the solution bounds. Relational domains such as the neural zonotope abstraction [34, 59, 81] and the neural polyhedron abstraction [82] can define tighter bounds and are thus often preferable.

3.4 Challenges Specific to DRL

DRL methods fundamentally rely on a DNN; however, verifying DRL systems requires solving a set of challenges distinct from those associated with typical DNN verification [98]. As described in Section 2, DRL algorithms are designed to solve sequential decision-making problems. Practically, this means a DNN must be invoked many times. In other words, the DNN must decide which action to take at each timestep $t \in \mathcal{T}$. Most other deep learning problems, by contrast, require a single DNN invocation. Thus, although scalability has become a principal challenge in DNN verification, the problem is aggravated for DRL verification [6, 31]. Consider robustness verification. As described in Section 3.1, for an image classifier, this task entails ensuring that small perturbations of the input (e.g., slightly altering the brightness of the pixels) do not affect the network's output. Importantly, this property needs to be proved for just one DNN invocation. DRL robustness certification (further described in Section 6.1), by contrast, requires guaranteeing the property holds for each $t \in \mathcal{T}$. For instance, verifying a self-driving car never changes its actions when its inputs are perturbed requires proving the DNN returns the same action for both the unperturbed and perturbed state *each time the car makes a decision*. In terms of complexity, verifying t consecutive invocations of a DNN with n nodes is effectively equivalent to encoding a single DNN with $(t \cdot n)$ nodes into the underlying verification engine. Because the verification problem is NP-complete [44], the worst-case complexity increases from 2^n to $2^{t \cdot n}$ [31].

As also described in Section 2, in DRL the output of one DNN invocation affects the inputs for subsequent invocations. Consequently, DRL verification requires reasoning about a series of results rather than a single output [6, 24, 31, 46]. For example, verifying a self-driving car will safely arrive at its destination requires proving that the cumulative effect of all the vehicle's decisions will be a safe arrival. Verifying properties that account for this sort of sequential behavior requires a different set of tools than those used to verify an isolated DNN output.

Finally, as detailed in Section 2, whether due to the policy or environment, DRL agents typically have to contend with stochasticity. This means verification methods often must account for probabilistic outcomes [6, 10, 27, 110]. Continuing with our self-driving car example, a learned policy might specify the vehicle should proceed through an intersection when a traffic light is green. However, an unanticipated event—such as the car's sensor failing to recognize the traffic light's true color—might occur. To guarantee the car safely navigates to its destination, verification methods must account for the probabilistic nature of events.

To contend with these unique challenges, a class of techniques specifically designed to verify DRL has emerged.

4 DRL VERIFICATION METHODS

In this section, we develop a taxonomy of DRL verification methods. The categories within this taxonomy include reduction to a simpler form, construction of a verified barrier function, reachability analysis, and model checking. Within each group, we describe many of the most important state-of-the-art methods. Note that the taxonomy's divisions are not mutually exclusive; approaches can fit within multiple categories.

4.1 Reduction to a Simpler-to-Verify Form

To circumvent DRL's verification challenges, several approaches reduce DRL's DNN to a simpler-to-verify form with neurosymbolic program synthesis [7, 13, 100, 101, 110].

In the classical setting, program synthesis attempts to find a program that provably satisfies a high-level correctness specification. In other words, the task is to find a function f such that a logical formula ϕ specifying f's correctness is valid. Neurosymbolic programs comprise both a neural and symbolic component. Like classical program synthesis, neurosymbolic programming's synthesis objective is defined by ϕ; however, it also includes a standard ML objective—learning a function that fits a training dataset while generalizing to unseen inputs [20].

In DRL verification, neurosymbolic program synthesis is typically used to create a highly structured—and therefore simple-to-verify—policy. For example, in the work of Bastani et al. [13], policies are constructed as decision trees. To form the trees, the approach relies on imitation learning. In imitation learning [1, 77], a classifier or regressor is trained to predict an expert's behavior given a training dataset containing the expert's observations and corresponding actions. For example, an autonomous vehicle system's DNN might learn from a dataset of driving scenes and a human expert's associated steering angles and speeds. Typically, the supervised learner assumes i.i.d. data. As described previously, however, RL violates the i.i.d. expectation. Consequently, if even a single mistake is made by the learner (due to the stochasticity of the environment or small but compounding differences between the learned policy and the expert policy), imitation-based DRL systems may encounter observations that are quite different from those experienced under expert demonstration; this makes learning very difficult [75].

Thus, Bastani et al. [13] use DAGGER [75], an imitation learning approach that is more robust to the changes in the distribution of observations induced by DRL. In its simplest form, DAGGER proceeds as follows. A policy $\hat{\pi}_0$ is initialized by cloning the expert's demonstration. Using $\hat{\pi}_0$, the agent collects a set of trajectories. During the agent's execution of $\hat{\pi}_0$, DAGGER asks the expert which actions it would take in each state. By compiling a set of trajectories generated by $\hat{\pi}_0$ and the expert's corresponding actions, DAGGER expands the training dataset $\mathcal{D}$. Using $\mathcal{D}$, the supervised learner learns a new policy $\hat{\pi}_1$ that best mimics the expert on the set of trajectories in $\mathcal{D}$. This process repeats for a generic iteration n, during which $\hat{\pi}_n$ is used to collect more trajectories and labeled actions. These data are then added to $\mathcal{D}$. The next policy, $\hat{\pi}_{n+1}$, is the policy that best mimics the expert on the aggregate of all collected datasets in $\mathcal{D}$.

DAGGER, however, learns decision trees that are much larger than necessary and thus makes the verification task avoidably difficult. To learn smaller, more easily verified trees, two novel methods are introduced in the work of Bastani et al. [13]: Q-DAGGER and VIPER (Verifiability via Iterative Policy Extraction). Q-DAGGER runs the original DAGGER algorithm with a modified loss function. Specifically, DAGGER attempts to find a policy that minimizes the observed surrogate loss (e.g., 0–1 loss) under its induced distribution of states, whereas Q-DAGGER uses the Q-function to measure loss. Intuitively, Q-DAGGER uses the Q-function to emphasize training states for which the difference between the value of the best action and the value of the worst action is largest.

The neural policy $\hat{\pi}_n$ in each iteration of Q-DAGGER is constructed with VIPER, which uses the CART algorithm [17] to learn a decision tree. The standard CART algorithm cannot be used because DAGGER (and therefore Q-DAGGER) requires a strongly convex surrogate loss function [75] while the loss function for decision trees is not convex. To address this incongruity, VIPER does not modify the typical CART loss function. Instead, the algorithm creates a new training dataset $\mathcal{D}'$ in which examples from $\mathcal{D}$ are reweighed according to the difference between the value of a state's best action and worst action. Then, with $\mathcal{D}'$, the standard CART algorithm can be used to create a highly structured decision tree policy that is relatively small. Finally, several of the decision tree's properties—including correctness, robustness, and stability—can be verified with an SMT solver (discussed further in Section 6).

Like VIPER, PIRL (Programatically Interpretable Reinforcement Learning) [101] generates a highly structured policy. Instead of a decision tree, however, PIRL represents the learned policy in a human-readable **Domain-Specific Language (DSL)**. For example, a programmatic policy learned for controlling the acceleration of an autonomous car is presented in Figure 2.

The number of policies expressible in PIRL's DSL is vast and nonsmooth, thus NDPS (Neurally Directed Program Synthesis) is introduced to search for the policy that maximizes long-term

```
if $(0.001 - \textbf{peek}(h_{\texttt{TrackPos}}, -1) > 0)$ and $(0.001 + \textbf{peek}(h_{\texttt{TrackPos}}, -1) > 0)$
  then $3.97 * \textbf{peek}((0.44 - h_{\texttt{RPM}}), -1) + 0.01 * \textbf{fold}(+, (0.44 - h_{\texttt{RPM}})) + 48.79 * (\textbf{peek}(h_{\texttt{RPM}}, -2) - \textbf{peek}(h_{\texttt{RPM}}, -1))$
  else $3.97 * \textbf{peek}((0.40 - h_{\texttt{RPM}}), -1) + 0.01 * \textbf{fold}(+, (0.40 - h_{\texttt{RPM}})) + 48.79 * (\textbf{peek}(h_{\texttt{RPM}}, -2) - \textbf{peek}(h_{\texttt{RPM}}, -1))$
```

Fig. 2. A programmatic policy learned for controlling the acceleration of an autonomous car where peek and fold are constructs of PIRL's DSL [101].

rewards. The approach proceeds by using DRL to compute an "oracle" neural network policy $e_{\mathcal{N}}$ that has high performance but may not be expressible in the DSL. This neural policy $e_{\mathcal{N}}$ is then used to direct a local search over programmatic policies—that is, policies that can be expressed with the DSL—to find a program e^* that closely imitates $e_{\mathcal{N}}$. During each iteration, NDPS searches the neighborhood around the current estimate of e^*, denoted e, to find a program e' that minimizes the distance between e and $e_{\mathcal{N}}$ for a set of "interesting" inputs (where "interesting" is defined by some heuristic). At the end of the iteration, e' becomes the new estimate for e^*. This procedure can be considered a form of imitation learning, which, like VIPER, is based upon DAGGER. The PIRL framework generates interpretable program source code that can be verified with traditional symbolic program verification techniques [49].

Although the human-interpretable form into which PIRL distills a neural policy is easily verified, imitating a pre-trained neural policy into a short program can produce highly suboptimal programmatic policies. PROPEL (Imitation-Projected Programmatic Reinforcement Learning) [100] was developed to address this limitation by iteratively distilling a neural policy into a programmatic policy inside a training loop. Specifically, PROPEL constructs the discovery of an optimal symbolic policy as a constrained optimization problem in which the desired policies are constrained to be those with a programmatic representation. The task is to find a policy that minimizes the expected cost (or alternatively maximizes the expected reward) of a programmatic policy $\pi \in \Pi$, where $\Pi \subset \mathcal{H}$ is the class of programmatic policies that resides within a larger space of policies $\mathcal{H} \equiv \Pi \oplus \mathcal{F}$, and $\mathcal{F}$ is the set of neural policies. Here, $\oplus$ denotes a mixing [16] of programmatic policies Π and neural policies $\mathcal{F}$ where any mixed policy $h \equiv \pi + f$ can be invoked as $h(s) = \pi(s) + f(s)$. The learning algorithm proceeds as follows. Given an initial neural policy $f_0 \in \mathcal{F}$, PROPEL projects f_0 onto the constrained space Π with program synthesis via imitation learning—that is, by synthesizing a policy $\pi_0 \in \Pi$ to best imitate f_0 (e.g., with DAGGER). Then, during each iteration, the algorithm makes a series of gradient updates to the neurosymbolic policy $h(x) \equiv f_i(x) + \pi_i(s)$ using classic DRL techniques such as deep deterministic policy gradient (DDPG) [58]. This gradient update is not the true gradient in $\mathcal{H}$. Instead, the updates are based on the gradient of the neural policy f_i because there are well-established neural gradient methods, whereas there are no existing gradient methods for programmatic policy classes Π. After the gradient update, PROPEL projects h onto the constrained space Π with program synthesis via imitation learning. In other words, a policy $\pi_{i+1} \in \Pi$ is synthesized to best imitate demonstrations provided by the teaching oracle h. Then π_{i+1} is lifted back into the neurosymbolic space $\mathcal{H}$ so that the gradient updates can be applied. This process repeats for t iterations, after which a symbolic policy is returned. This policy can then be verified with an SMT solver.

4.2 Construction of a Verified Barrier Function

Because the policies constructed by VIPER, PIRL, and PROPEL are verified only after learning, it is difficult to repair any specification-violating policy. Approaches that constitute a separate class of neurosymbolic synthesis methods address this limitation by synthesizing a verifiable barrier function that prevents an agent from taking an action that causes a violation of the specification [7, 107, 110].

For example, in the work of Zhu et al. [110], syntax-guided synthesis is used to find a deterministic program P^* that both approximates a neural policy π_{w} and satisfies a safety specification.

Because P^* is deterministic, off-the-shelf formal verification algorithms can be used to guarantee its safety. However, verifying P^*'s safety is not akin to verifying π_w's safety. Thus, whereas π_w is used to make decisions during runtime, P^* is used to shield the agent from unsafe states. Because P^* must dictate actions that protect the agent from unsafe states while following π_w, the two policies must be reasonably similar. Thus, in addition to satisfying a safety specification, P^* must also satisfy a quantitative specification that ensures P^* is relatively comparable to the neural policy π_w.

The approach relies on a counterexample guided inductive synthesis loop. Specifically, the process initially sets the unknown parameters (i.e., those to be filled in by the synthesis procedure) to zero, $\boldsymbol{\theta} = \mathbf{0}$. It then runs the synthesized deterministic program P_θ with a set of initial states $\mathcal{S}_0$ for which the synthesized program is not yet proved safe. These states are referred to as counterexamples.

The safety property of the synthesized program P in the environment C can be expressed as an inductive invariant ϕ that represents all safe states over the state transition system $C[P]$. Specifically, ϕ is disjoint with all unsafe states, ϕ includes all initial states, and any state in ϕ transitions to another state in ϕ, thus an unsafe state cannot be reached. Because P is deterministic, the safety of $C[P]$ can be guaranteed with off-the-shelf verification algorithms. Specifically, a constraint solver is used to look for an inductive invariant in the form of a convex barrier certificate [72] that maps all states in the safe reachable set to non-positive reals and all states in the unsafe set to positive reals. Fundamentally, the solver looks for a polynomial function $E : \mathbb{R}^n \rightarrow \mathbb{R}$ such that $E(s) > 0$ for any unsafe state, $E(s) \leq 0$ for any initial state, and $E(s') - E(s) \leq 0$ for any state s that transitions to s' in $C[P]$. Together, the second and third conditions guarantee that $E(s) \leq 0$ for any state s in the reachable set, which means an unsafe state can never be reached. Note that the barrier E must be a polynomial function because the method's policy programming language only allows the invariant sketch [83] (i.e., the human-defined partial program that encodes the structure of a solution) to be postulated as a polynomial function. This restriction is imposed because there are efficient SMT solvers and constraint solvers for nonlinear polynomial reals.

When verification for P_θ fails, the initial state space $\mathcal{S}_0$ is reduced in an effort to make the synthesis of a safe policy program easier. With this now smaller set of counterexample states, a new $\boldsymbol{\theta}$ is synthesized. The new value of $\boldsymbol{\theta}$ is determined by a random search-based optimization algorithm [61] that iteratively samples new positions of $\boldsymbol{\theta}$ from a small hypersphere surrounding the current position. The value of $\boldsymbol{\theta}$ is set to the position that minimizes the distance between P_θ and π_w while simultaneously maximizing the likelihood that P_θ is safe.

When verification succeeds, a new policy program P_{θ_n}—which is guaranteed safe in the state space covered by ϕ_n—is synthesized.

This synthesize-and-verify procedure continues until the entire initial state space $\mathcal{S}_0$ is verified safe. The approach outputs a set of synthesized program-state space pairs $[(P_{\theta_1}, \phi_1), (P_{\theta_2}, \phi_2), \ldots]$. For each pair, the program is guaranteed safe over the corresponding state space.

Unlike the post-learning synthesis methods, this counterexample guided inductive synthesis approach avoids any possibility of constructing a policy that may violate the specification; however, its synthesized policies are verified safe only after training. A separate method, Revel (Reinforcement Learning with Verified Exploration) [7], builds on Propel to guide the synthesis of a barrier function that guarantees safety at all times during training by updating both the symbolic and neural components of a neurosymbolic policy.

To maintain safety during and after training, Revel discovers a learning process $\mathcal{L} = \pi_0, \pi_1, \ldots \pi_m$ such that the final policy π_m is safe and optimal and every policy in $\mathcal{L}$ is safe. Given a policy π, a verifier attempts to construct an inductive proof of a safety property. This proof takes the form of an inductive invariant defined as a set of states ϕ such that the system can never transition to an unsafe state from any state in ϕ even under worst-case dynamics.

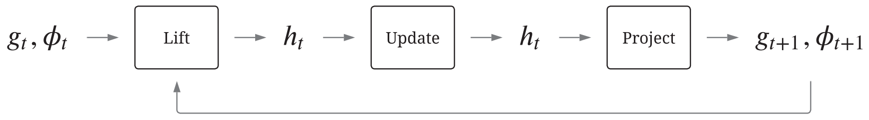

Fig. 3. Revel's iterative procedure. First, the symbolic policy g_t and proof ϕ_t are lifted into the blended space to create the neurosymbolic policy h_t. Next, h_t is updated in the neural policy space. Finally, a safe symbolic policy g_{t+1} is synthesized and its corresponding invariant ϕ_{t+1} is defined. This lift-update-project procedure continues for T iterations, after which a neurosymbolic policy h_T is returned.

To learn $\mathcal{L}$, Revel constructs a neurosymbolic policy $h \in \mathcal{H}$ in the form

$$h(s) = \textbf{ if } P^{\#}(s, f(s)) \subseteq \phi) \textbf{ then } f(s) \textbf{ else } g(s),$$

where $P^{\#}$ is a deterministic function that defines worst-case bounds on the environment behavior, $f \in \mathcal{F}$ is a neural policy, and $g \in \mathcal{G} \subseteq \mathcal{H}$ is a verifiably safe shielding policy that is also expressible as a policy in $\mathcal{F}$ (i.e., $\mathcal{G} \subseteq \mathcal{F}$). The policies in class $\mathcal{G}$ can be efficiently verified by existing algorithms because they are syntactically restricted to be compact and symbolic. The condition $P^{\#}(s, f(s)) \subseteq \phi$ is effectively a safety monitor; if it evaluates to true, then the action $f(s)$ is safe, as it can only lead to states in ϕ. If this condition is false, $f(s)$ can lead to an unsafe state and the shield $g(s)$ is executed instead of f.

Revel follows a lift, update, project learning procedure. The approach starts with a manually constructed shield $g_0 \in \mathcal{G}$ and a corresponding invariant ϕ_0. The algorithm then iteratively performs three steps. First, it lifts a policy $g_0 \in \mathcal{G}$ into the unconstrained policy space $\mathcal{H}$ to construct a policy h_0 for which the neural component is just the neural representation of g_0. The neural representation of g_0 can be constructed with imitation learning (e.g., with Dagger). Next, the algorithm updates h_0 using approximate gradients where the approximation is computed in a similar manner to Propel. Finally, the updated policy is projected back into the constrained space $\mathcal{G}$ to compute a new invariant ϕ_1 and a policy g_1 that is a minimum imitation distance from h_0 (Figure 3). The inductive invariant ϕ_1 is constructed with abstract interpretation. Specifically, by propagating an abstraction of the initial states through the environment transitions, Revel defines an abstract state that over-approximates the system's reachable states when following policy g_1. If this final abstract state does not include any unsafe states, no unsafe states are reachable by any concrete trajectory of the system.

4.3 Reachability Analysis

Verification algorithms perform one of three analyses: optimization, reachability, or search. Optimization methods solve an optimization problem (e.g., with linear programming or MILP) to falsify the specification. Reachability approaches find the system's reachable set $\mathcal{R}(\mathcal{X}, f) := \{\mathbf{y} : \mathbf{y} = f(\mathbf{x}), \forall \mathbf{x} \in \mathcal{X}\}$; if $\mathcal{R}(\mathcal{X}, f) \nsubseteq \mathcal{Y}$, the specification is violated. Finally, search methods attempt to find an input that falsifies the specification often by using optimization or reachability approaches to guide the search procedure [59].

Because DRL specifications are typically defined in terms of safety (informally, nothing "bad" ever happens) and liveness (informally, something "good" eventually happens), determining reachability is a natural approach for verifying these systems [9, 26, 36, 39, 84, 85, 93]. Specifically, reachability methods can be used to determine whether an unsafe state (for a safety specification) or a desirable state (for a liveness specification) is reachable from any initial state [56].

One such approach, Verisig [39], was designed to verify closed-loop systems. In control theory, a closed-loop system is a system in which an agent's actions are dependent upon feedback from the process. In an open-loop system, by contrast, the control actions are selected independent of the process outputs. For example, a vehicle's cruise control system can be designed as either a closed or an open system. A closed-loop cruise control system maintains the vehicles speed by accounting for hills, wind gusts, and any other external influences. In the open implementation, the accelerator's position is fixed regardless of the terrain or environmental conditions. Because DRL uses feedback from its environment to guide decision making, it is well suited for finding the optimal policy in a closed-loop system.

Verisig divides the verification task into two distinct yet related problems—in the first, it determines the DNN's reachable set, and in the second, it verifies properties of the closed-loop system. To solve the first problem, Verisig transforms the DNN controller f into an equivalent hybrid system H_f such that if $\mathbf{x}_0$ is the initial state, then $f(\mathbf{x}_0)$ is the system's only reachable state. To solve the second problem, Verisig verifies a composed hybrid system $H_f \parallel H_S$, where $\parallel$ denotes a composed system and H_S is the hybrid system in which the controller is operating.

A hybrid system is a system that evolves based on both discrete and continuous variables. Consider, for example, an air-conditioning system. For such a system, the discrete variable—or *mode*—is the (Boolean) on/off state of the device, whereas the (real-valued) continuous variable is the room's temperature. To function effectively, the air conditioner needs to turn on once the air temperature exceeds some threshold and turn off once the temperature drops below some value. In this way, the system relies on both a discrete and continuous value and is thus a hybrid system [97].

Given H_f and H_S, there are known methods for composing $H_f \parallel H_S$, and there are several tools that can determine the reachability of hybrid systems like $H_f \parallel H_S$ [21]. Thus, because Verisig assumes the system's dynamics are given as a hybrid system H_S, the primary challenge becomes transforming the DNN into H_f.

Verisig is designed to verify DNNs with sigmoid or tanh activation functions (we will describe the method using the sigmoid function), as the sigmoid derivative can be expressed in terms of the sigmoid itself:

$$\sigma(x)\frac{d}{dx} = \sigma(x)(1-\sigma(x)).$$

This property allows the sigmoid to be treated like a quadratic dynamical system.

To establish the possible values of the sigmoid for a given set of inputs, Verisig inserts a "time" variable t into the standard sigmoid function,

$$g(t,\mathbf{x}) = \sigma(t\mathbf{x}) = \frac{1}{1+e^{-\mathbf{x}t}}, \tag{1}$$

such that $g(1,\mathbf{x}) = \sigma(\mathbf{x})$, and by the chain rule,

$$g(t,\mathbf{x})\frac{\partial}{\partial t} = \dot{g}(t,\mathbf{x}) = \mathbf{x}g(t,\mathbf{x})(1-g(t,\mathbf{x})). \tag{2}$$

By tracing the change in g with respect to time until $t = 1$ with (2), Verisig can determine the exact value of a sigmoid $\sigma(x)$. Moreover, because each neuron in a sigmoid-based DNN is a sigmoid function, (1) can be used to transform the entire DNN into a hybrid system.

Recall that a hybrid system is composed of continuous states and discrete modes. To create a hybrid system from a DNN, Verisig considers each hidden neuron a continuous state and each layer a mode. Specifically, let n_i be the number of neurons in hidden layer $\mathbf{f}_i$, and let f_{ij} denote neuron j in $\mathbf{f}_i$. The value of a single neuron can be calculated:

$$f_{ij}(\mathbf{x}) = \sigma((\mathbf{w}_i^j)^\top \mathbf{x} + b_i^j),$$

where $(\mathbf{w}_i^j)^\top$ is the jth row of ith hidden layer's weight matrix $\mathbf{W}_i$ and b_i^j is the jth element of the ith hidden layer's bias vector $\mathbf{b}_i$.

The value of a specific neuron at a specific time can then be defined with (1):

$$g_{ij}(t,\mathbf{x}) = \sigma(t \cdot ((\mathbf{w}_i^j)^\top \mathbf{x} + b_i^j)) = \frac{1}{1+\exp\{-t \cdot ((\mathbf{w}_i^j)^\top \mathbf{x} + b_i^j)\}}. \tag{3}$$

Using (2) and (3), the value of a hidden layer $\mathbf{f}_i(x)$ can be determined by tracing all $g_{ij}(t,\mathbf{x})$ from $t = 0$ to $t = 1$:

$$g_{ij}(t,\mathbf{x})\frac{\partial}{\partial t} = \dot{g}_{ij}(t,\mathbf{x}) = ((\mathbf{w}_i^j)^\top \mathbf{x} + b_i^j) g_{ij}(t,\mathbf{x})(1 - g_{ij}(t,\mathbf{x})). \tag{4}$$

In this way, each hidden layer can be represented as a set of differential equations $\dot{g}_{ij}(t,\mathbf{x})$ wherein each hidden neuron g_{ij} is a continuous state. The system's modes are then defined as the DNN's layers. In each mode, Verisig traces $g_{ij}(t,\mathbf{x})$ until $t = 1$ using (4) to ensure the hybrid system is equivalent to the DNN.

Finally, in addition to each mode's n_i continuous variables representing the neurons in $\mathbf{f}_i$, Verisig includes n_i supplementary continuous states (one per neuron) in each mode. These states are required because the inputs to each neuron are functions of states reached in the previous mode.

Having constructed a hybrid system that faithfully represents the DNN controller, Verisig relies on established hybrid system verification tools to determine reachability [21, 51, 66].

Verisig requires a model of the system dynamics H_S; however, another approach [36] uses model-based DRL to approximate the underlying system. The learned model is then verified using reachable tube analysis.

Reachable tube analysis is an exhaustive verification technique that tracks the evolution of a system's states over a finite time period from a set of initial states $\mathcal{S}_0 \subset \mathcal{S}$. A forward reachable tube tracks the system's development starting from $\mathcal{S}_0$, whereas a backward reachable tube evaluates the system as it progresses from a terminal state to a state in $\mathcal{S}_0$. Using a forward reachable tube, the approach in the work of Gupta and Hwang [36] determines if any trajectory $\xi_i \in \boldsymbol{\xi}$ executed under policy π leads to any unsafe state $\mathbf{s} \in C$ when starting in any initial state $\mathbf{s} \in \mathcal{S}_0$. Using a backward reachable tube, the approach finds the safe initial states $\mathcal{S}_{safe} \subset \mathcal{S}_0$ such that any trajectory $\xi_i \in \boldsymbol{\xi}$ executed under policy π does not lead to any unsafe state $\mathbf{s} \in C$ when starting in any initial state $\mathbf{s} \in \mathcal{S}_{safe}$.

This reachable tube analysis requires two steps. First, model-based DRL is used to construct a DNN $\hat{f}$ that models the true system dynamics. Because model-based DRL only approximates the underlying system, $\hat{f}$ is augmented with a conservative error bound. Second, the forward and backward reachable tubes are formulated over the system dynamics modeled by $\hat{f}$.

To learn $\hat{f}$, a labeled dataset $\mathcal{D} = \{(\mathbf{s}_t, a_t, \Delta\mathbf{s}_{t+1})^{(i)}\}\ \forall 1 \leq i \leq |\mathcal{D}|$, where $\Delta\mathbf{s}_{t+1}$ is the change in the state's value at time $t+1$, is split into a training dataset $\mathcal{D}_t$ and a validation dataset $\mathcal{D}_v$. Then a supervised learning algorithm is used to train $\hat{f}$ with D_t by minimizing the squared loss: $\frac{1}{n_t}\sum_{i=1}^{n_t}||\Delta\hat{\mathbf{s}}_{t+1}^{(i)} - \Delta\mathbf{s}_{t+1}^{(i)}||^2$, where $\Delta\hat{\mathbf{s}}$ is the predicted change in the state's value, $\Delta\mathbf{s}$ is the true change in the state's value, and $n_t = |\mathcal{D}_t|$.

Because $\hat{f}$ only approximates the true system dynamics, the model is augmented with an error term $\mathbf{e}$ [64]. Specifically, the approach uses error estimates from the validation set $\hat{e}_j = \Delta\hat{\mathbf{s}}_{t+1}^{(j)} - \Delta\mathbf{s}_{t+1}^{(j)}$ to define an upper confidence bound $\mathbf{e}^+ = [e_1, e_2, \ldots, e_n]$ and a lower confidence bound $\mathbf{e}^- = [-e_1, -e_2, \ldots, -e_n]$ for each of the state's dimensions. Then a high-confidence bounded error set is constructed $\mathcal{E} = \{\mathbf{e} : \forall i \in \{1,\ldots,n\}, e_i^- < e_i < e_i^+\}$. Finally, the learned system dynamics can be represented: $\hat{f} \coloneqq \hat{f}(\mathbf{s},a) + \mathbf{e},\ \mathbf{e} \in \mathcal{E}$.

With the learned model $\hat{f}$, the forward and backward reachable tubes can be formulated. The forward reachable tube—the set of all states that can be reached from an initial set $\mathcal{S}_0$ when trajectories $\boldsymbol{\xi}$ are executed under policy π given system dynamics $\hat{f}(\mathbf{s},a,\mathbf{e})$—is formally defined over a finite length of time T as follows:

$$\mathcal{F}(T) := \{\mathbf{s} : \forall \mathbf{e} \in \mathcal{E}, s(\cdot) \text{ satisfies } \dot{s} = \hat{f}(\mathbf{s}, a, \mathbf{e}) \text{ where } a = \pi(\mathbf{s}), \mathbf{s}_{t_0} \in \mathcal{S}_0, t_f = T\},$$

where t_0 and t_f are the initial and final time of the trajectory ξ_i, respectively.

The backward reachable tube is the set of all states that can reach a bounded target set $\mathcal{T} \subset \mathbb{R}^n$ when the trajectories ξ are executed under policy π given system dynamics $\hat{f}(\mathbf{s}, a, \mathbf{e})$. The backward reachable tube is formally defined over a finite length of time T as follows:

$$\mathcal{B}(-T) := \{\mathbf{s}_0 : \forall \mathbf{e} \in \mathcal{E}, \mathbf{s}(\cdot) \text{ satisfies } \dot{\mathbf{s}} = \hat{f}(\mathbf{s}, a, \mathbf{e}) \text{ where } a = \pi(\mathbf{s}), \mathbf{s}_{t_0} = \mathbf{s}_{-T}, \mathbf{s}_{t_f} \in \mathcal{T}, t_f \in [-T, 0]\}.$$

Having formalized $\hat{f}$ and the reachable tubes, the verification task is relatively simple. First, given a model-based policy π, the set of initial states $\mathcal{S}_0$, and the set of unsafe states C, the modeling error $\mathcal{E}$ is calculated. Next, the forward reachable tube is created. If the forward reachable tube does not contain any state in C, the policy π is safe; otherwise, π is unsafe. If π is unsafe, a backward reachable tube is used to determine the subset $\mathcal{S}_{safe} \subset \mathcal{S}_0$ for which π generates safe trajectories.

Reachable tube analysis is designed to be an exhaustive verification technique. However, when state and/or action spaces are continuous, it is neither feasible to sample trajectories for every state in $\mathcal{S}_0$ nor to verify whether all these trajectories satisfy the specified properties. Thus, reachable sets are often approximated by convex polyhedrons, and the system's evolution is evaluated by pushing the boundaries of the polyhedron according to the system dynamics. This approach, however, can lead to large errors in the approximation of the reachable set. To mitigate the magnitude of these errors, the method in the work of Gupta and Hwang [36] frames the construction of the reachable tubes as an optimal control problem using the Hamilton-Jacobi partial differential equation. This equation is solved using the level set method [62]—an approach that allows non-convex boundaries to be represented and thereby allows for a better approximation of the true reachable tube.

The level set method offers a better approximation of the reachable set than convex polyhedrons for reachable tube analysis; however, other DRL verification methods use alternative approaches for approximating the reachable set. For example, in the work of Tran et al. [93], the convex hull of star sets [94] is used to approximate the reachable set of a discrete linear control system with a ReLU-based neural network controller.

Specifically, the approach solves two verification problems. First, given a discrete linear system with a neural network controller f and an initial state set $\mathcal{S}_0$, the method verifies whether the system satisfies a safety property within k_{max} timesteps. More formally, the approach verifies if $\forall \mathbf{s}_0 \in \mathcal{S}_0 \rightarrow g(\mathbf{s}_k) \models P(g(\mathbf{s}_k)), \forall 0 \leq k \leq k_{max}$, where g is is a nonlinear transformation function (e.g., ReLU), P is a linear predicate over the transformed state variables $g(\mathbf{s}_k)$ that defines the system's safety specification (i.e., the state is safe if the predicate is true, and otherwise the state is unsafe), and $\models$ means $g(\mathbf{s}_k)$ satisfies $P(g(\mathbf{s}_k))$.

Second, the method can determine the initial condition of the ith state $\mathbf{s}^i \in \mathcal{S}_0$ for which the system may become unsafe while keeping the initial conditions of all other initial states $\mathbf{s}^j \in \mathcal{S}_0\ \forall j \neq i$ unchanged. This is useful when determining the maximum value of $\mathbf{s}^i$ for which the system's safety is still guaranteed—for example, a vehicle's maximum velocity such that applying the brake is guaranteed to stop the vehicle before it hits an object n feet away.

The approach can compute the controller's exact reachable set and an over-approximation of its reachable set. We begin by describing the two-step iterative procedure for finding the exact reachable set. First, the controller f takes the system's output set $\mathcal{Y}_0$ to compute the control set $\mathcal{U}_0 = f(\mathcal{Y}_0)$. Note that $\mathcal{Y}_0$ is just an affine mapping of the initial set $\mathcal{S}_0$ with the output matrix C (i.e., $\mathcal{Y}_0 = \mathrm{C}\mathcal{X}_0$). Then the approach determines the next state $\mathcal{S}_1$ by applying the control set $\mathcal{U}_0$

to the system $\mathcal{S}_1 = \mathbf{A}\mathcal{S}_0 + \mathbf{B}\mathcal{U}_0$. This loop is repeated k_{max} times, after which the exact reachable set is returned $\mathcal{S}_0, \mathcal{S}_1 \ldots, \mathcal{S}_k \; \forall 0 \leq k \leq k_{max}$.

The exact control set $\mathcal{U}_k = f(\mathbf{C}\mathcal{S}_k)$ is determined with a two-step layer-by-layer analysis using star sets. First, an affine map of the input set is calculated with the weight matrix $\mathbf{W}$ and bias vector $\mathbf{b}$. Second, using step-ReLU operations [95], the layer's reachable set is obtained by applying the ReLU activation function on the affine-mapped set. Notably, determining the neural network controller's exact reachable set can be done without star sets—for example, with a polyhedron-based approach. However, the polyhedron-based method is inefficient and generates an overly conservative reachable set, as it cannot leverage the relationship between $\mathcal{U}_k$ and $\mathcal{S}_k$ (i.e., $\mathcal{U}_k = f(\mathcal{Y}_k) = f(\mathbf{C}\mathcal{S}_k)$).

Although star sets allow for the efficient discovery of the neural network controller's exact reachable set, they are still prone to an explosion of the state sets. This makes an exact analysis impractical or infeasible. Thus, the approach can also over-approximate the reachable sets with an interval hull of the star sets—that is, the tightest outer interval solution that contains all the state sets at each iteration.

To solve the second verification problem—determining the initial condition of $\mathbf{s}^i \in \mathcal{S}_0$ such that the system may become unsafe while keeping the initial conditions of all other states in $\mathcal{S}_0$ unchanged—the approach starts from the initial safe condition. It then increases the upper bound of $\mathbf{s}^i$ by some $\boldsymbol{\delta}$ (i.e., $\mathbf{s}^i = \mathbf{s}^i + \boldsymbol{\delta}$). The safety of the system is then determined. If the system's safety is uncertain, the procedure terminates. If the system is still verified safe, the value of $\boldsymbol{\delta}$ is increased.

Despite its emphasis on linear systems, the approach in the work of Tran et al. [93] can be extended to nonlinear systems with existing hybrid systems reachability methods (e.g., the zonotope-based reachability algorithm in CORA [5]).

4.4 Model Checking

Like the methods described in Section 4.3, model checking [22] techniques determine a system's set of reachable states. Model checking, however, requires a distinct problem structure and can thus be considered a separate subfield [99]. Specifically, model checking methods formalize a system as a state-transition graph. This graph is traversed to determine if the system reaches any state in which the specification is violated. If a property fails, a counterexample is typically generated in the form of a sequence of states [15].

Model checking methods generally require four values to be specified: the system's state space $\mathcal{S}$, the set of initial states $\mathcal{S}_0 \subseteq \mathcal{S}$, a transition relation T that specifies the states to which a state $\mathbf{s}$ can transition in a single step, and a property to be verified [31].

Because RL problems are structured as MDPs, these four values can be naturally defined for DRL verification. Recall from Section 2.1 that an MDP is a collection of objects $\mathcal{M} = \langle \mathcal{T}, \mathcal{S}, \mathcal{A}, \mathbf{P}, \mathbf{R}, \gamma \rangle$. We can then see the state space $\mathcal{S}$ is given by the MDP's construction. The set of initial states is then some subset of $\mathcal{S}$. The transition relation T can be defined from the state transition probability matrix $\mathbf{P}$. For example, $T(\mathbf{s}, \mathbf{s}') = false$ if $\sum_{a \in \mathcal{A}} \mathbf{P}(\mathbf{s}, a, \mathbf{s}') = 0$, whereas $T(\mathbf{s}, \mathbf{s}') = true$ if $\sum_{a \in \mathcal{A}} \mathbf{P}(\mathbf{s}, a, \mathbf{s}') > 0$ [31]. Finally, like reachability analysis methods, properties are typically defined in terms of safety and liveness.

Because model checking can be so naturally applied to MDPs, several of these methods have been proposed for DRL verification [6, 10, 11, 26, 31, 43, 46, 67]. In fact, the first DRL verification method, Verily [46], and its subsequent iterations, whiRL 1.0 and whiRL 2.0, rely on model checking. Specifically, whiRL 1.0 [31] uses existing DNN verification tools (e.g., Marabou [45]) to guarantee the system's safety by searching the transition system for bad states that are reachable from an initial state and the system's liveness by running a nested depth-first search algorithm that looks for reachable cycles that do not include a "good" state [12].

Because a DRL system may yield a transition graph that is impractically or infinitely large, whiRL 1.0 uses a *bounded* model checking method in which the safety and liveness search procedures terminate after some finite number of timesteps k. The query for safety is formulated:

$$\exists\ \mathbf{s}_1 \ldots \mathbf{s}_k \ \ I(\mathbf{s}_1) \wedge \left(\bigwedge_{i=1}^{k-1} T(\mathbf{s}_i, \mathbf{s}_{i+t}) \right) \wedge \left(\bigvee_{i=1}^{k} B(\mathbf{s}_i) \right),$$

where $I(\mathbf{s})$ is a predicate that returns true iff $\mathbf{s}$ is an initial state and $B(\mathbf{s})$ is a predicate that returns true iff $\mathbf{s}$ is an unsafe state.

If, with this query, the verifier (e.g., Marabou) returns any state, the safety property cannot be satisfied; otherwise, the system is verified safe.

The liveness property can be constructed in a similar fashion:

$$\exists\ \mathbf{s}_1 \ldots \mathbf{s}_k \ \ I(\mathbf{s}_1) \wedge \left(\bigwedge_{i=1}^{k-1} T(\mathbf{s}_i, \mathbf{s}_{i+t}) \right) \wedge \left(\bigwedge_{i=1}^{k} \neg G(\mathbf{s}_i) \right) \wedge \left(\bigvee_{i=1}^{k-1} \mathbf{s}_k = \mathbf{s}_i \right),$$

where $G(\mathbf{s})$ is a predicate that returns true iff $\mathbf{s}$ is a "good" state.

If the verifier returns any cycle, the liveness property cannot be satisfied; otherwise, the system is guaranteed to enter a "good" state within k timesteps.

Although whiRL 1.0 can find safety and liveness violations within k steps of an initial state, it cannot prove the absence of these violations. A subsequent version, whiRL 2.0 [6], addresses this limitation by employing both bounded model checking and k-induction [23]. Specifically, whiRL 2.0's iterative procedure alternates between bounded model checking and k-induction queries until the property is refuted, the property is proved, or the query exceeds a timeout threshold.

k-Induction is fundamentally similar to bounded model checking with an important difference—the former is used to prove a property, whereas the latter is used to refute a property. Intuitively, k-induction looks for state sequences of length k for which the property is violated. Importantly, the k-induction step does not require the search process to be initiated from an initial state $\mathbf{s} \in \mathcal{S}_0$. Instead, the procedure can begin from any state in $\mathcal{S}$. Thus, if the verifier (e.g., Marabou) finds a violating sequence that does not start at a state in $\mathcal{S}_0$, the value of k must be increased. If the verifier cannot find such a sequence, the property is proved.

5 ADDRESSING STOCHASTICITY

Whether due to environmental changes, noisy sensors, or hardware faults, DRL methods typically contend with some amount of stochasticity. Because this stochasticity can manifest in the form of suboptimal or even dangerous agent actions, verification methods must often prove that policies are robust to unexpected outcomes. These guarantees are especially important in safety-critical domains with high human costs such as healthcare [41, 108] or autonomous driving [47, 50, 79].

Thus, in the work of Bacci and Parker [10], a method called *MOSAIC* (Model Safe Intelligent Control) is used to prove probabilistic guarantees for DRL systems. Specifically, MOSAIC models the runtime execution of a DRL controller as a continuous-space **Discrete-Time Markov Process (DTMP)**. To make the continuous space verification feasible, the approach builds and iteratively refines an abstraction of the model. It then uses standard probabilistic model checking methods on that abstraction to establish probabilistic guarantees on safe behavior over a finite time horizon.

To model the execution of the DRL controller as a DTMP, MOSAIC requires a controller policy, an environment model, and a controller fault model. MOSAIC's controller policy is defined in standard RL terms; it is a function $\pi : \mathbf{s} \rightarrow a$ that selects the agent's action a for a given state $\mathbf{s}$. The environment model is a function $E : \mathbf{s} \times a \rightarrow \mathbf{s}$ that returns the state of the system after one timestep. The controller fault model is a function $f : \mathbf{A} \rightarrow Dist(\mathbf{A})$ that describes, for each possible controller action, the actions that may actually result and their respective probabilities. In other words, because the policy's selected action may not be executed (e.g., due to hardware faults), the fault model characterizes the respective probabilities of all actions.

The agent's behavior can then be formally modeled as a DTMP for which the probability of transitioning from state $\mathbf{s}$ to $\mathbf{s}'$ can be calculated:

$$\mathbb{P}(\mathbf{s}, \mathbf{s}') = \sum \{ f(\pi(\mathbf{s}))(a) \mid a \in \mathbf{A} \;\; s.t. \; E(\mathbf{s}, a) = \mathbf{s}' \}. \tag{5}$$

In other words, for each state $\mathbf{s}$, the action is determined by the policy $\pi(\mathbf{s})$. However, because we cannot guarantee the policy's action will be executed, all possible actions that lead to $\mathbf{s}'$ from $\mathbf{s}$ must be considered; (5) is the combined probability of these actions. The probability that any single action is executed is given by the support of the distribution determined by the controller fault model $f(\pi(\mathbf{s}))$. Whether any a taken in $\mathbf{s}$ results in $\mathbf{s}'$ is determined by $E(\mathbf{s}, a)$.

Given the DTMP, MOSAIC can determine the probability of reaching failure states within k timesteps when starting in state $\mathbf{s}$:

$$\mathbb{P}_{\mathbf{s}}(\diamond^{\leq k} fail) = \mathbb{P}_{\mathbf{s}}(\{\mathbf{s}_0\mathbf{s}_1\mathbf{s}_2 \cdots \in \mathcal{P}(\mathbf{s}) \mid \mathbf{s}_i \models fail \text{ for some } 0 \leq i \leq k\}),$$

where $\mathcal{P}(\mathbf{s})$ is the set of all paths starting in state $\mathbf{s}$, a path is a sequence of states $\mathbf{s}_0, \mathbf{s}_1\mathbf{s}_2, \ldots$ through the model, $\mathbb{P}_{\mathbf{s}}$ is the probability space over $\mathcal{P}(\mathbf{s})$, and $\mathbf{s}_i \models fail$ means $\mathbf{s}_i$ is a failure state.

Because the execution model is discrete time and branches in a finite manner, the reachability probabilities can be directly computed using recursion:

$$\mathbb{P}_{\mathbf{s}}(\diamond^{\leq k} fail) = \begin{cases} 1 & \text{if } \mathbf{s} \models fail \\ 0 & \text{if } \mathbf{s} \not\models fail \wedge k = 0 \\ \sum_{\mathbf{s}' \in Supp(\mathbb{P}(\mathbf{s}, \cdot))} \mathbb{P}(\mathbf{s}, \mathbf{s}') \cdot \mathbb{P}_{\mathbf{s}'}(\diamond^{\leq k-1} fail) & \text{otherwise,} \end{cases}$$

where $Supp(\mu) = \{x \in \mathcal{X} : \mu(x) > 0\}$ is the support of μ and μ is the set of discrete probability distributions over the set $\mathcal{X}$, i.e., $\mu : \mathcal{X} \to [0, 1]$ and $\sum_{x \in \mathcal{X}} \mu(x) = 1$.

With a method for calculating the probability of reaching a failure state within k timesteps when starting in $\mathbf{s}$, MOSAIC solves two verification problems. First, the approach can determine the subset of initial states for which the probability of a failure is below some threshold $\mathcal{S}_0^{safe} = \{\mathbf{s} \in \mathcal{S}_0 \mid \mathbb{P}_{\mathbf{s}}(\diamond^{\leq k} fail) < p_{safe}\}$. Second, for some set of states (likely $\mathcal{S}_0$ or a subset of it), MOSAIC can determine the worst-case error probability: $p^+_{\mathcal{S}'} = \sup\{\mathbb{P}_{\mathbf{s}}(\diamond^{\leq k} fail) \mid \mathbf{s} \in \mathcal{S}'\}$.

The principles underlying MOSAIC were further applied to verifying stochastic policies in the work of Bacci and Parker [11]. Stochastic (or probabilistic) policies choose an action according to a probability distribution defined by the policy for that state $\pi(a \mid s) = \mathbb{P}[a_t = a \mid s_t = s]$. Among other things, stochastic policies are useful for mitigating risk in adversarial environments and managing the tradeoff between exploration and exploitation.

Like MOSAIC, this approach formally models the execution of the DRL policy as a DTMP. The fundamental difference between the methods is their respective abstraction approaches. In other words, both methods are designed to verify a set of inputs over continuous state spaces. Practically, this means finite abstraction must be used to make the problem tractable. However, the complexity induced by the stochastic policies in the work of Bacci and Parker [11] requires a more sophisticated abstraction method.

MOSAIC constructs the finite-state abstraction as an MDP. By computing the maximum probabilities of reaching failure states in the MDP (e.g., with PRISM [55]), MOSAIC is able to determine the upper bounds on the actual probabilities in the concrete model. However, because the action distributions specified by stochastic DRL policies can vary considerably across states, an MDP abstraction is not appropriate for the approach in the work of Bacci and Parker [11]. Thus, a novel abstraction based on **Interval Markov Decision Processes (IMDPs)** is introduced. In an IMDP, transitions are labeled with intervals of probabilities that represent the range of possible events that can occur. After abstracting the execution into IMDPs, these Markov processes are solved over a finite time horizon with a probabilistic model checker (e.g., PRISM) that produces upper bounds on the actual probability of the agent landing in an unsafe state.

6 PROPERTIES TO VERIFY

As we have detailed, the majority of DRL verification methods are designed to prove or disprove a system's safety and/or liveness. However, it is often prudent to verify other properties. For example, applications in robotics typically require stability [2, 14, 53], whereas many control systems need robustness guarantees [28, 44]. In this section, we first describe methods for certifying a policy is robust to input perturbations [27, 36, 43, 54, 105]. Next, we detail approaches suitable for verifying a diverse set of properties [13, 24, 27, 36, 100]. We then describe methods for improving specifications [24, 27]. Then, in Table 1, we enumerate the most popular DRL verification testing environments/tasks. Finally, we define the metrics used to evaluate DRL verification methods.

6.1 Certifying Robustness Against Input Perturbations

Learned policies may have to contend with several sources of input perturbations such as modeling uncertainty [91] or adversarial attacks [38, 109]. Thus, several methods have been designed to verify that a DRL system is robust to these perturbations [27, 36, 43, 54, 105]. For example, in the work of Jin et al. [43], a perturbation vector is used to over-approximate the set of reachable states. Specifically, the approach trains the DRL system on finite abstract domains instead of the concrete continuous system states. In other words, instead of providing continuous values to the DNN controller during learning (e.g., $(0.23, 1.4)$), the method transforms the state into an interval vector (e.g., $([0.0, 0.25], [1.0, 1.5])$. The controller then determines which action the agent should take based on the abstract interval. Finally, the action is applied to the concrete continuous state. This approach is feasible because a trained controller usually adopts the same action for "adjacent" concrete states [10] where two states $\mathbf{s}$ and $\mathbf{s}'$ are "adjacent" with respect to an ℓ_p norm distance δ iff $||\mathbf{s} - \mathbf{s}'||_p < \delta$.

To verify the system's properties, the approach builds an abstract-state transition system K, which can be verified with existing model checkers [30]. To build K, the approach first abstracts a continuous state into a finite state (as was done during learning). Then the abstract state $\mathbf{s}$ is provided as an input to the trained controller that produces an action a. The action a is then applied to the abstract state $\mathbf{s}$ to produce the interval vector $\mathbf{v}$. To provide assurances against input perturbations, a perturbation vector $\boldsymbol{\epsilon}$ is used to expand the set of reachable states—that is, $[l_i - \boldsymbol{\epsilon}_i, u_i + \boldsymbol{\epsilon}_i]$, where $[l_i, u_i]$ is a vector in $\mathbf{v}$—and thereby over-approximate $\mathbf{v}$. To build the full transition system K, this methodology is applied to successive states with breadth-first search.

Another approach, CROP (Certifying Robust Policies for Reinforcement Learning through Functional Smoothing) [105], is designed to certify the robustness of Q-learning-based methods (e.g., deep Q-learning described in Section 2.2) against the standard Q-learning adversarial attack [38, 52, 109]. Such an attack is defined by an adversary seeking to induce the selection of sub-optimal actions by applying an ℓ_2-bounded perturbation $\mathcal{B}^{\epsilon} = \{\delta \in \mathbb{R}^n \mid ||\delta||_2 \leq \epsilon\}$ to the input state during decision (test) time: $a = \pi(s + \delta)$.

CROP defines robustness certification for per-state actions as the maximum perturbation magnitude $\bar{\epsilon}$ such that for any perturbation $\delta \in \mathcal{B}^{\bar{\epsilon}}$, the action for the perturbed state is the same as the action for the unperturbed state—that is, $\pi(s + \delta) = \pi(s)\ \forall \delta \in \mathcal{B}^{\bar{\epsilon}}$.

The per-state action certification algorithm proceeds as follows. Given a DQN Q_π, a smoothed function $\widetilde{Q}_\pi$ is defined by drawing random noise from a Gaussian distribution at each timestep

$t \in \mathcal{T}$ for each action $a \in \mathcal{A}$:

$$\widetilde{Q}_\pi(s_t, a) := \mathbb{E}_{\Delta_t \sim \mathcal{N}(0,\sigma^2 I_N)} Q_\pi(s_t + \Delta_t, a) \ \forall s_t \in \mathcal{S}, a \in \mathcal{A},$$

where $\mathcal{N}(0, \sigma^2 I_N)$ is the Gaussian distribution, I_N is the identity matrix, and σ is the smoothing parameter.

The maximum perturbation magnitude $\bar{\epsilon}$ is found by exploiting the fact that the smoothed function $\widetilde{Q}_\pi$ is L-Lipschitz continuous. From this property, the lower bound r_t of the maximum perturbation magnitude $\bar{\epsilon}(s_t)$ (i.e., $r_t < \bar{\epsilon}(s_t)$) can be defined such that if $\|\delta_t\|_2 \leq r_t$, then the action is guaranteed not to change—that is, $\tilde{\pi}(s_t + \delta_t) = \tilde{\pi}(s_t)$:

$$r_t = \frac{\sigma}{2}\left(\Phi^{-1}\left(\frac{\widetilde{Q}_\pi(s_t, a_1) - v_{min}}{v_{max} - v_{min}}\right) - \Phi^{-1}\left(\frac{\widetilde{Q}_\pi(s_t, a_2) - v_{min}}{v_{max} - v_{min}}\right)\right), \tag{6}$$

where v_{min} and v_{max} are the minimum and maximum possible Q values given by the (non-smoothed) DQN, respectively (i.e., $Q_\pi : \mathcal{S} \times \mathcal{A} \rightarrow [v_{min}, v_{max}]$), Φ is the cumulative distribution function, a_1 is the action with the highest $\widetilde{Q}_\pi$ value in state s_t, and a_2 is the action with the second highest $\widetilde{Q}_\pi$ value in state s_t. Note that in each state, the smoothed policy $\tilde{\pi}$ selects a_1—that is, $\tilde{\pi}(s) = a_1$.

In addition to certifying per-state actions, CROP can also find the lower bound of the perturbed cumulative reward $\underline{g}$ such that $\underline{g} \leq g_\epsilon(\pi)$, where $g_\epsilon(\pi) = \sum_{t=0}^{\infty} \gamma^t \mathbf{R}(s_t, \pi(s_t + \delta_t))$ is the cumulative reward under perturbations $\delta_t \in \mathcal{B}^\epsilon$.

Certifying the cumulative reward can be done with either a global or local smoothing approach. The former considers the entire state trajectory a function to be smoothed. Although efficient, this approach produces a relatively loose certification bound. Local smoothing builds on principles from the per-state action certification procedure to find a tight lower bound for $g_\epsilon(\pi)$.

Notice that for a trajectory of states $\mathbf{h}$ generated by following the smoothed policy $\tilde{\pi}$, a certified radius can be computed for each timestep using (6). Recall that when the perturbation magnitude $\epsilon < r_t$, the optimal action at time t is unchanged. This means that when $\epsilon < \min_{t=0}^{\mathbf{h}-1} r_t$, none of the actions in the trajectory $\mathbf{h}$ will change. Therefore, when $\epsilon < \min_{t=0}^{\mathbf{h}-1} r_t$, the lower bound of the perturbed cumulative reward is equal to the return of $\mathbf{h}$ in a deterministic environment.

Changing the perturbation magnitude ϵ affects both the number of timesteps in which the action might deviate from the optimal action and the number of actions that can be taken in each timestep. To measure these effects on the cumulative reward CROP generalizes (6):

$$r_t^k = \frac{\sigma}{2}\left(\Phi^{-1}\left(\frac{\widetilde{Q}_\pi(s_t, a_1) - v_{min}}{v_{max} - v_{min}}\right) - \Phi^{-1}\left(\frac{\widetilde{Q}_\pi(s_t, a_{k+1}) - v_{min}}{v_{max} - v_{min}}\right)\right), 1 \leq k < |\mathcal{A}|, \tag{7}$$

where r_t^k denotes the radius such that if $\epsilon < r_t^k$, the set of actions that can be taken in state s_t are the k actions for which the $\widetilde{Q}_\pi$ value is highest (note that (6) is equivalent to r_t^1) and a_{k+1} is the action with the $(k+1)$-th highest $\widetilde{Q}_\pi$ value in state s_t.

For any ϵ, in each state, CROP uses (7) to compute all possible actions under perturbations in $\mathcal{B}^\epsilon$. Then the approach exhaustively traverses all trajectories resultant from these actions. Finally, the perturbed cumulative reward $g_\epsilon(\pi)$ is lower bounded by the minimum return over all of these possible trajectories.

6.2 Verifying Alternative Properties

The highly structured nature of the decision trees generated by Viper [13] makes the learned policies amenable to the verification of correctness, stability, and robustness properties. Consider, for example, a toy Pong environment in which there is a bouncing ball and a player-controlled paddle located on the bottom of the screen. If the ball hits the top of the screen, the side of the screen, or the paddle, it is reflected. If the ball hits the bottom of the screen where the paddle is not present, the game is over.

The game's action space comprises the paddle's possible movements {left, right, stay}, whereas its states are defined by a tuple $(x, y, v_x, v_y, x_p) \in \mathbb{R}^5$. In each state, the values (x, y) indicate the ball's position, where $x \in [0, x_{max}]$ and $y \in [0, y_{max}]$. The values (v_x, v_y) represent the ball's velocity, where $v_x, v_y \in [-v_{max}, v_{max}]$. Finally, x_p is the paddle's position, where $x_p \in [0, x_{max}]$.

The correctness property for toy Pong specifies that the controller never loses—that is, the ball never hits the bottom of the screen at a position where the paddle is not present. More formally, assuming the system's initial state is safe $y \in \mathcal{Y}_0 = [y_{max}/2, y_{max}]$, then the system should avoid an unsafe region $y = 0 \wedge (x \leq x_p - l \vee x \geq x_p + l)$, where l is half the paddle length.

This property can be proved with an inductive invariant. Specifically, because the game dynamics ensure that the ball's velocity is conserved in each direction, the assumption that the ball's speed in the y direction is lower bounded (i.e., $|v_y| > v_{min}$) is equivalent to assuming that the initial y velocity is in $[-v_{max}, -v_{min}] \cup [v_{min}, v_{max}]$. Thus, correctness can be proved with the inductive invariant: as long as the ball starts in a safe state, then it re-enters a safe state after at most $t_{max} = \lceil 2y_{max}/v_{min} \rceil$ steps.

Because both the system dynamics $f : \mathbf{s} \times a \rightarrow \mathbf{s}$ and the controller: $\pi : \mathbf{s} \rightarrow a$ are piecewise linear, their joint dynamics $f_\pi(s) = f(\mathbf{s}, \pi(\mathbf{s}))$ must be as well. If $\mathcal{S} = \mathcal{S}_1 \cup \cdots \cup \mathcal{S}_k$ is a partition of the state space such that $f_\pi(\mathbf{s}) = f_i(\mathbf{s}) = \beta_i^T \mathbf{s}$ for all $\mathbf{s} \in \mathcal{S}_i$, then let $\mathbf{s}_t$ denote the state of the system at time $t \in \{0, \ldots, t_{max}\}$. Note that β is a hyperparameter required by Dagger. For Viper, $\beta_i^T \mathbf{s}$ indicates that during the first iteration of Q-Dagger, the action is determined by the expert policy; otherwise, the action is determined by the learned policy. Thus, the system dynamics can be formalized as

$$\phi_t = \bigvee_{i=1}^{k} (\mathbf{s}_{t-1} \in \mathcal{S}_i \Rightarrow \mathbf{s}_t = \beta_i^T \mathbf{s}_{t-1}) \qquad \forall t \in \{1, \ldots, t_{max}\},$$

and the correctness property of the inductive invariant can be formalized as

$$\psi = \left(\bigwedge_{t=1}^{t_{max}} \phi_t \right) \wedge \psi_0 \Rightarrow \bigvee_{t=1}^{t_{max}} \psi_t,$$

where $\psi_t = (\mathbf{s}_t \in \mathcal{Y}_0)$.

Because ψ is composed of conjunctions and disjunctions of linear inequalities, an SMT solver [66] can determine whether it is satisfiable.

The stability of the decision tree policies learned by Viper can also be verified. In control theory, a system f is stable if it asymptotically reaches its goal [89]. Given a continuous-time dynamical system with states $\mathbf{s} \in \mathcal{S} = \mathbb{R}^n$, actions $\mathbf{a} \in \mathcal{A} = \mathbb{R}^m$, and dynamics $\dot{\mathbf{s}} = f(\mathbf{s}, \mathbf{a})$, the system $f_\pi(\mathbf{s}) = f(\mathbf{s}, \pi(\mathbf{s}))$ is considered stable for policy $\pi : \mathbf{s} \rightarrow \mathbf{a}$ if there is a region of attraction $\mathcal{U} \subseteq \mathbb{R}^n$ containing 0 such that for any initial state $\mathbf{s}_0 \in \mathcal{U}$, we have $\lim_{t\rightarrow\infty} \mathbf{s}(t) = 0$, where $\mathbf{s}(t)$ is a solution to $\dot{\mathbf{s}} = f(\mathbf{s}, \mathbf{a})$ with initial condition $\mathbf{s}(0) = \mathbf{s}_0$. In other words, there is an invariant set such that all trajectories starting inside this set converge to the equilibrium point [13, 48].

When f_π is nonlinear, stability can be verified by finding a Lyapunov function $V : \mathcal{S} \rightarrow \mathbb{R}$ that satisfies $V(s) > 0 \ \forall s \in \mathcal{U} \setminus \{0\}$, $V(0) = 0$, and $\dot{V}(s) = (\nabla V)(s) \cdot f(s) < 0 \ \forall s \in \mathcal{U} \setminus \{0\}$ [90]. When f_π is polynomial—as is required by Viper—sum-of-squares optimization can be used to devise a candidate Lyapunov function, check the Lyapunov properties, and thus verify stability.

Finally, the robustness of Viper's policies can be verified. A policy π is said to be ϵ robust at $\mathbf{s}_0 \in \mathcal{S} = \mathbb{R}^d$ if

$$\pi(\mathbf{s}) = \pi(\mathbf{s}_0) \;\; \forall \mathbf{s} \in B_\infty(\mathbf{s}_0, \epsilon),$$

where $B_\infty(\mathbf{s}_0, \epsilon)$ is the L_∞-ball of radius ϵ around $\mathbf{s}_0$.

Because π is a decision tree, the largest ϵ such that π is ϵ-robust at $\mathbf{s}_0$ (denoted $\epsilon(\mathbf{s}_0; \pi)$) can be computed efficiently. Specifically, the distance from $\mathbf{s}_0$ to the closest point $\mathbf{s} \in \mathcal{S}$ (in L_∞ norm) such that $\mathbf{s}$ is in the set of states routed by the decision tree to a leaf node $\ell \in \text{leaves}(\pi)$ labeled with action $a_\ell \neq \pi(\mathbf{s}_0)$—that is, the closest state for which the action selected by π is different from the action selected for $\mathbf{s}_0$—can be computed with a linear equation:

$$\epsilon(\mathbf{s}_0; \ell, \pi) = \max_{\mathbf{s} \in \mathcal{S}, \epsilon \in \mathbb{R}_+} \epsilon \quad \text{subj. to} \left(\bigwedge_{n \in \text{path}(\ell; \pi)} \delta_n s_{i_n} \leq t_n \right) \wedge \left(\bigwedge_{i \in [d]} |s_i - (s_0)_i| \leq \epsilon \right),$$

where path$(\ell; \pi)$ is the set of nodes on the path from the root of π to the leaf node ℓ, $\delta_n = 1$ if n is a left child and $\delta_n = -1$ otherwise, i_n is the feature index of node n, and t_n is the threshold of n. Then,

$$\epsilon(\mathbf{s}_0; \pi) = \operatorname*{argmin}_{\ell \in \text{leaves}(\pi)} \begin{cases} \infty & \text{if } a_\ell = \pi(\mathbf{s}_0) \\ \epsilon(\mathbf{s}_0; \ell, \pi) & \text{otherwise.} \end{cases}$$

Like Viper, Pirl [101] also learns highly structured policies and thus allows for the verification of interesting properties such as smoothness and universal bounds. Pirl's efficacy was tested in the Torcs (The Open Racing Car Simulator) environment [106]. In Torcs, a controller drives a car on a racetrack by managing acceleration, brake, clutch, gear, and steering. The controller's decisions are based on data from up to 89 sensors. An example of a smoothness property verifiable by Pirl is informally "if the sum of the consecutive differences of the last six RPM sensor values is less than 0.006, then the acceleration actions calculated at the last and penultimate step will not differ by more than 0.49." Such a property can be expressed with Pirl's DSL:

$$\sum_{i=k}^{k+5} \|\mathbf{peek}(h_{RPM}, i{+}1) - \mathbf{peek}(h_{RPM}, i)\| < 0.006 \Rightarrow \|\mathbf{peek}(h_{Acl}, k{+}1) - \mathbf{peek}(h_{Acl}, k)\| < 0.49, \;\forall k.$$

Universal bounds properties define global bounds for action values. For example:

$$(0 \leq \mathbf{peek}(h_{RPM}, i) \leq 1 \wedge -1 \leq \mathbf{peek}(h_{pos}, i) \leq 1) \Rightarrow$$
$$(\|\mathbf{peek}(h_{steer}, i)\| < 101.08 \wedge -54.53 < \mathbf{peek}(h_{Acl}, i) < 53.03) \;\forall i.$$

6.3 Methods for Improving Specifications

DRL environments are often quite complex. This makes it difficult for human experts to rigorously define all of a system's desirable properties *a priori*. Thus, methods have been developed to make writing specifications easier [24, 27]. One such approach proposes the use of "behavioral properties" to encode general, "rational" behaviors [24]. For example, consider this typical safety property which specifies that a network's output should lie within an interval:

$$x_0 \in [l_0, u_0] \wedge \cdots \wedge x_n \in [l_n, u_n] \Rightarrow y_j \in [l_y, u_y],$$

where $x_k \in \mathcal{X}$ is an input to the network and y_j is a generic output. A behavioral property reformulates the specification as such:

$$x_0 \in [l_0, u_0] \wedge \cdots \wedge x_n \in [l_n, u_n] \Rightarrow y_j > y_i. \tag{8}$$

This behavioral property ensures that one action (y_j) is always preferred over another (y_i) for a given input set ($\mathcal{X}$). For example, a collision avoidance system's behavioral property might specify a vehicle should never turn right if there is an obstacle close to the right. If $y_{left} > y_{right}$, it is guaranteed that the agent will not turn right and the property is verified.

As demonstrated by (8), verifying behavioral properties requires proving an inequality for abstract output intervals. This means if the output bounds overlap—for example, if $y_{left} = [0.25, 1.5]$ and $y_{right} = [0.3, 1.9]$—we cannot determine whether the behavioral property is satisfied. Thus, it is critical to tightly bound each node's output. To generate tight bounds, a novel method called *ProVe* (Property Verifier) is introduced. ProVe relies on iterative refinement [103], an approach that subdivides the input area into smaller subareas and computes the corresponding output bounds such that the union of the subareas' output bounds results in a tighter bound than the undivided, original area. The standard iterative refinement procedure requires significant computational time. ProVe, however, parallelizes the computation such that the bounds can be feasibly computed online inside the training loop.

Another method [27] introduces building blocks from which complex properties can be constructed by domain experts unaccustomed to writing formal specifications. These properties are then—in addition to the DNN to be verified—automatically encoded as an MILP that can be verified with existing tools.

To encode the DNN, the framework defines each layer as a set of constraints. For example, encoding a fully connected layer with a ReLU activation $\mathbf{y} = \text{ReLU}(\mathbf{Ax}+\mathbf{b})$ requires the weights $\mathbf{A}$ and biases $\mathbf{b}$ from the trained model, the inputs $\mathbf{x}$, the outputs $\mathbf{y}$, the ReLU state indicator variable $\mathbf{z}$, and an auxiliary real variable $\mathbf{s}$ (size $|\mathbf{y}|$). The constraints between the variables can then be expressed:

$$\begin{aligned} y_i &= \sum_j A_{i,j} x_j + b_i - s_i \\ z_i &= 1 \Rightarrow s_i = 0 \\ z_i &= 0 \Rightarrow y_i = 0, \quad \forall i \in \{1, \ldots, n\}, \end{aligned}$$

where z_i is a Boolean variable and y_i, s_i are both positive real variables.

Next, the specifications are encoded. Instead of requiring a human expert to express properties using complex logical formulas, the framework introduces four building blocks from which specifications can be written more naturally. The first building block allows a human expert to express that an input or output must fall within a range of values (constant values can be trivially encoded with this building block). The second allows linear relationships (e.g., $a_1x_1 + a_2x_2 + \cdots \leq b$) to be constructed. With the third building block, a human expert can express which input or output should have the greatest value. For example, as described previously, a DQN outputs the Q value for every action that can be taken from the input state. This third building block allows the human expert to define which action should have the highest Q value. The final building block is used to specify that a variable must be discrete (DNNs typically assume values can be continuous).

To improve the efficiency of the verification procedure, domain constraints can also be encoded. Domain constraints are distinct from properties, as the former are inherent characteristics of the system and the latter represent the expected behavior of the agent. For example, a video client's domain constraint might specify that the client cannot buffer more data than the maximum buffer capacity. In this way, domain constraints can significantly reduce the search space.

These constraints can be either manually specified or automatically discovered from the training data. Specifically, the method declares an input or output variable to be discrete if the number of distinct values is lower than a configurable threshold; otherwise, the approach defines lower and upper bounds for the variable based upon its lowest and highest values.

6.4 DRL Verification Testing Environments, Tasks, Property Types, and Specifications

To demonstrate their respective capabilities, DRL verification methods generally repurpose typical (i.e., non-verification) DRL testing environments. In Table 1, we list the most popular of these environments and their associated control tasks, property types, and specifications.

Table 1. List of the Most Common DRL Verification Testing Environments Including Their Respective Control Tasks, Property Types, and Specifications

DRL Verification Test Environments			
Environment	**Control Task**	**Property Type**	**Specification Description**
Cart pole [18]	Keep pole upright	Safety	Pole angle must stay within specified range
Cart pole	Keep pole upright	Safety	Cart displacement must stay within specified range
Cart pole	Keep pole upright	Stability	Stability of policy π with respect to the degree-5 Taylor approximation of cart-pole dynamics
Atari pong [18]	Deflect ball into opponent's goal	Robustness	$\pi(\mathbf{s}) = \pi(\mathbf{s}_0)$ $(\forall \mathbf{s} \in B_\infty(\mathbf{s}_0, \epsilon))$, where $B_\infty(\mathbf{s}_0, \epsilon)$ is the L_∞-ball of radius ϵ around $\mathbf{s}_0$
TORCS (practice mode) [106]	Race car around track	Smoothness	Control values do not change radically
TORCS (practice mode)	Race car around track	Universal bounds	Action values do not exceed global bounds (assuming reasonable bounds on input values)
TORCS (practice mode)	Race car around track	Safety	Do not crash into obstacles
Quadcopter (e.g., [76])	Reach goal state	Safety	Quadcopter does not collide with any object
Inverted pendulum [18]	Swing pendulum into upright position	Safety	Pendulum angle does not exceed a specified value
Mountain car [18]	Accelerate an under-powered car in a valley to reach the top of the right hill	Safety	Car does not go over the crest of the left hill
Mountain car	Accelerate an under-powered car in a valley to reach the top of the right hill	Liveness	Eventually the car reaches the top of the right hill
Adaptive cruise control [32]	Maintain safe distance between cars	Safety	Do not crash into lead car regardless of any behavior it exhibits (e.g., deceleration)
n-Car platoon [78]	Coordinate n vehicles to stay within specified distance of each other	Safety	Do not allow cars to get closer than threshold
Pensieve video streamer [60]	Determine the bitrate with which the next video chunk should be downloaded	Safety	When the chunk download history indicates excellent network conditions, the resolution at which chunks are requested should increase
Pensieve video streamer	Determine the bitrate with which the next video chunk should be downloaded	Safety	When the chunk download history indicates poor network conditions, the resolution at which chunks are requested should decrease

Pensieve video streamer	Determine the bitrate with which the next video chunk should be downloaded	Robustness	Decision does not change when some input features receive arbitrary values
Pensieve video streamer	Determine the bitrate with which the next video chunk should be downloaded	Robustness	Decision does not change when feature values change by a factor of no more than $\epsilon \cdot k_i$
Aurora congestion [42]	Determine whether Internet sending rate should be increased, decreased, or maintained	Liveness	When the controller observes history of excellent network conditions, the sending rate should eventually increase
Aurora congestion	Determine whether Internet sending rate should be increased, decreased, or maintained	Liveness	When the controller observes history of poor network conditions, the sending rate should eventually decrease
Advanced emergency braking [93]	Stop the car to avoid a collision while not stopping too far from the obstacle	Safety	Do not hit the obstacle
Bouncing ball [40]	Keep the ball bouncing by either hitting the ball downward with a paddle or by doing nothing	Safety	Never allow the ball to stop bouncing
Atari freeway [18]	Guide chicken across a road while avoiding traffic	Robustness	Bounded perturbation δ to the input state (i.e., $a = \pi(s + \delta)$) does not affect policy action
Cart pole	Keep pole upright	Robustness	Bounded perturbation δ to the input state (i.e., $a = \pi(s + \delta)$) does not affect policy action
Mountain car	Accelerate an under-powered car in a valley to reach the top of the right hill	Robustness	Bounded perturbation δ to the input state (i.e., $a = \pi(s + \delta)$) does not affect policy action

6.5 Evaluating DRL Verification Methods

Guaranteeing a learned policy's properties is critical for many real-world applications. Verification, however, comes with costs. Because DRL verifiers achieve their guarantees in distinct ways, these costs differ across method classes. Thus, evaluating these classes requires different sets of metrics. In this section, we describe how to measure the tradeoffs associated with each type of DRL verifier.

Notice that the verifiers that reduce DRL's DNN to a simpler-to-verify form (Section 4.1) may have to sacrifice performance to achieve their guarantees. Specifically, these approaches attempt to find a highly structured (and thus easily verified) policy that best approximates the optimal policy (i.e., the best policy irrespective of the specification). Importantly, the space of highly structured policies may not include the optimal policy. Moreover, the best policy in the space of highly structured policies may not satisfy the specification. In this case, a worse—but specification-satisfying—policy must be selected. Thus, evaluating these methods requires two metrics: (1) the difference in total reward earned by the verified policy and the total reward earned by the optimal policy, and (2) a count of the times the optimal policy violates the specification during its execution.

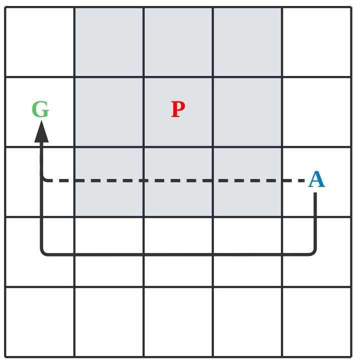

Fig. 4. Grid world wherein an agent (A) maximizes its reward by taking the fewest actions necessary to reach the goal (G). The specification prohibits the agent from entering any state surrounding the pit (P). The optimal policy (dash lined) violates this specification. A shield (solid line) intervenes to ensure the specification is never violated but negatively affects the agent's reward.

Depending on the application, a large value for the former or a small value for the latter may negate the benefits of the verified policy's guarantees.

Because methods that construct a verified barrier function (Section 4.2) may prevent the agent from taking optimal actions, they are also evaluated with these two metrics. Consider, for example, the grid world environment represented in Figure 4. The objective of the agent (denoted by "A") is to arrive in the goal state (denoted by "G") by taking the fewest possible actions. The specification prohibits the agent from entering any state surrounding the pit (denoted by "P"). Notice that the optimal policy (denoted by the dashed line) gives actions that violate the specification. The shielded policy (denoted by the solid line) avoids violating the specification but requires the agent to take two extra actions. In this way, the shielded policy is worse than the optimal policy. Determining the effectiveness of the shield thus requires evaluating how the shielded policy performs relative to the optimal policy. Because learning a barrier function can require significant computational overhead, the runtime required to construct the shield is also often measured.

Reachability-based methods (Section 4.3) and model checking methods (Section 4.4) are DRL variants of typical DNN verification approaches. In other words, as described in their respective sections, reachability and model checking methods exist for non-DRL contexts and have been adapted for DRL verification. As such, standard DNN verification metrics [59] are typically used to assess these two method classes. The first metric—verification time—is important because in real-world systems practical considerations like runtime can significantly affect the viability of an approach. Next, if a method is incomplete, the number of properties that cannot be definitively proved or disproved for a given initial condition is counted. The value of this metric is evident; the more properties a verifier can guarantee, the greater our confidence in the system. Finally, if the approach is abstraction based, bound tightness is often evaluated because, as described in Section 3.3, this value directly affects which properties can be proved.

7 DISCUSSION AND CONCLUSION

In this article, we provide a rigorous survey of DRL verification. We first established the foundations of DRL and DRL verification. Then, we defined a taxonomy for DRL verification methods that includes reduction to a simpler form, construction of a verified barrier function, reachability analysis, and model checking. We proceeded by detailing many of the most important methods within each category of that taxonomy. Next, we described approaches for dealing with stochasticity.

Finally, we characterized considerations related to writing specifications, enumerated common testing tasks/environments, and defined the metrics used to evaluate DRL verification methods. The source material was largely discovered with the Google Scholar search engine. Given that DRL verification is a relatively nascent field, we emphasized methods published in 2018 or later.

Although there has been significant progress in the field, there are many open research opportunities. For example, approaches like Viper [13] and Pirl [101] are particularly appealing in domains that require interpretability because their learned policies are highly structured. As we described, however, these methods may learn policies that violate the specification. These violating policies are often difficult to repair. Other methods, by contrast, avoid any possibility of constructing a policy that may violate the specification [7, 110]; however, the learned policies are not inherently interpretable. We believe there is opportunity to develop methods that either automatically repair violating yet interpretable policies or learn non-violating interpretable policies.

Next, while the verification of multi-agent systems has not been completely neglected [67], we believe this is an under-explored research area. Multi-agent DRL has been proven capable of making decisions for complex tasks such as large-scale fleet management, traffic light control, and energy sharing optimization [69]. As multi-agent methods continue to be deployed in real-world systems, it is imperative to guarantee their critical properties. We suspect that many of the single-agent verification techniques can extend naturally to the multi-agent setting; however, additional challenges will arise such as increased state and action space dimensionality [19].

We believe there is also significant opportunity to develop methods for verifying stochastic systems. Because DRL is often employed in safety-critical domains with high human costs such as healthcare [108] and autonomous driving [47, 50, 79], it is vital that verification methods elucidate the ways in which an agent might take undesirable actions, regardless of how improbable they may be. As we described in Section 5, there are foundational approaches for managing stochasticity, but we feel that relative to its importance, this complexity has been under-addressed.

Finally, a primary challenge in DRL verification—and, in fact, all of DNN verification—is scalability. We believe there is opportunity to develop methods for improving the computational efficiency of DRL verification and for guaranteeing tighter bounds when using abstraction-based verification.

As DRL approaches continue to proliferate, guaranteeing their critical properties is becoming increasingly important. DRL verification is an emerging field with both deep technical challenges and practical utility. With this survey, we endeavored to both provide the background necessary for new researchers to contribute to the field and to apprise current contributors of the advances in this developing area.

REFERENCES

[1] Pieter Abbeel and Andrew Y. Ng. 2004. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the 21st International Conference on Machine Learning (ICML'04)*. ACM, New York, NY, 1. https://doi.org/10.1145/1015330.1015430

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

[3] Aws Albarghouthi. 2021. Introduction to neural network verification. *arXiv preprint arXiv:2109.10317* (2021).

[4] Aws Albarghouthi. 2021. Neural network verification: Where are we and where do we go from here? *PL Perspectives*. Retrieved May 10, 2023 from https://blog.sigplan.org/2021/11/04/neural-network-verification-where-are-we-and-where-do-we-go-from-here/.

[5] Matthias Althoff. 2015. An introduction to CORA 2015. In *Proceedings of the Workshop on Applied Verification for Continuous and Hybrid Systems*. 120–151.

[6] Guy Amir, Michael Schapira, and Guy Katz. 2021. Towards scalable verification of deep reinforcement learning. In *Proceedings of the 2021 Conference on Formal Methods in Computer Aided Design (FMCAD'21)*. IEEE, Los Alamitos, CA, 193–203.

[7] Greg Anderson, Abhinav Verma, Isil Dillig, and Swarat Chaudhuri. 2020. Neurosymbolic reinforcement learning with formally verified exploration. *Advances in Neural Information Processing Systems* 33 (2020), 6172–6183.

[8] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. 2017. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine* 34, 6 (2017), 26–38.

[9] Edoardo Bacci, Mirco Giacobbe, and David Parker. 2021. Verifying reinforcement learning up to infinity. In *Proceedings of the International Joint Conference on Artificial Intelligence.*

[10] Edoardo Bacci and David Parker. 2020. Probabilistic guarantees for safe deep reinforcement learning. In *Proceedings of the International Conference on Formal Modeling and Analysis of Timed Systems.* 231–248.

[11] Edoardo Bacci and David Parker. 2022. Verified probabilistic policies for deep reinforcement learning. *arXiv preprint arXiv:2201.03698* (2022).

[12] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking.* MIT Press, Cambridge, MA.

[13] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. 2018. Verifiable reinforcement learning via policy extraction. *Advances in Neural Information Processing Systems* 31 (2018), 2499–2509.

[14] Felix Berkenkamp, Matteo Turchetta, Angela Schoellig, and Andreas Krause. 2017. Safe model-based reinforcement learning with stability guarantees. *Advances in Neural Information Processing Systems* 30 (2017), 909–919.

[15] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded model checking. *Advances in Computers* 58 (2003), 1–27.

[16] Christopher M. Bishop and Nasser M. Nasrabadi. 2006. *Pattern Recognition and Machine Learning.* Vol. 4. Springer.

[17] Leo Breiman, Jerome Friedman, Charles J. Stone, and R. A. Olshen. 1984. *Classification and Regression Trees.* Chapman & Hall/CRC.

[18] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *arXiv preprint arXiv:1606.01540* (2016). https://doi.org/10.48550/ARXIV.1606.01540

[19] Lucian Busoniu, Robert Babuska, and Bart De Schutter. 2008. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 38, 2 (2008), 156–172.

[20] Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, and Yisong Yue. 2021. *Neurosymbolic Programming.* Now Publishers.

[21] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. 2013. Flow*: An analyzer for non-linear hybrid systems. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer, Berlin, Germany, 258–263.

[22] Edmund M. Clarke. 1997. Model checking. In *Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science.* 54–56.

[23] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). 2018. *Handbook of Model Checking.* Vol. 10. Springer.

[24] Davide Corsi, Enrico Marchesini, and Alessandro Farinelli. 2021. Formal verification of neural networks for safety-critical tasks in deep reinforcement learning. In *Proceedings of the 37th Conference on Uncertainty in Artificial Intelligence.* 333–343.

[25] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'77).* ACM, New York, NY, 238–252. https://doi.org/10.1145/512950.512973

[26] Florent Delgrange, Ann Nowé, and Guillermo A Pérez. 2021. Distillation of RL policies with formal guarantees via variational abstraction of Markov decision processes (technical report). *arXiv preprint arXiv:2112.09655* (2021).

[27] Arnaud Dethise, Marco Canini, and Nina Narodytska. 2021. Analyzing learning-based networked systems with formal verification. In *Proceedings of the IEEE INFOCOM 2021-IEEE Conference on Computer Communications.* IEEE, Los Alamitos, CA, 1–10.

[28] Priya L. Donti, Melrose Roderick, Mahyar Fazlyab, and J. Zico Kolter. 2020. Enforcing robust control guarantees within neural network policies. *arXiv preprint arXiv:2011.08105* (2020).

[29] Douglas D. Dunlop and Victor R. Basili. 1982. A comparative analysis of functional correctness. *ACM Computing Surveys* 14, 2 (1982), 229–244.

[30] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. 2016. Spot 2.0—A framework for LTL and ω-automata manipulation. In *Proceedings of the International Symposium on Automated Technology for Verification and Analysis.* 122–129.

[31] Tomer Eliyahu, Yafim Kazak, Guy Katz, and Michael Schapira. 2021. Verifying learning-augmented systems. In *Proceedings of the 2021 Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'21).* 305–318.

[32] Nathan Fulton and André Platzer. 2018. Safe reinforcement learning via formal methods: Toward safe control through proof and learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[33] Javier García and Fernando Fernández. 2015. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research* 16, 42 (2015), 1437–1480. http://jmlr.org/papers/v16/garcia15a.html.

[34] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. 2018. AI2: Safety and robustness certification of neural networks with abstract interpretation. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP'18).* IEEE, Los Alamitos, CA, 3–18.

[35] Sven Gowal, Krishnamurthy Dvijotham, Robert Stanforth, Rudy Bunel, Chongli Qin, Jonathan Uesato, Relja Arandjelovic, Timothy Mann, and Pushmeet Kohli. 2019. Scalable verified training for provably robust image classification. In *Proceedings of the IEEE/CVF International Conference on Computer Vision.* 4842–4851.

[36] Akshita Gupta and Inseok Hwang. 2020. Safety verification of model based reinforcement learning controllers. *arXiv preprint arXiv:2010.10740* (2020).

[37] Po-Sen Huang, Robert Stanforth, Johannes Welbl, Chris Dyer, Dani Yogatama, Sven Gowal, Krishnamurthy Dvijotham, and Pushmeet Kohli. 2019. Achieving verified robustness to symbol substitutions via interval bound propagation. *arXiv preprint arXiv:1909.01492* (2019).

[38] Sandy H. Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. 2017. Adversarial attacks on neural network policies. *arXiv preprint arXiv:1702.02284* (2017).

[39] Radoslav Ivanov, James Weimer, Rajeev Alur, George J. Pappas, and Insup Lee. 2019. Verisig: Verifying safety properties of hybrid systems with neural network controllers. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control.* 169–178.

[40] Manfred Jaeger, Peter Gjøl Jensen, Kim Guldstrand Larsen, Axel Legay, Sean Sedwards, and Jakob Haahr Taankvist. 2019. Teaching Stratego to play ball: Optimal synthesis for continuous space MDPs. In *Proceedings of the International Symposium on Automated Technology for Verification and Analysis.* 81–97.

[41] Mahsa Oroojeni Mohammad Javad, Stephen Olusegun Agboola, Kamal Jethwani, Abe Zeid, and Sagar Kamarthi. 2019. A reinforcement learning–based method for management of type 1 diabetes: Exploratory study. *JMIR Diabetes* 4, 3 (2019), e12905.

[42] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. A deep reinforcement learning perspective on Internet congestion control. In *Proceedings of the International Conference on Machine Learning.* 3050–3059.

[43] Peng Jin, Min Zhang, Jianwen Li, Li Han, and Xuejun Wen. 2021. Learning on abstract domains: A new approach for verifiable guarantee in reinforcement learning. *arXiv preprint arXiv:2106.06931* (2021).

[44] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *Proceedings of the International Conference on Computer Aided Verification.* 97–117.

[45] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, et al. 2019. The Marabou framework for verification and analysis of deep neural networks. In *Proceedings of the International Conference on Computer Aided Verification.* 443–452.

[46] Yafim Kazak, Clark Barrett, Guy Katz, and Michael Schapira. 2019. Verifying deep-RL-driven systems. In *Proceedings of the 2019 Workshop on Network Meets AI and ML.* 83–89.

[47] Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Bewley, and Amar Shah. 2019. Learning to drive in a day. In *Proceedings of the 2019 International Conference on Robotics and Automation (ICRA'19).* IEEE, Los Alamitos, CA, 8248–8254.

[48] Larissa Khodadadi, Behzad Samadi, and Hamid Khaloozadeh. 2014. Estimation of region of attraction for polynomial nonlinear systems: A numerical method. *ISA Transactions* 53, 1 (2014), 25–32.

[49] James C. King. 1976. Symbolic execution and program testing. *Communications of the ACM* 19, 7 (1976), 385–394.

[50] B. Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Pérez. 2021. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems* 23, 6 (2021), 4909–4926. https://doi.org/10.1109/TITS.2021.3054625

[51] Soonho Kong, Sicun Gao, Wei Chen, and Edmund Clarke. 2015. dReach: δ-Reachability analysis for hybrid systems. In *Proceedings of the International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems.* 200–205.

[52] Jernej Kos and Dawn Song. 2017. Delving into adversarial attacks on deep policies. *arXiv preprint arXiv:1705.06452* (2017).

[53] Scott Kuindersma, Robin Deits, Maurice Fallon, Andrés Valenzuela, Hongkai Dai, Frank Permenter, Twan Koolen, Pat Marion, and Russ Tedrake. 2016. Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot. *Autonomous Robots* 40, 3 (2016), 429–455.

[54] Aounon Kumar, Alexander Levine, and Soheil Feizi. 2022. Policy smoothing for provably robust reinforcement learning. In *Proceedings of the International Conference on Learning Representations.*

[55] Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of probabilistic real-time systems. In *Proceedings of the International Conference on Computer Aided Verification.* 585–591.

[56] Gerardo Lafferriere, George J. Pappas, and Sergio Yovine. 1999. A new class of decidable hybrid systems. In *Proceedings of the International Workshop on Hybrid Systems: Computation and Control.* 137–151.

[57] Jiwei Li, Will Monroe, Alan Ritter, Michel Galley, Jianfeng Gao, and Dan Jurafsky. 2016. Deep reinforcement learning for dialogue generation. *arXiv preprint arXiv:1606.01541* (2016).

[58] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[59] Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher Strong, Clark Barrett, and Mykel J. Kochenderfer. 2021. Algorithms for verifying deep neural networks. *Foundations and Trends® in Optimization* 4, 3-4 (2021), 244–404.

[60] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural adaptive video streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'17)*. 197–210.

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

[62] Ian M. Mitchell, Alexandre M. Bayen, and Claire J. Tomlin. 2005. A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games. *IEEE Transactions on Automatic Control* 50, 7 (2005), 947–957.

[63] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[64] Teodor Mihai Moldovan, Sergey Levine, Michael I Jordan, and Pieter Abbeel. 2015. Optimism-driven exploration for nonlinear systems. In *Proceedings of the 2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, Los Alamitos, CA, 3239–3246.

[65] Guido F. Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. 2014. On the number of linear regions of deep neural networks. *Advances in Neural Information Processing Systems* 27 (2014).

[66] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[67] Pierre El Mqirmi, Francesco Belardinelli, and Borja G. León. 2021. An abstraction-based method to check multi-agent deep reinforcement-learning behaviors. *arXiv preprint arXiv:2102.01434* (2021).

[68] Mark Niklas Müller, Gleb Makarchuk, Gagandeep Singh, Markus Püschel, and Martin Vechev. 2022. PRIMA: General and precise neural network certification via scalable convex hull approximations. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–33.

[69] Thanh Thi Nguyen, Ngoc Duy Nguyen, and Saeid Nahavandi. 2020. Deep reinforcement learning for multiagent systems: A review of challenges, solutions, and applications. *IEEE Transactions on Cybernetics* 50, 9 (2020), 3826–3839.

[70] Erik Nikko, Zoran Sjanic, and Fredrik Heintz. 2021. Towards verification and validation of reinforcement learning in safety-critical systems a position paper from the aerospace industry. In *Robust and Reliable Autonomy in the Wild, International Joint Conference on Artificial Intelligence*.

[71] Razvan Pascanu, Guido Montufar, and Yoshua Bengio. 2013. On the number of response regions of deep feed forward networks with piece-wise linear activations. *arXiv preprint arXiv:1312.6098* (2013).

[72] Stephen Prajna and Ali Jadbabaie. 2004. Safety verification of hybrid systems using barrier certificates. In *Proceedings of the International Workshop on Hybrid Systems: Computation and Control*. Springer, 477–492.

[73] Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming* (1st ed.). John Wiley & Sons, Inc., USA.

[74] Benjamin Recht. 2018. A tour of reinforcement learning: The view from continuous control. *arXiv preprint arXiv:1806.09460* (2018).

[75] Stephane Ross, Geoffrey Gordon, and Drew Bagnell. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 15)*, Geoffrey Gordon, David Dunson, and Miroslav Dudík (Eds.). PMLR, Fort Lauderdale, FL, USA, 627–635. https://proceedings.mlr.press/v15/ross11a.html.

[76] Vicenç Rubies-Royo, David Fridovich-Keil, Sylvia Herbert, and Claire J. Tomlin. 2019. A classification-based approach for approximate reachability. In *Proceedings of the 2019 International Conference on Robotics and Automation (ICRA)*. IEEE, Los Alamitos, CA, 7697–7704.

[77] Stefan Schaal. 1999. Is imitation learning the route to humanoid robots? *Trends in Cognitive Sciences* 3, 6 (1999), 233–242. https://doi.org/10.1016/S1364-6613(99)01327-3

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

[79] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. 2016. Safe, multi-agent, reinforcement learning for autonomous driving. *arXiv preprint arXiv:1610.03295* (2016).

[80] Claude E. Shannon. 1988. *Programming a Computer for Playing Chess*. Springer New York, New York, NY, 2–13. https://doi.org/10.1007/978-1-4757-1968-0_1

[81] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. 2018. Fast and effective robustness certification. *Advances in Neural Information Processing Systems* 31 (2018).

[82] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30.

[83] Armando Solar-Lezama. 2009. The sketching approach to program synthesis. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Springer, 4–13.

[84] Xiaowu Sun, Wael Fatnassi, Ulices Santa Cruz, and Yasser Shoukry. 2021. Provably safe model-based meta reinforcement learning: An abstraction-based approach. *arXiv preprint arXiv:2109.01255* (2021).

[85] Xiaowu Sun and Yasser Shoukry. 2021. Provably correct training of neural network controllers using reachability analysis. *arXiv preprint arXiv:2102.10806* (2021).

[86] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (2nd ed.). The MIT Press, Cambridge, MA, USA.

[87] Csaba Szepesvari. 2010. *Algorithms for Reinforcement Learning*. Morgan and Claypool Publishers.

[88] Lei Tai, Jingwei Zhang, Ming Liu, Joschka Boedecker, and Wolfram Burgard. 2016. A survey of deep network solutions for learning control in robotics: From reinforcement to imitation. *arXiv preprint arXiv:1612.07139* (2016).

[89] Russ Tedrake. 2022. *Underactuated Robotics*. http://underactuated.mit.edu.

[90] Russ Tedrake. 2022. Underactuated robotics: Algorithms for walking, running, swimming, flying, and manipulation. (2022).

[91] Chen Tessler, Yonathan Efroni, and Shie Mannor. 2019. Action robust reinforcement learning and applications in continuous control. In *Proceedings of the International Conference on Machine Learning*. PMLR, 6215–6224.

[92] Vincent Tjeng, Kai Xiao, and Russ Tedrake. 2017. Evaluating robustness of neural networks with mixed integer programming. *arXiv preprint arXiv:1711.07356* (2017).

[93] Hoang-Dung Tran, Feiyang Cai, Manzanas Lopez Diego, Patrick Musau, Taylor T. Johnson, and Xenofon Koutsoukos. 2019. Safety verification of cyber-physical systems with reinforcement learning control. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–22.

[94] Hoang-Dung Tran, Diago Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson. 2019. Star-based reachability analysis of deep neural networks. In *Proceedings of the International Symposium on Formal Methods*. Springer, 670–686.

[95] Hoang-Dung Tran, Patrick Musau, Diego Manzanas Lopez, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson. 2019. Parallelizable reachability analysis algorithms for feed-forward neural networks. In *Proceedings of the 2019 IEEE/ACM 7th International Conference on Formal Methods in Software Engineering (FormaliSE)*. IEEE, Los Alamitos, CA, 51–60.

[96] John Tromp and Gunnar Farnebäck. 2007. Combinatorics of Go. In *Computers and Games*, H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 84–99.

[97] Arjan J. Van Der Schaft and Johannes Maria Schumacher. 2000. *An Introduction to Hybrid Dynamical Systems*. Vol. 251. Springer London.

[98] Perry Van Wesel and Alwyn E. Goodloe. 2017. *Challenges in the Verification of Reinforcement Learning Algorithms*. Technical Report. NASA.

[99] Moshe Y. Vardi. 2009. Model checking as a reachability problem. In *Reachability Problems*, Olivier Bournez and Igor Potapov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 35–35.

[100] Abhinav Verma, Hoang Le, Yisong Yue, and Swarat Chaudhuri. 2019. Imitation-projected programmatic reinforcement learning. *Advances in Neural Information Processing Systems* 32 (2019).

[101] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. 2018. Programmatically interpretable reinforcement learning. In *Proceedings of the International Conference on Machine Learning*. PMLR, 5045–5054.

[102] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Efficient formal safety analysis of neural networks. *Advances in Neural Information Processing Systems* 31 (2018).

[103] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal security analysis of neural networks using symbolic intervals. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. 1599–1614.

[104] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J. Zico Kolter. 2021. Beta-crown: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification. *arXiv preprint arXiv:2103.06624* (2021).

[105] Fan Wu, Linyi Li, Zijian Huang, Yevgeniy Vorobeychik, Ding Zhao, and Bo Li. 2022. CROP: Certifying robust policies for reinforcement learning through functional smoothing. In *Proceedings of the International Conference on Learning Representations*.

[106] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. 2000. Torcs, the open racing car simulator. *Software Available at http://torcs. sourceforge. net* 4, 6 (2000), 2.

[107] Zikang Xiong and Suresh Jagannathan. 2021. Scalable synthesis of verified controllers in deep reinforcement learning. *arXiv preprint arXiv:2104.10219* (2021).

[108] Chao Yu, Jiming Liu, Shamim Nemati, and Guosheng Yin. 2021. Reinforcement learning in healthcare: A survey. *ACM Computing Surveys (CSUR)* 55, 1 (2021), 1–36.

[109] Huan Zhang, Hongge Chen, Chaowei Xiao, Bo Li, Mingyan Liu, Duane Boning, and Cho-Jui Hsieh. 2020. Robust deep reinforcement learning against adversarial perturbations on state observations. *Advances in Neural Information Processing Systems* 33 (2020), 21024–21037.

[110] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. 2019. An inductive synthesis framework for verifiable reinforcement learning. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 686–701.

Received 14 July 2022; revised 21 April 2023; accepted 2 May 2023