TRAINIFY: A CEGAR-Driven Training and Verification Framework for Safe Deep Reinforcement Learning

Peng Jin[1], Jiaxu Tian[1], Dapeng Zhi[1], Xuejun Wen[2], and Min Zhang[1,3(✉)]

[1] Shanghai Key Laboratory of Trustworthy Computing, ECNU, Shanghai, China
zhangmin@sei.ecnu.edu.cn
[2] Huawei International, Singapore, Singapore
[3] Shanghai Institute of Intelligent Science and Technology, Tongji University, Shanghai, China

Abstract. Deep Reinforcement Learning (DRL) has demonstrated its strength in developing intelligent systems. These systems shall be formally guaranteed to be trustworthy when applied to safety-critical domains, which is typically achieved by formal verification performed after training. This *train-then-verify* process has two limits: (i) trained systems are difficult to formally verify due to their continuous and infinite state space and inexplicable AI components (*i.e.*, deep neural networks), and (ii) the *ex post facto* detection of bugs increases both the time- and money-wise cost of training and deployment. In this paper, we propose a novel verification-in-the-loop training framework called TRAINIFY for developing safe DRL systems driven by counterexample-guided abstraction and refinement. Specifically, TRAINIFY trains a DRL system on a finite set of coarsely abstracted but efficiently verifiable state spaces. When verification fails, we refine the abstraction based on returned counterexamples and train again on the finer abstract states. The process is iterated until all predefined properties are verified against the trained system. We demonstrate the effectiveness of our framework on six classic control systems. The experimental results show that our framework yields more reliable DRL systems with provable guarantees without sacrificing system performance such as cumulative reward and robustness than conventional DRL approaches.

Keywords: Deep reinforcement learning · Model checking · CEGAR · ACTL

1 Introduction

Deep Reinforcement Learning (DRL) has shown its strength in developing intelligent systems for complex control tasks such as autonomous driving [37,40]. Verifiable safety and robustness guarantees are crucial to these safety-critical DRL systems before deploying [23,44]. A typical example is autonomous driving, which is arguably still a long way off due to safety concerns [21,39]. Recently,

S. Shoham and Y. Vizel (Eds.): CAV 2022, LNCS 13371, pp. 193–218, 2022.
https://doi.org/10.1007/978-3-031-13185-1_10

tremendous efforts have been made toward adapting existing and devising new formal methods for DRL systems in order to provide provable safety guarantees [18,25,45,46,51].

Formally verifying DRL systems is still a challenging problem. The challenge arises from DRL systems' three features. First, the state space of a DRL system is usually continuous and infinite [28]. Second, the behavior of a DRL system is non-linear and determined by high-order system dynamics [17]. Last but not least, the controllers, typically deep neural networks (DNN), are almost inexplicable because of their black-box development [20,52]. The three features make it unattainable to verify DRL systems using conventional formal methods, *i.e.*, modeling them as state transition systems and verifying temporal properties using dedicated decision procedures [4]. Most existing approaches have to simplify the problem by abstraction or over-approximation techniques and restrict to specific properties such as safety or reachability [46].

Another common problem with most existing formal verification approaches to DRL systems is that they are applied after the training is concluded. These *train-then-verify* approaches have two limitations. First, verification results may be inconclusive due to abstraction or overestimation. The non-linearity of both system dynamics and deep neural networks makes it difficult to control the overestimation in a reasonable range, resulting in false positives in verification results [50]. Second, the *ex post facto* detection of bugs increases both the time- and money-wise cost of training and deployment. No evidence shows that the iterative training and verification help improve system reliability, as tuning the parameters in neural networks may cause an unpredictable impact on the properties because of the inexplicability [24].

To address the challenges in training and verifying DRL systems, in this paper we propose a novel *verification-in-the-loop* framework for training safe and reliable DRL systems with verifiable guarantees. Provided that a set of properties are predefined for a target DRL system to develop, our framework trains the system and verifies it against the properties in every iteration. To overcome the verification challenges in DRL systems, for the first time, we propose a novel approach in our framework to train the systems on a finite set of *abstract states*, based on the observation that *approximate abstractions can still preserve near-optimal behavior* [1]. These states are the abstractions of the actual states. Training on the finite abstract states allows us to model the AI-embedded systems as finite-state transition systems. We can leverage classic model checking techniques to verify their more complicated temporal properties than safety and reachability.

As system performance may be affected by the abstraction granularity, we employ the idea of the counterexample-guided abstraction and refinement (CEGAR) [8] in model checking along the training process. We start with a coarsely abstracted but efficiently verifiable state space and train and verify DRL systems on the abstract state space. Once verification fails, we refine the abstract state space based on the returned counterexamples and retrain the system on the finer-grained refined state space. The process is repeated until all the

properties are verified successfully. We, therefore, call the training and verification framework *CEGAR-driven*, by which we can reach an appropriate abstraction granularity that guarantees both system performance and verification scalability.

Our verification-in-the-loop training framework has four advantages compared with conventional DRL training and verification approaches. Firstly, our approach produces correct-by-construction DRL systems that are verifiably safe with respect to user-defined safety requirements. Secondly, more complicated properties such as safety and liveness can be verified thanks to the dedicated training approach on abstracted state space. Another advantage of the training approach is that it is orthogonal to state-of-the-art DRL algorithms such as Deep Q-Network (DQN) [34] and Deep Deterministic Policy Gradient (DDPG) [32]. Thirdly, our approach provides a flexible mechanism for fine-tuning an appropriate abstraction granularity to balance system performance and verification scalability. Lastly, training on abstract states renders DRL systems to be more robust against adversarial and environmental perturbations because small perturbation to an actual state may not alter the decision of the neural network on the same abstract state.

We implement a prototype tool called TRAINIFY (abbreviated for Train and Verify, available at https://github.com/aptx4869tjx/RL_verification). We perform extensive experiments on six classic control tasks in public benchmarks to evaluate the effectiveness of our framework. For each task, we train two DRL systems under the same settings in our approach and corresponding conventional DRL algorithm, respectively. We compare the two systems in terms of the properties that they shall satisfy and the performance in terms of cumulative reward and robustness. Experimental results show that the systems trained in our approach are more efficient to verify and more reliable than those trained in conventional methods; moreover, their performance is competitive and higher.

In summary, this paper makes the following three major contributions:

1. A novel verification-in-the-loop training framework for developing verifiable and reliable DRL systems with correct-by-construction guarantees.
2. A CEGAR-driven approach for fine-tuning abstraction granularity during training to reach a balance between system performance and verification scalability.
3. A resulting prototype tool called TRAINIFY for training and verifying DRL systems and a thorough evaluation of the proposed approach on public benchmarks.

Paper Organization. Section 2 briefly introduces deep reinforcement learning. Section 3 presents the model-checking problem of DRL systems. Section 4 presents our training and verification framework. Section 5 shows six case studies and experimental results. Section 6 mentions some related work, and Sect. 7 concludes the paper.

2 Deep Reinforcement Learning (DRL)

DRL is a technique for learning optimal control policies using deep neural networks according to evaluative feedback [31]. An agent in a DRL system interacts with the environment and records its state s_t at each time step t. It feeds s_t into a deep neural network to compute an action a_t and transitions to the next state s_{t+1} according to a_t and the system dynamics. The system dynamics describe the non-linear behavior of the agent over time. The agent receives a scalar reward according to reward functions. Some algorithms estimate the distance between the action determined by the network and the expected action in the same state. Then, it updates the parameters in the network according to the estimated distance to maximize the cumulative reward.

A Running Example. Figure 1 shows a classic DRL task of learning a control policy to drive a car to the right hilltop. The car is initially positioned on a track between two mountains. The track is one-dimensional, and thus the car's position is represented as a real number. Velocity is another dimension in the car's state and is represented as a real number too. Thus, the car's state is a pair (p, v) of position p and velocity v. An action a is a real number representing the force imposed on the car. The action is computed by a neural network on both p and v.

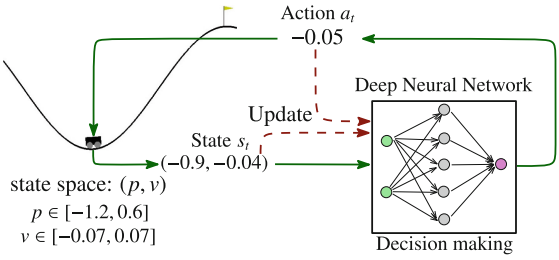


Fig. 1. A DRL example of mountain car system.

The sign of a means the direction of the force, *i.e.*, positive for the right and negative for the left, respectively. Given a state $s_t = (p_t, v_t)$ and an action a_t at time step t, the system transitions to the next step $s_{t+1} = (p_{t+1}, v_{t+1})$ following the given dynamics:

$$p_{t+1} = p_t + v_t \Delta_t, \tag{1}$$

$$v_{t+1} = v_t + (a_t - m_c \times g \times cos(3p_t))\Delta_t, \tag{2}$$

where m_c denotes the car's mass, g denotes the gravity, and Δ_t is the unit interval between two consecutive steps. In DRL, time is usually discretized to facilitate implementation. The car is assumed to move in uniform motion during a unit interval.

Reward Setting. The reward function R maps state s_t, action a_t and state s_{t+1} to a real number, which represents the rewarded value by applying a_t to s_t to transition to s_{t+1}. The purpose of R is to guide the agent to achieve the preset goals by making cumulative reward as great as possible. The definition of R is based on prior knowledge or expert experience before training.

In the Mountain Car example, the controller receives the reward which is defined as $R(\langle p_t, v_t \rangle, a_t, \langle p_{t+1}, v_{t+1} \rangle) = -1.0$ at each time step when $p_{t+1} < 0.45$.

The reward is a negative constant because the goal in this example is to force the car to reach the right hilltop ($p = 0.45$) as quickly as possible. If the corresponding cumulative reward value is larger than another when the car reaches the destination, it means that the car takes fewer steps. A reward function can be a more complex formula than a constant when the reward strategy is related to states and actions.

Training. The essence of DRL training is to update parameters in neural networks so that the networks can compute optimal actions for input states. A deep neural network is a directed graph comprised of an input layer, multiple hidden layers, and an output layer, as shown in Fig. 2. Each layer contains several nodes called *neurons*. They are connected to the neurons on the following layer. Each edge has a weight. The values passed on the edge are multiplied by the weight. A neuron on hidden layers takes the sum of all the incoming values, adds a bias, and feeds the result to its activation function σ. The output of σ is passed to the neurons on the following layer. There are several commonly used activation functions, e.g., ReLU ($\sigma(x) = max(x, 0)$), Sigmoid ($\sigma(x) = \frac{1}{1+e^{-x}}$) and Tanh ($\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$), etc. In DRL, the inputs to a neural network are system states. The outputs are (probably continuous) actions that shall be performed to the present state.

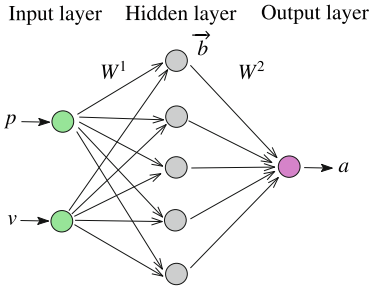


Fig. 2. A simple neural network.

During training, agents continuously interact with the environment to obtain trajectories. A trajectory is a 4-tuple, consisting of a state s, the action a on s, the reward of executing a on s, and the successor state after the execution. A predefined loss function uses the collected trajectories to estimate an action value and compute the distance between the estimated value and the one computed by the neural network for the same state. Guided by the distance, the parameters in the network are updated using gradient descent algorithms [12]. The process is repeated until the system reaches a predefined maximal iteration limit or a preset cumulative reward threshold.

Algorithm 1: Training for the Mountain Car Task using DQN

```
for episode = 1, ..., M do
    Initialize s_0 = (p_0, v_0)
    for t = 0, ..., T do
        a_t ← N(p_t, v_t);            /* To determine a_t based on s_t = (p_t, v_t) and N */
        (s_{t+1}, −1.0) ← system(s_t, a_t);   /* To execute a_t and transition to the next
          state s_{t+1} */
        P ← L(N, ⟨s_i, a_i, −1.0, s_{i+1}⟩, ..., ⟨s_j, a_j, −1.0, s_{j+1}⟩);   /* To compute the
          distance */
        N ← update(N, P);             /* To update parameters in N based on P */
```

There are several well-established training algorithms, such as Deep Q-Network (DQN) [35] and Deep Deterministic Policy Gradient (DDPG) [32]. Algorithm 1 depicts a high-level process of training the mountain car using DQN. We call the process of training the car to move from the initial position to the destination an *episode*. For each episode, the initial state is firstly determined (Line 2). Then, the controller determines the action to be adopted based on the current state s_t and the neural network $\mathcal{N}$ (Line 4). After performing the action, the controller receives a reward value (-1.0 in this case) and transitions to the next state based on the system dynamics (Line 5). A loss is estimated by calling the loss function $\mathcal{L}$ with partially sampled trajectories. The loss is represented by $\mathcal{P}$ (Line 6) used to update the parameters of the network $\mathcal{N}$ (Line 7). We omit the details of $\mathcal{L}$, as it is not the emphasis of our paper.

The Target DRL Systems in this Work. The types of DRL systems are diverse from different perspectives, such as the availability of system dynamics [17] and the determinism of actions. In this work, we assume system dynamics is prior knowledge for training, and the actions are deterministic. That is, a unique action is determined to take on the present state, and its successor state is also uniquely determined by system dynamics.

3 Model Checking of DRL Systems

3.1 The Model Checking Problem

A trained deterministic DRL system can be represented as a tuple $M = (S, A, f, \pi, S^0, L)$, where S is the state space which is usually infinite, $S^0 \subseteq S$ is the initial state space, A is a set of actions, $f : S \times A \to S$ is the system dynamics, $\pi : S \to A$ is a policy function, and $L : S \to 2^{AP}$ is a state labeling function. In this work, we use π to denote the policy that is implemented by the trained deep neural network in the system.

The model M of a DRL system is essentially a Kripke structure [10], which is a 4-tuple (S, R, S^0, L). Given two arbitrary states s, s' in S, there is a transition from s to s', denoted by $(s, s') \in R$, if and only if there is an action a in A such that $a = \pi(s)$ and $s' = f(s, a)$. Given that a property is formalized by a formula Φ in some logic, the model checking problem of the system is to decide whether M satisfies Φ, denoted by $M \models \Phi$.

In this work, we formulate properties in ACTL [4], a segment of CTL where only universal path quantifiers are allowed and negation is restricted to atomic propositions [14,15]. ACTL consists of state formula Φ and path formula φ in the following syntax:

$$\begin{aligned} \Phi &::= \mathit{true} \mid \mathit{false} \mid a \mid \neg a \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid A\,\varphi, \\ \varphi &::= X\,\Phi \mid \Phi_1\,U\,\Phi_2 \mid \Phi_1\,R\,\Phi_2. \end{aligned}$$

The temporal operators fall into two main categories, *i.e.*, quantifiers over paths and path-specific quantifiers. In ACTL, only the universal path quantifier A is considered. Path-specific quantifiers refer to X, U and R.

- $A\ \varphi$: Path formula φ has to hold on all paths starting from the current state.
- $X\ \Phi$: State formula Φ has to hold at the next state.
- $\Phi_1\ U\ \Phi_2$: State formula Φ_1 has to hold at least until state formula Φ_2.
- $\Phi_1\ R\ \Phi_2$: Formula Φ_2 has to hold until and including a point where Φ_1 first becomes true. If Φ_1 never becomes true, Φ_2 must hold forever.

Using the above basic temporal operators, we can define another two important path-specific quantifiers G (*globally*) and F (*finally*) with $G\ \Phi = \mathit{false}\ R\ \Phi$ and $F\ \Phi = \mathit{true}\ U\ \Phi$. Intuitively, $G\ \Phi$ means that Φ has to hold on the entire subsequent path, and $F\ \Phi$ means that Φ eventually has to hold (somewhere on the subsequent path).

We choose ACTL to formulate system properties or requirements in our framework for two main reasons. Firstly, in our framework, we rely on refinement to the abstract states where system properties are violated. Such states can be obtained as counterexamples returned by model checkers when the system properties defined in ACTL are verified not valid by model checking. Secondly, the verification results of ACTL formulas can be preserved by property-based abstraction [9,11]. Such preservation is vital to ensure the correctness of our verification results because the abstraction is necessary for our framework to guarantee the scalability of the verification algorithm.

3.2 Challenges in Model Checking DRL Systems

Unlike the model checking problems for finite-state systems, model checking $M \models \Phi$ for DRL systems is particularly challenging. The challenge arises from the three features of DRL systems, *i.e.*, (i) the infinity and continuity of state space S, (ii) the non-linearity of system dynamics f, and (iii) the inexplicability of the policy π that is encoded as deep neural networks. Usually, the state space of DRL systems is continuous and infinite, and behaviors are non-linear due to high-order system dynamics. Even worse, the actions of states are determined by inexplicable deep neural networks, which means that the transitions between states cannot be defined as straightforwardly as those of traditional software systems.

To build a model M for a DRL system, we have to compute the successor of each state s by applying the neural network π on s to compute the action a and then performing a to s according to the system's dynamics f. Specifically, the successor of s can be represented as $f(s, \pi(s))$. The non-linearity of both f and π and the infinity of S makes the verification problem difficult. Most existing approaches rely on the over-approximation of f and π to simplify the problem [16,25,29,46]. However, over-approximation inevitably introduces over-estimation and restricts to only safety properties and reachability analysis in bounded steps.

4 The CEGAR-Driven DRL Approach

4.1 The Framework

Figure 3 shows the overview of our framework. It consists of three parts, *i.e.*, training, verification and refinement. In the training part, a DRL system is trained on a finite set of abstract states. An actual state is first mapped to its corresponding abstract state, then fed into the neural network to compute a corresponding action. The action is applied to the actual state to drive the system to transition to the next state. The reward is accumulated according to a predefined reward function, and the neural network is updated in the same way as conventional DRL algorithms. In the verification part, we build a Kripke structure on the finite abstract state space based on the trained neural network. Then, we verify the desired properties that are predefined in ACTL formulas Φ. If all the properties are verified valid, we stop training, and a DRL system is developed. If some property is verified not valid, we move to the refinement part. When verification fails, counterexamples are returned. They are the abstract states where the property is violated. We refine these states by subdividing them into fine-grained sub-states and substitute those *bad* states. We resume to train the system on the refined abstract state space and repeat the whole process.

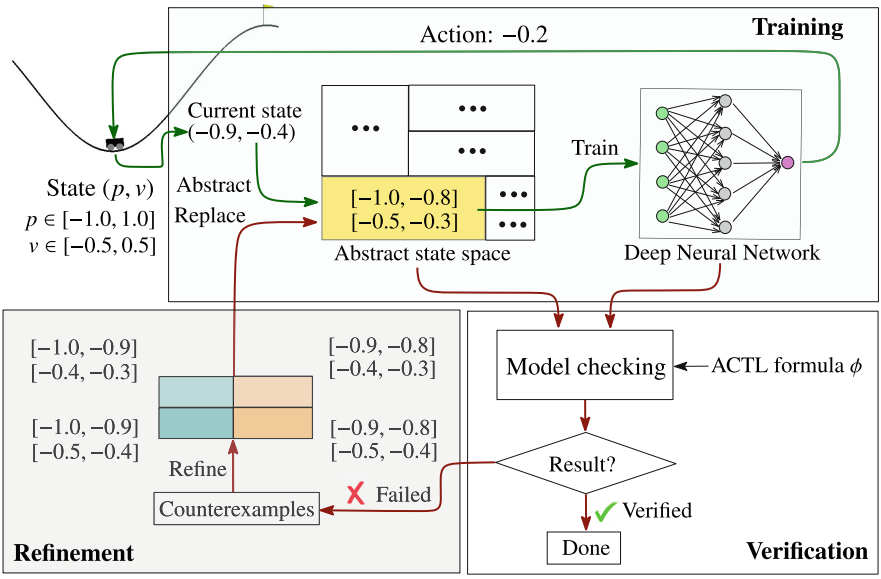
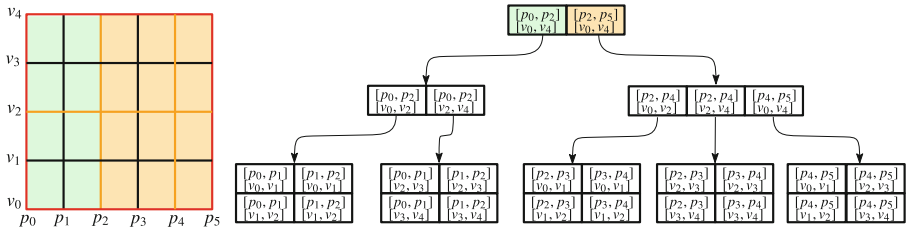


Fig. 3. The training, verification and refinement framework for developing DRL systems.

The integration of training, verification and refinement seamlessly constitutes a *verification-in-the-loop* DRL approach, driven by the counterexample-guided abstraction and refinement. We start with a coarse abstraction. After every training episode, we model check the system against all the predefined properties. If all the properties are verified, we stop training and obtain a verified system. Otherwise, counterexamples are returned. The abstract state space is refined for further training. After several iterations, a DRL system is trained with all the predefined properties rigorously verified.

4.2 Training on Abstract States

DRL is a process of learning optimal actions on all system states for specific objectives. A trained model partitions the state space into a family of sets such that the same action is taken in the states from a set [38]. Continuous state spaces can be adaptively discretized into finite ones for learning without affecting learning performance [41,42]. Motivated by this observation, we discretize a continuous state space into a finite set of fragments. We call each fragment an abstract state and train the DRL system by feeding abstract states into the deep neural network for decision making.



(a) An abstract state space. (b) An R-tree of the abstract state space.

Fig. 4. An example of encoding an abstract state space into an R-tree.

System State Abstraction. Given an n-dimension DRL system, a concrete system state s is represented as a vector of n real numbers. Each number has a physical meaning about the system, such as speed and position in the running example. Let L_i and U_i be the lower and upper bounds for the i-th dimension value of S. Then, the state space S of the control system is $\Pi_{i=1}^{n}[L_i, U_i]$.

Initially, we use interval boxes to discretize S. An interval box I is a vector of n intervals, denoted by $(I_1, I_2, \ldots, I_n)$. Each interval $I_i (1 \leq i \leq n)$ represents all the system states, denoted by S_{I_i}, where a state s belongs to S_{I_i} if and only if the i-th value in s is in I_i. An interval box I represents the intersection of all the sets $S_{I_i} (i = 1, \ldots, n)$.

Let $d_i \in \mathbb{R}$ $(0 < d_i \leq U_i - L_i)$ be the diameter by which we subdivide evenly the interval $[L_i, U_i]$ in each dimension i into $(U_i - L_i)/d_i$ unit intervals, and $\mathcal{I}_i = [L_i, U_i]/d_i$ denote the set of all the unit intervals. Then, we obtain the abstract state space $\mathbf{S} = \mathcal{I}_1 \times \ldots \times \mathcal{I}_n$, which is an abstraction of the infinite continuous state space S. We call the vector $(d_1, d_2, \ldots, d_n)$ of the n diameters *abstraction granularity* and denote it by δ.

Given a continuous state space S and its corresponding abstract state space $\mathbf{S}$, we call the mapping function from the states in S to the corresponding abstract states in $\mathbf{S}$ a *transformer* $\mathcal{A} : S \rightarrow \mathbf{S}$. The transformer can be encoded as an R-tree, a tree-like data structure devised for efficiently indexing multi-dimensional objects [22]. Figure 4 depicts an example of building an R-tree to index an abstract state space of the continuous space $[v_0, v_4] \times [p_0, p_5]$. A rectangle on a leaf node represents an abstract state, and the one on a non-leaf node represents the minimum bounding rectangle enclosing all the rectangles on its child nodes. There can be multiple rectangles on a single node. R-tree supports intersection search, *i.e.*, searching for the abstract states that intersect with the interval we are querying. Given a concrete state, an R-tree can quickly return its corresponding abstract state. Note that in Fig. 4, we assume state space is discretized evenly for clarity. During training, the size of abstract states becomes diverse after iterative refinement, and the R-tree should be updated correspondingly.

The Training Algorithms. The algorithms for training on abstract states can be achieved by extending existing DRL algorithms such as DQN and DDPG. The extension can be easily achieved by adapting the neural networks and loss functions in DRL systems so that they can admit abstract states as inputs.

Algorithm 2: Abstraction-Based DRL Training

```
1 for episode = 1, ..., M do
2     A ← discretize(S, δ);        /* To discretize S by abstraction granularity δ */
3     Initialize s_0;
4     for t = 0, ..., T do
5         s_t ← A(s_t) ;                          /* To get abstract state of s_t */
6         a_t ← N'(s_t);            /* To determine action a_t based on s_t and N' */
7         (s_{t+1}, r_t) ← system(s_t, a_t) ;     /* To execute a_t on s_t and transition to
              s_{t+1} with reward r_t */
8         P = Loss(N', ⟨s_i, a_i, r_i, s_{i+1}⟩,...,⟨s_j, a_j, r_j, s_{j+1}⟩); /* To get loss due to
              a_t */
9         N' ← update(N', P);       /* To update parameters in N' based on P */
```

For neural networks, we only need to modify the input layer by doubling the number of neurons on the input layer, denoted by $\mathcal{N}'$. Given an n-dimension system, we declare $2n$ neurons. Each pair of neurons read the lower and upper bounds of an interval in an abstract state, respectively. This dedicated structure guarantees that a trained network can produce the same action for all the states that correspond to the same abstract state.

Figure 5 shows an example of adapting the network in the Mountain Car for training it on abstract states. For traditional DRL algorithms, two input neurons are needed in the neural network to take p and v as inputs, respectively. To train on abstract states, four input neurons are needed to take the lower and upper bounds of the position and velocity intervals in abstract states. For instance, let the interval box (I_p, I_v) be the abstract state of (p, v). Then, the lower bounds $\underline{I_p}, \underline{I_v}$ and the upper bounds $\overline{I_p}, \overline{I_v}$ of p, v are input to the four neurons, respectively. Apparently, this adaption guarantees that the neural network always produces the same action on the states that are transformed into the same abstract state.

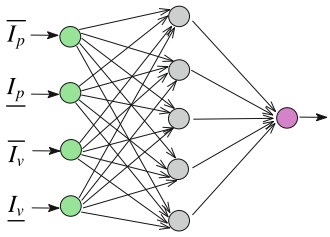


Fig. 5. Adapting neural networks for abstract states.

We consider incorporating these two steps to extend Algorithm 1 as an illustrative example. Algorithm 2 depicts the main workflow where the differences are highlighted. The main difference from the traditional training process lies in line 6. Given a concrete state $s = (s_1, \ldots, s_n)$, $\mathcal{A}$ will return the abstract state $\mathbf{s} = ([l_1, u_1], \ldots, [l_n, u_n])$ such that $l_i \leq s_i \leq u_i$ with $i = 1, \ldots, n$, which is also the result fed into neural network. Although the dimension of input states increases, the form of corresponding output actions does not change. Therefore, the loss function can naturally adapt to changes in input states.

4.3 Model Checking Trained DRL Systems

A DRL system can be naturally verified using abstract model checking [26]. The actual states of the system are first abstracted in the same way used in training, and then the transitions between abstract states are determined by the corresponding action and dynamics. ACTL formulas are then model checked on the abstract state transition system.

Building Kripke Structure. During the training phase, the actual state space has already been abstracted into a finite set $\mathbf{S}$ of abstract states. Therefore, the main task for abstract model checking is to build a Kripke structure by defining the transition relation on $\mathbf{S}$.

Algorithm 3 depicts the process of building a Kripke structure $\mathcal{K}$ for a trained DRL system. Firstly, $\mathcal{K}$ is initialized on set $\mathbf{S}$ with R being empty. Starting from an initial abstract state $\mathbf{s}^0$, we compute its successors and define the transitions from $\mathbf{s}^0$ to them. We repeat the process until all reachable states are traversed.

Algorithm 3: Building Kripke Structure

Input: Initial state $\mathbf{s}^0$, state space $\mathbf{S}$, system dynamics f, neural network $\mathcal{N}'$
Output: A Kripke Structure $\mathcal{K}$

```
1  K = Initialize_Kripke_Structure()
2  Queue ← {s^0}
3  while Queue is not empty do
4      Fetch s from Queue
5      for i = 1, ..., n do
6          [l_i, u_i] ← g(f(s, N'(s)), i)
7      {s^1, ..., s^m} :=
         h([l_1, u_1], ..., [l_n, u_n], S)
8      for j = 1, ..., m do
9          K.add_edge(s → s^j)
10         if s^j is not traversed then
11             Push s^j into Queue
12 return K
```

Given an abstract state $\mathbf{s}$, we compute its abstract successor states by applying the corresponding action a and the dynamics to $\mathbf{s}$. Because the system is trained on abstract states, all the actual states in $\mathbf{s}$ have the same action, *i.e.*, $a = \mathcal{N}'(\mathbf{s})$. Let $f^*(\mathbf{s}, a) = \{f(s, a) | s \in \mathbf{s}\}$ be the set of all the successors of the actual states in $\mathbf{s}$. Due to the non-linearity of f and the infinity of $\mathbf{s}$, we over-approximate the set $f^*(\mathbf{s}, a) = \{f(s, a) | s \in \mathbf{s}\}$ as an interval box. As shown in Fig. 6, the dashed box is an over-approximation of $f^*(\mathbf{s}, a)$. The over-approximation may overlap one or more abstract states, e.g., $\mathbf{s}^1, \ldots, \mathbf{s}^4$ in the example. All the overlapped abstract states are successors of $\mathbf{s}$. In Algorithm 3, function g calculates the interval box and function h determines the overlapped abstract states. Note that the shapes of abstract states may be different because they are refined during training, which is to be detailed in Sect. 4.4.

We use an interval to approximate the i-th dimension's values in all the successor states. Then, all the successor states are approximated as a vector of n intervals. We can compute the upper and lower bounds for each i by solving the following two optimization problems, respectively:

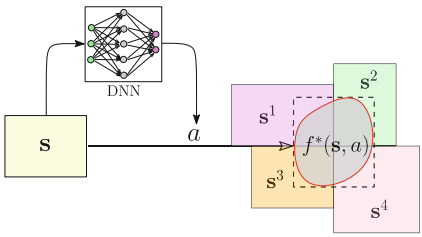


Fig. 6. Transitions between abstract states

$$\arg\max_{s \in \mathbf{s}} \quad v_i \cdot f(s, \mathcal{N}'(\mathbf{s}))$$

$$\arg\min_{s \in \mathbf{s}} \quad v_i \cdot f(s, \mathcal{N}'(\mathbf{s}))$$

where, v_i is a one-hot vector with the i-th element being 1. Because all the states in $\mathbf{s}$ have the same action according to the network, $\mathcal{N}'(\mathbf{s})$ in the above optimization problems can be substituted for a constant, *i.e.*, the action taken

by the system on all the states in $\mathbf{s}$. The substitution significantly simplifies the optimization problems; no information of the networks is needed in the simplified problems. The simplified problems can be efficiently solved using off-the-shelf scientific computing tools such as SciPy [48].

We consider an example in the mountain car system. We assume that the current abstract state $\mathbf{s}$ is $([0, 0.2], [0, 0.02])$ and the adopted action is 0.001, which says that the controller accelerates to the right for all states in $\mathbf{s}$. Based on the dynamics defined by Eq. 1, we can compute the upper bounds of both position and velocity in the next step by solving the following two optimization problems:

$$\underset{p_t \in [0,0.2], v_t \in [0,0.02]}{\arg\max} \quad p_t + v_t \tag{p_{t+1}}$$

$$\underset{p_t \in [0,0.2], v_t \in [0,0.02]}{\arg\max} \quad v_t + 0.001 - 0.0025\cos(3p_t) \tag{v_{t+1}}$$

The lower bounds of p_{t+1} and v_{t+1} are calculated similarly. Then, we obtain an abstract state $\mathbf{s}' = ([0, 0.22], [-0.0035, 0.0165])$, which is an overestimated set of all the actual successors of the states in $\mathbf{s}$. There is a transition from $\mathbf{s}$ to any abstract state $\mathbf{s}'' = ([\underline{p}, \overline{p}], [\underline{v}, \overline{v}])$ in $\mathbf{S}$, if $\mathbf{s}'$ and $\mathbf{s}''$ overlap, *i.e.*, $(0 < \underline{p} < 0.22 \vee 0 < \overline{p} < 0.22) \wedge (-0.0035 < \underline{v} < 0.0165 \vee -0.0035 < \overline{v} < 0.0165)$ is true. Note that the transition from $\mathbf{s}$ to $\mathbf{s}'$ includes all the transitions between the actual states in $\mathbf{s}$ and $\mathbf{s}'$, respectively. It may also include those that do not actually exist due to the overestimation.

There are other approaches for over-approximating the set $f^*(\mathbf{s}, a)$, such as template polyhedrons like rectangle and octagon [2]. Note that there is always a trade-off between the tightness of the polyhedral and the efficiency of computing it. For instance, an octagon can approximate the set more tightly than a rectangle. However, it costs double effort to compute the borders. The tighter an over-approximation is, the more accurate the set of computed successors is, but the more time it costs to compute the approximation.

Property-Based Abstraction. For those high-dimensional DRL systems, the abstract state space may be still too huge to model check directly when the abstraction granularity becomes small after refinement. To improve the model checking scalability, we further abstract the constructed Kripke structure based on the ACTL formula Φ to be model checked using the abstraction approach in the work [9].

Definition 1 (State Abstraction). *Given an abstract state space* $\mathbf{S} = \mathcal{I}_1 \times \ldots, \times \mathcal{I}_n$ *and an ACTL formula* Φ*, let* D_Φ *be the set of dimensions that occur in* Φ *and* $\widehat{\mathbf{S}} = \Pi_{\mathsf{d} \in D_\Phi} \mathcal{I}_\mathsf{d}$*. Function* $\alpha_\Phi : \mathbf{S} \to \widehat{\mathbf{S}}$ *is an abstract transformer such that for every* $\mathbf{s} \in \mathbf{S}$ *and* $\widehat{\mathbf{s}} \in \widehat{\mathbf{S}}$*,* $\widehat{\mathbf{s}} = \alpha_\Phi(\mathbf{s})$ *if and only if* $\mathbf{s}[\mathsf{d}] = \widehat{\mathbf{s}}[\mathsf{d}]$ *for all* $\mathsf{d} \in D_\Phi$*.*

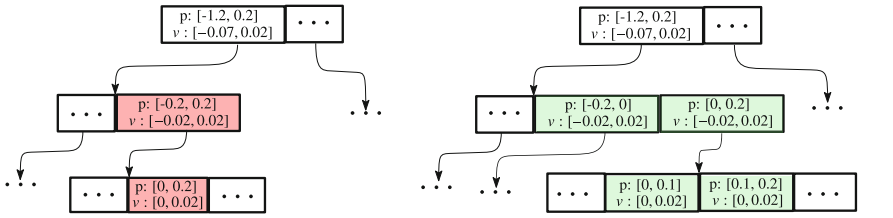
Given a Kripke structure $\mathcal{K} = (\mathbf{S}, R, \mathbf{S}^0, L)$ and an ACTL formula Φ, let $\alpha_\Phi : \mathbf{S} \to \widehat{\mathbf{S}}$ be the abstract transformer, and $\widehat{AP} \subseteq AP$ be all the atomic propositions in Φ. We can construct the following abstract Kripke structure $\widehat{\mathcal{K}} = (\widehat{\mathbf{S}}, \widehat{R}, \widehat{\mathbf{S}}^0, \widehat{L})$ based on α_Φ, where:

- $\widehat{\mathbf{S}} = \Pi_{\mathsf{d} \in D_\Phi} \mathcal{I}_\mathsf{d}$;
- $\widehat{R} = \{(\alpha_\Phi(\mathbf{s}), \alpha_\Phi(\mathbf{s}')) | \mathbf{s}, \mathbf{s}' \in \mathbf{S}.(\mathbf{s}, \mathbf{s}') \in R\}$;
- $\widehat{\mathbf{S}}^0 = \{\alpha_\Phi(\mathbf{s}) \mid \mathbf{s} \in \mathbf{S}^0\}$;
- $\widehat{L} : \widehat{\mathbf{S}} \to 2^{\widehat{AP}}$ such that $\widehat{L}(\widehat{\mathbf{s}}) = L(\mathbf{s}) \cap \widehat{AP}$ where $\mathbf{s} \in \mathbf{S}$ and $\widehat{\mathbf{s}} = \alpha_\Phi(\mathbf{s})$.

We call $\widehat{\mathcal{K}}$ a simulation of $\mathcal{K}$ with respect to Φ. An important property of the simulation is that the property represented by Φ is preserved by the abstract model $\widehat{\mathcal{K}}$.

Theorem 1 (Soundness). *Let $\widehat{\mathcal{K}}$ be a simulation of $\mathcal{K}$ with respect to an ACTL formula Φ, $\widehat{\mathcal{K}} \models \Phi$ implies $\mathcal{K} \models \Phi$.*

The proof of Theorem 1 is straightforward. We omit the proof due to space limit. According to the theorem, we can conclude that $\mathcal{K} \models \Phi$ holds whenever we find a simulation $\widehat{\mathcal{K}}$ of $\mathcal{K}$ and model check that $\widehat{\mathcal{K}} \models \Phi$ holds.



(a) Counterexamples on an R-tree. (b) The R-tree after refinement on the counterexample.

Fig. 7. An example of refinements on abstract states where properties are violated.

4.4 Counterexample-Guided Refinement

If a formula Φ is verified not true, our algorithm returns corresponding counterexamples. A counterexample is an abstract state where Φ is violated. We refine the abstract state into finer ones and substitute them in the abstract state space for further training.

A naïve refinement approach subdivides each dimension of states into two intervals. Assuming that a property is violated on an abstract state $\mathbf{s} = ([l_0, u_0], \ldots, [l_n, u_n])$, we can simply divide each dimension evenly into two intervals $([l_i, (l_i + u_i)/2], [(l_i + u_i)/2, u_i])$, and obtain 2^n finer abstract states. Apparently, the refinement may lead to state space explosion, particularly for high-dimensional systems.

In our approach, we only refine the states on the dimensions that are used to define the properties being verified to avoid state explosion. Considering the mountain car example, we assume that the formula is $AF[p \geq 0.45]$, saying that the car will eventually reach the hilltop where $p = 0.45$. Suppose that the property fails and counterexamples are returned. We assume $\mathbf{s} = ([0, 0.2], [0, 0.02])$ is the state

where the property is violated, as shown in Fig. 7 (a). We bisect the state into two fine-grained sub-states, $\mathbf{s}^1 = ([0, 0.1], [0, 0.02])$ and $\mathbf{s}^2 = ([0.1, 0.2], [0, 0.02])$. Then, we substitute the two fine-grained states for $\mathbf{s}$ on the R-tree for further training. Figure 7 (b) shows the new R-tree after the substitution.

It is worth mentioning that counterexamples may be false positives. Abstract states may include the actual states that are unreachable in the trained system because of the approximation of system dynamics. Unfortunately, it is difficult to check which states are actually unreachable because we need to know their corresponding initial state to check the reachability of these bad states. However, the corresponding initial state is *enclosed* in an abstract state and cannot be identified due to the abstraction. In our approach, we perform refinement without checking whether the counterexamples are real or not. After refinement, the abstract states become finer-grained. Counterexamples can be discarded by training and verifying on these finer-grained abstract states. The price of such extra refinements is that more iterations of training and verification are conducted, but the benefit is that the performance of the trained systems is better.

5 Implementation and Evaluation

5.1 Implementation

We implement our framework into a prototype toolkit called TRAINIFY in Python. In the toolkit, we leverage the open-source library *pyModelChecking* [6] as the back-end model checker and the scientific computing tool SciPy [48] as an optimization solver.

5.2 Benchmarks and Experimental Settings

We evaluate the effectiveness of our approach on a wide range of classic control tasks from public benchmarks. For each control task, we train two DRL systems using our approach and the corresponding conventional DRL approach, respectively. We compare the two trained systems in terms of their reliability, verifiability and system performance.

Benchmarks. We choose six classic control problems. Three of them are from the DRL training platform Gym [5], including Mountain Car, Pendulum and Cartpole. The other three, *i.e.*, B1, B2 and Tora, are the problems that are widely used for evaluation by state-of-the-art tools [19,25,27,28].

1. **Mountain Car (MC).** The running example in Sect. 2.
2. **Pendulum (PD).** A pendulum that can rotate around an endpoint is delineated. Starting from a random position, the pendulum shall swing up and stay upright.
3. **CartPole (CP).** A pole is attached by an un-actuated joint to a cart. The goal of training is to learn a controller that prevents the pole from falling over by applying a force of $+1$ or -1 to the cart.

4. **B1** and **B2**. Two classic nonlinear systems, where agents in both systems aim to arrive at the destination region from the preset initial state space [19].
5. **Tora.** A cart is attached to a wall with a spring. It is free to move on a frictionless surface. Inside the cart, there is an arm free to rotate about an axis. The controller's goal is to stabilize the system at the equilibrium state where all the system variables are equal to 0.

Training Configurations and Evaluation Metrics. We adopt the same system configurations and training parameters for each task, including neural network architecture, system dynamics, time interval, DRL algorithms and the number of training episodes.

We choose three metrics, including the satisfaction of predefined properties, cumulative reward and robustness, to evaluate and compare the reliability, verifiability and performance of the DRL systems trained in our approach and those trained in the conventional DRL approach for the same task. The first metric is about reliability and verifiability. The other two are about performance. The cumulative reward is an important figure to evaluate a trained system's performance because maximizing the cumulative reward is the objective of learning. Robustness is another essential criterion for DRL systems because the systems are expected to be robust against perturbations from both the environment and adversarial attacks. Note that we classify robustness into performance category instead of reliability because we restrict the reliability of DRL systems to the safety and functional requirements.

Experimental Settings. All experiments are conducted on a workstation running Ubuntu 18.04 with a 32-core AMD Ryzen Threadripper CPU @ 3.7 GHz and 128 GB RAM.

5.3 Reliability and Verifiability Comparison

We first evaluate the reliability and verifiability of the DRL systems trained in our approach and conventional approach, respectively. For each task, we predefined system properties according to their safety and functional requirements. The functional requirement is usually the objective of control tasks. For instance, the controller's objective to train in the mountain car example is to drive the car to the hilltop. We define an atomic proposition $p > 0.45$ to indicate that the car reaches the hilltop. Then, we can define an ACTL formula $\Phi_1 = AF(p > 0.45)$ to represent the liveness property. Safety requirements in DRL systems usually specify important parameters of the systems that must always be kept in safe ranges. For instance, a safety requirement in the mountain car example is that the car's velocity must be greater than 0.02 when the car moves to a position around 0.2 within a 0.05 deviation. The property can be represented by the ACTL formula Φ_2 as defined in Table 1. The properties of other tasks are formalized similarly. The formulas and the types of properties are shown in the table.

Table 1. Expected properties and their definitions in ACTL of the selected control tasks.

Task	ID	ACTL formula	Type	Meaning
MC	ϕ_1	$AF(p > 0.45)$	Liveness	The car always reaches the target finally.
	ϕ_2	$AG(\lvert p - 0.2\rvert < 0.05 \rightarrow v > 0.02)$	Safety	The car's speed should be greater than 0.02 at the position 0.2 within a 0.05 deviation.
PD	ϕ_3	$AG(\lvert\theta\rvert \leq \frac{\pi}{2})$	Safety	The pendulum's angle θ must always be in the preset range $[-\frac{\pi}{2}, \frac{\pi}{2}]$.
CP	ϕ_4	$AG_{t \leq n}(\lvert p\rvert \leq 2.4 \wedge a \leq \lvert 0.21\rvert)$	Safety	The cart always stays in the safe region and the pole cannot fall down in n time steps.
B1	ϕ_5	$AF(x_1 \in [0, 0.2] \wedge x_2 \in [0.05, 0.3])$	Liveness	The agent always reaches the target finally.
	ϕ_6	$AG(\lvert x_1\rvert \leq 1.5 \wedge \lvert x_2\rvert \leq 1.5)$	Safety	The agent always stays in the safe region.
B2	ϕ_7	$AF(target)$	Liveness	The agent always reaches the target finally.
	ϕ_8	$A((\lvert x_1\rvert \leq 1.5 \wedge \lvert x_2\rvert \leq 1.5)\ U\ target)$ $\vee\ AG(\lvert x_1\rvert \leq 1.5 \wedge \lvert x_2\rvert \leq 1.5)$	Safety	The agent must stay in the safe region until it reaches the target region.
Tora	ϕ_9	$AG_{t \leq n}(\lvert x_1\rvert \leq 1.5 \wedge \lvert x_3\rvert \leq 1.5)$	Safety	The agent can stay in the preset state space with n time steps.

Remarks. *target* is an atomic proposition *i.e.*, $x_1 \in [-0.3, 0.1] \wedge x_2 \in [-0.35, 0.5]$ in B2.

We compare the reliability and verifiability of all the trained DRL systems with respect to their predefined properties using both verification and simulation. The DRL systems trained in our approach can be naturally verified in our framework. For those trained in the conventional DRL approaches, our verification approach is not applicable because we cannot construct abstract Kripke structures for them. The main reason is that we cannot abstract the system states such that there is a unique action on all the actual states represented by the same abstract state. We therefore resort to the state-of-the-art reachability analysis tool Verisig 2.0 [25] to verify them. We also simulate all the trained systems in a fixed number of rounds and detect the occurrences of property violations. The purposes of the simulation are twofold: (i) to partially reflect the reliability of systems; and (ii) to validate the verification results in a bounded number of steps.

Table 2 shows the comparison results. We can observe that all the systems trained in our approach are successfully verified, and the corresponding properties hold on them. No violations are detected by simulation. For those systems trained in conventional DRL algorithms, only 8 out of 16 are successfully verified by Verisig. There are two cases, where Verisig returns **Unknown** when verifying ϕ_7 for task B2. It means that the verification fails because Verisig 2.0 cannot determine whether the destination region (defined by $x_1 \in [-0.3, 0.1] \wedge x_2 \in [-0.35, 0.5]$) must always be reached when it computes a larger region that overlaps the *target*. The extra part in the larger region may be an overestimation caused by the over-approximation. By simulation, we detect violations to ϕ_7. The violations can be considered as counterexamples to the property. The other properties such as ϕ_2, ϕ_3, ϕ_4, and ϕ_8 are not supported by Verisig 2.0. Among these unverified properties, we detect there exist violations by simulation for three of them. The violations indicate that the systems trained in conventional DRL approaches may not satisfy expected properties, and existing

Table 2. Comparison of the verification and simulation results between the DRL systems trained in our approach and conventional DRL algorithms, respectively.

Task	Network		Property	By *Trainify*				By conventional algorithms			
	A.F.	Size		T.T.	V.R.	V.T.	Vio.	T.T.	V.R.	V.T.	Vio.
MC	Sigmoid	2×16	ϕ_1	306	✓	26.8	0	297	✓	45.5	0
			ϕ_2	302	✓	5.9	0	297	N/A	–	0
	Sigmoid	2×200	ϕ_1	453	✓	29.1	0	441	✓	3709	0
			ϕ_2	462	✓	7.1	0	441	N/A	–	0
PD	ReLU	3×128	ϕ_3	771	✓	1.2	0	501	N/A	–	0
CP	ReLU	3×64	ϕ_4	135	✓	3266	0	101	N/A	–	**12**
B1	Tanh	2×20	ϕ_5	52	✓	89.0	0	31	✓	4.6	0
			ϕ_6	43	✓	5.3	0	31	✓	4.6	0
	Tanh	2×100	ϕ_5	32	✓	66	0	41	✓	28.2	0
			ϕ_6	25	✓	3.8	0	41	✓	28.2	0
B2	Tanh	2×20	ϕ_7	17	✓	1.2	0	9	**Unknown**	4.8	**27**
			ϕ_8	9	✓	1.3	0	9	N/A	-	0
	Tanh	2×100	ϕ_7	9	✓	1.3	0	11	**Unknown**	55.3	**23**
			ϕ_8	6	✓	1.7	0	11	N/A	–	0
Tora	Tanh	3×100	ϕ_9	402	✓	1132	0	217	✓	1271	0
	Tanh	3×200	ϕ_9	495	✓	1242	0	239	✓	6829	0

Remarks. A.F.: activation function; **T.T.**: average training time per iteration; **V.R.**: verification result; **V.T.**: average verification time per iteration; **Vio.**: the number of violations in simulation; **N/A**: not applicable; **Unknown**: verification fails. Time is recorded in seconds.

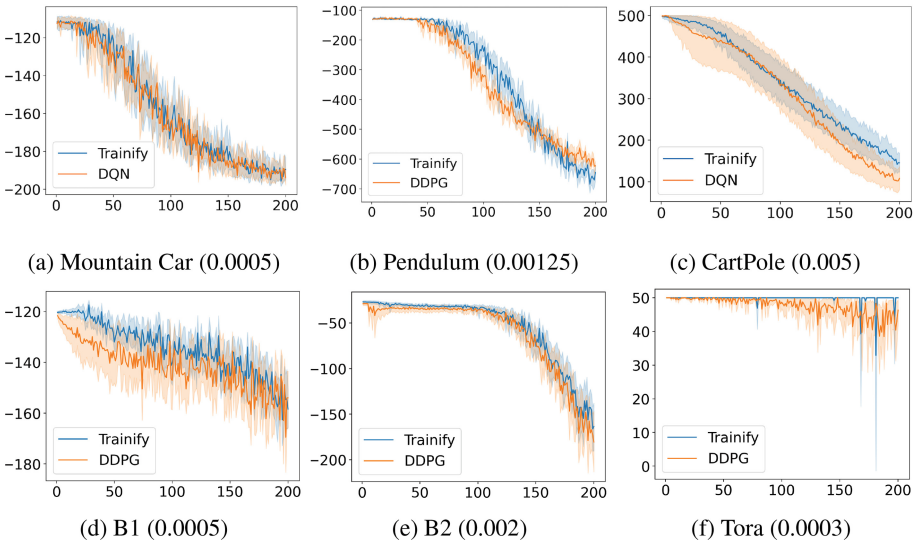
state-of-the-art verification tools cannot always verify them or find violations. Our approach can guarantee that the trained systems satisfy the properties. The simulation results show there are indeed no violations.

As for efficiency, on average, our approach costs slightly more time on the training because it takes extra time to look up the corresponding abstract state for an actual state at every training step. But the small-time overhead is worthwhile for the sake of being verifiable. Besides verifiability, another benefit from this extra time cost is that the efficiency of verification in our approach is not affected by the size and type of neural networks because we treat them as black-box in the verification. On the contrary, the efficiency of verifying the systems that are trained in conventional approaches is restricted by neural networks, as the verification time cost by Verisig 2.0 shows.

Based on the above analysis, we conclude that the reliability of the DRL systems developed in our approach are more trustworthy as their predefined properties are provably satisfied by the systems. Besides, their verification is more amenable and scalable than the systems trained in conventional DRL approaches.

5.4 Performance Comparison

We compare the performance of the DRL systems trained in our approach and the conventional approaches in terms of cumulative reward and robustness, respectively.



(a) Mountain Car (0.0005) (b) Pendulum (0.00125) (c) CartPole (0.005)

(d) B1 (0.0005) (e) B2 (0.002) (f) Tora (0.0003)

Fig. 8. Robustness comparison of the systems trained in our approach (blue) and in conventional approaches (orange). The number in the parentheses is the base of σ. For example, in Mountain Car, when the abscissa is equal to 50, $\sigma = 50 \times 0.0005 = 0.025$. (Color figure online)

Cumulative Reward. We record the cumulative reward by running each system for 100 episodes in the simulation environment and calculating the averages. A larger reward implies that a system has a better performance. Table 3 shows the cumulative reward of the six DRL systems trained in our approach and conventional approaches, respectively. All the trained systems can achieve almost optimal cumulative reward. Among the ten cases, the systems trained in our approach have better performances in four cases, equivalent in four cases, and lower in the rest two cases. Note that there is a difference, which is due to floating point errors, but it is almost negligible. In this sense, we say that the performance of the systems trained in the two different approaches is comparable.

Table 3. Comparison of accumulated reward.

Case	Alg.	Network		TRAINIFY	Base
MC	DQN	Sigmoid	2×16	**−112**	**−116**
		Sigmoid	2×200	**−110**	**−111**
PD	DDPG	ReLU	3×128	**−131**	**−133**
CP	DQN	ReLU	3×64	500	500
B1	DDPG	Tanh	2×20	−120	−120
		Tanh	2×100	**−117**	**−118**
B2	DDPG	Tanh	2×20	−29	−26
		Tanh	2×100	−27	−24
Tora	DDPG	Tanh	3×100	50	50
		Tanh	3×200	50	50

Another observation from the results is that a system with a bigger neural network produces a larger reward. This characteristic is shared by both our approach and the conventional approaches. Thus, we can increase the size of

networks and even modify network architectures for better performance in our approach. Such change will not cause the extra cost to the verification of the systems because our approach is entirely black-box, using the network only to output actions for the given abstract state.

Robustness. We demonstrate that the systems trained in our approach can be more robust than those trained in conventional DRL algorithms when the perturbation is set in a reasonable range. To examine the robustness, we add Gaussian noise to the actual states of systems and check the cumulative reward of the systems under different levels of perturbations. Given an actual state $s = (s_1, \ldots, s_n)$, we add a noise $X_1, ...X_n$ to s and obtain a perturbed state $s' = (s_1 + X_1, \ldots, s_n + X_n)$, where $X_i \sim \mathbf{N}(\mu, \sigma^2)$ for $1 \leq i \leq n$ with $\mu = 0$. We start with $\sigma = 0$ and increase it gradually.

Figure 8 shows the trend of cumulative reward of the systems with the increase of perturbations. For each system, we evaluate 200 different levels of perturbations, and for each level of perturbation, we conduct 20 repetitions to obtain the average and standard deviation of the reward, represented by the solid lines and shadows in Fig. 8. The general trend is that the cumulative reward deteriorate for all the systems that are trained in either of the approaches. The result is reasonable because the actions computed by neural networks are optimal to non-perturbed states but may not be optimal to the perturbed ones, leading to lower reward at some steps. However, we can observe that the decline ratio of the systems trained in our approach (blue) is smaller than the one trained in conventional approaches (orange). When $\sigma = 0$, the accumulated reward of the two systems for the same task is almost the same. With the increase of σ, the performance declines more slowly for the systems trained in our approach than for those trained in the conventional approaches when σ is in a reasonably small range. That is because a perturbed state may belong to the same abstract state as its original state, and thus has the optimal action. In this sense, we say the perturbation is *absorbed* by the abstract state and the neural networks become less sensitive to perturbations. Our additional experiments on these examples show that a larger abstraction granularity produces a more robust system.

6 Related Work

Our work has been inspired by several related works, which attempted to integrate formal methods and DRL approaches. We classify them into the following three categories.

Verification-in-the-Loop Training. Verification-in-the-loop training has been proposed for developing reliable AI-powered systems. A pioneering work is that Nilsson *et al.* proposed a correct-by-construction approach for developing Adaptive Cruise Control (ACC) by first formally defining safety properties in Linear Temporal Logic (LTL) and then computing the safe domain where the LTL specification can be enforced [36]. Wang *et al.* proposed a correct-by-construction control learning framework by leveraging verification during

training to formally guarantee that the learned controller satisfies the required reach-avoid property [49]. Lin *et al.* proposed an approach for training robust neural networks for general classification problems by fine-tuning the parameters in the networks based on the verification result [33]. Our work is a sequel of these previous works with new features of training on abstract states, counterexample-guided abstraction and refinement, and supporting more complex properties.

Safe DRL via Formal Methods. Most of the existing approaches for formal verification of DRL systems follow the *train-then-verify* style. Bacci and Parker [3] proposed an approach to split an abstract domain into fine-grained ones and compute their successor abstract states separately for probabilistic model checking of DRL systems. The approach can reduce the overestimation and meanwhile construct a transition system upon abstract states, which allows us to verify more complex liveness and probabilistic properties than safety using bounded model checking [29] and probabilistic model checking. A criteria of subdividing an abstract domain is to ensure that all the states in the same sub-domain have the same action. Identifying these sub-domains is computationally expensive because it relies on iterative branching and bounding [3]. Furthermore, these approaches need to compute the output range of the neural networks on the abstract domains, and therefore are restricted to specific types and scales of networks. Besides model checking, reachability analysis [13,16,25,46] has been well studied to ensure the safety of DRL systems. The basic idea is to over-approximate system dynamics and neural networks to compute over-estimated safe regions and check whether they have interactions with unsafe regions. However, large overestimation, limited scalability, and requirements on specific network architectures are the common restrictions of these approaches. Online verification [47] and runtime monitoring [18] in formal methods is another lightweight but effective means to detect potential flaws timely during system execution. Another direction is to synthesize *safe shields* [7,54] and barrier functions [53] to prevent agents from adopting dangerous actions. A strong assumption of these methods is that the valid safe states set is given in advance. However, computing valid safe states set may be computationally intensive, and it is restricted to safety properties.

Abstraction and State Discretization in DRL. Abstraction in DRL has gained more attention in recent years. Abel presented a theory of abstraction for DRL in his dissertation and concluded that learning on abstraction can be more efficient while preserving near-optimal behaviors [1]. Abel's abstraction theory is focused on the systems with finite state space for learning efficiency. Our work demonstrates another advantage of learning on abstraction, *i.e.*, *formal reliability guarantee* to trained systems even with infinite state space.

The state-space abstraction approach in our framework is also inspired by *state space discretization*, a technique for discretizing continuous state space, by which a finer partition of the state-action space is maintained during training for higher payoff estimates [41,42]. Our work shows that, after being integrated with formal verification, state-space discretization is also useful in developing highly reliable DRL systems without loss of performance. In addition, our CEGAR-

driven approach provides a flexible mechanism for fine-tuning the granularity of discretization to reach an appropriate balance between system performance and the scale of state space for formal verification.

7 Discussion and Conclusion

We have presented a novel verification-in-the-loop framework for training and verifying DRL systems, driven by counterexample-guided abstract and refinement. The framework can be used to train reliable DRL systems with their desired properties on safeties and functionalities formally verified, without compromising system performances. We have implemented a prototype Trainify and evaluated it by training six classic control problems from public benchmarks. The experimental results showed that the systems trained in our approach were more reliable and verifiable than those trained in conventional DRL approaches, while their performances are comparable or even better than the latter.

Our verification-in-the-loop training approach sheds light on a new search direction for developing reliable and verifiable AI-empowered systems. It follows the idea of correctness-by-construction in traditional trustworthy software system development and makes it possible to take system properties (or requirements) into account during the training process. It also reveals that (i) it is not necessary to learn on actual data to build high-performance (e.g., high reward and robust) DRL systems, and (ii) abstraction is an effective means to deal with the challenges in verifying DRL systems and shall be introduced earlier during training, rather than an *ex post facto* method in verification.

Our work would inspire more research in this direction. One important research objective is to investigate appropriate abstractions for the DRL systems with high dimensions. In our current framework, we adopt the simplest interval abstraction that suffices to the systems with low dimensions. It would be interesting to investigate more sophisticated abstractions such as floating-point polyhedra combined with intervals, designed mainly for neural networks [43], to those high-dimensional DRL systems. Another direction is to extend our framework to non-deterministic DRL systems. In the non-deterministic case, a neural network returns both actions and their corresponding probabilities. We can associate probabilities to state transitions and obtain a probabilistic model. The model can be naturally verified using existing probabilistic model checkers such as Prism [30]. Thus, we believe that our approach is also applicable to those systems after a slight extension. It would be another piece of our future work.

Acknowledgments. The authors thank all the anonymous reviewers and Katz Guy from the Hebrew University of Jerusalem for their valuable comments on this work. The work has been supported by National Key Research Program (2020AAA0107800), Shanghai Science and Technology Commission (20DZ1100300), Shanghai Trusted Industry Internet Software Collaborative Innovation Center, Shanghai AI Innovation and Development Fund (2020-RGZN-02026), Shenzhen Institute of AI and Robotics for Society (AC01202005020), NSFC-ISF Joint Program (62161146001,3420/21) and NSFC project (61872146).

References

1. Abel, D.: A theory of abstraction in reinforcement learning. Dissertation, Brown University (2020)
2. Bacci, E., Giacobbe, M., Parker, D.: Verifying reinforcement learning up to infinity. In: IJCAI 2021, Montreal, Canada, pp. 2154–2160. ijcai.org (2021)
3. Bacci, E., Parker, D.: Probabilistic guarantees for safe deep reinforcement learning. In: Bertrand, N., Jansen, N. (eds.) FORMATS 2020. LNCS, vol. 12288, pp. 231–248. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57628-8_14
4. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
5. Brockman, G., et al.: OpenAI Gym (2016). arXiv:1606.01540
6. Casagrande, A.: pyModelChecking (2020). https://github.com/albertocasagrande/pyModelChecking
7. Cheng, R., Orosz, G., Murray, R.M., Burdick, J.W.: End-to-end safe reinforcement learning through barrier functions for safety-critical continuous control tasks. In: AAAI 2019, vol. 33, pp. 3387–3395. AAAI Press (2019)
8. Clarke, E., et al.: Abstraction and counterexample-guided refinement in model checking of hybrid systems. Int. J. Found. Comput. Sci. **14**(04), 583–604 (2003)
9. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15
10. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of model checking. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8
11. Cousot, P.: Abstract interpretation. ACM Comput. Surv. (CSUR) **28**(2), 324–328 (1996)
12. Du, S., Lee, J., Li, H., Wang, L., Zhai, X.: Gradient descent finds global minima of deep neural networks. In: ICML 2019, pp. 1675–1685. PMLR (2019)
13. Dutta, S., Chen, X., Sankaranarayanan, S.: Reachability analysis for neural feedback systems using regressive polynomial rule inference. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 157–168 (2019)
14. Emerson, E.A., Halpern, J.Y.: "sometimes" and "not never" revisited: on branching versus linear time temporal logic. J. ACM (JACM) **33**(1), 151–178 (1986)
15. Emerson, E.A., Sistla, A.P.: Deciding full branching time logic. Inf. Control **61**(3), 175–201 (1984)
16. Fan, J., Huang, C., Chen, X., Li, W., Zhu, Q.: ReachNN*: a tool for reachability analysis of neural-network controlled systems. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 537–542. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_30
17. Faust, A., Ruymgaart, P., Salman, M., Fierro, R., Tapia, L.: Continuous action reinforcement learning for control-affine systems with unknown dynamics. IEEE/CAA J. Automatica Sinica **1**(3), 323–336 (2014)
18. Fulton, N., Platzer, A.: Safe reinforcement learning via formal methods: toward safe control through proof and learning. In: AAAI 2018, pp. 6485–6492. AAAI Press (2018)

19. Gallestey, E., Hokayem, P.: Lecture notes in nonlinear systems and control (2019)
20. Gilpin, L., Bau, D., Yuan, B.Z., et al.: Explaining explanations: an overview of interpretability of machine learning. In: DSAA 2018, pp. 80–89 (2018)
21. Gomes, L.: When will Google's self-driving car really be ready? It depends on where you live and what you mean by "ready.". IEEE Spectr. **53**(5), 13–14 (2016)
22. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: SIGMOD 1984, pp. 47–57. ACM (1984)
23. Hasanbeig, M., Kroening, D., Abate, A.: Towards verifiable and safe model-free reinforcement learning. In: CEUR Workshop Proceedings (2020)
24. Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., Meger, D.: Deep reinforcement learning that matters. In: AAAI 2018, pp. 3207–3214. AAAI Press (2018)
25. Ivanov, R., Carpenter, T., Weimer, J., Alur, R., Pappas, G., Lee, I.: Verisig 2.0: verification of neural network controllers using Taylor model preconditioning. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 249–262. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_11
26. Jackson, D.: Abstract model checking of infinite specifications. In: Naftalin, M., Denvir, T., Bertran, M. (eds.) FME 1994. LNCS, vol. 873, pp. 519–531. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58555-9_113
27. Jankovic, M., Fontaine, D., KokotoviC, P.V.: Tora example: cascade-and passivity-based control designs. IEEE Trans. Control Syst. Technol. **4**(3), 292–297 (1996)
28. Johnson, T.T., Manzanas Lopez, D., Musau, P., et al.: Arch-comp20 category report: artificial intelligence and neural network control systems (AINNCS) for continuous and hybrid systems plants. EPiC Ser. Comput. **74**, 107–173 (2020)
29. Kazak, Y., Barrett, C., Katz, G., Schapira, M.: Verifying deep-RL-driven systems. In: 2019 Workshop on Network Meets AI & ML, pp. 83–89. ACM (2019)
30. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
31. Li, Y.: Deep reinforcement learning: an overview. arXiv preprint arXiv:1701.07274 (2017)
32. Lillicrap, T.P., Hunt, J.J., Pritzel, A., et al.: Continuous control with deep reinforcement learning. In: ICLR 2016. OpenReview.net (2016)
33. Lin, X., Zhu, H., Samanta, R., Jagannathan, S.: Art: abstraction refinement-guided training for provably correct neural networks. In: FMCAD, pp. 148–157. AAAI Press (2020)
34. Mnih, V., Kavukcuoglu, K., Silver, D., et al.: Playing Atari with deep reinforcement learning. CoRR abs/1312.5602 (2013)
35. Mnih, V., Kavukcuoglu, K., Silver, D., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
36. Nilsson, P., Hussien, O., Balkan, A., et al.: Correct-by-construction adaptive cruise control: two approaches. IEEE Trans. Control Syst. Technol. **24**(4), 1294–1307 (2015)
37. Ohn-Bar, E., Trivedi, M.M.: Looking at humans in the age of self-driving and highly automated vehicles. IEEE Trans. Intell. Veh. **1**(1), 90–104 (2016)

38. Pyeatt, L.D., Howe, A.E.: Decision tree function approximation in reinforcement learning. Technical report, ISAS 2011 (2011)
39. Schmidt, L.M., Kontes, G., Plinge, A., Mutschler, C.: Can you trust your autonomous car? interpretable and verifiably safe reinforcement learning. In: 2021 IEEE Intelligent Vehicles Symposium (IV), pp. 171–178. IEEE (2021)
40. Shalev-Shwartz, S., Shammah, S., Shashua, A.: Safe, multi-agent, reinforcement learning for autonomous driving. CoRR abs/1610.03295 (2016). http://arxiv.org/abs/1610.03295
41. Sinclair, S., Wang, T., Jain, G., Banerjee, S., Yu, C.: Adaptive discretization for model-based reinforcement learning. In: NeurIPS 2020. vol. 31, pp. 3858–3871 (2020)
42. Sinclair, S.R., Banerjee, S., Yu, C.L.: Adaptive discretization for episodic reinforcement learning in metric spaces. Proc. ACM Measur. Anal. Comput. Syst. **3**(3), 1–44 (2019)
43. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. In: POPL 2019, pp. 1–30. ACM (2019)
44. Srinivasan, K., Eysenbach, B., Ha, S., Tan, J., Finn, C.: Learning to be safe: deep RL with a safety critic. arXiv preprint arXiv:2010.14603 (2020)
45. Stevia, P., Mindom, N., Nikanjam, A., Khomh, F., Mullins, J.: On assessing the safety of reinforcement learning algorithms using formal methods. arXiv preprint arXiv:2111.04865 (2021)
46. Tran, H.D., Cai, F., Diego, M.L., Musau, P., Johnson, T.T., Koutsoukos, X.: Safety verification of cyber-physical systems with reinforcement learning control. ACM Trans. Emb. Comput. Syst. **18**(5s), 1–22 (2019)
47. Van Wesel, P., Goodloe, A.E.: Challenges in the verification of reinforcement learning algorithms. NASA STI Program (2017)
48. Virtanen, P., Gommers, R., Oliphant, T.E., et al.: SciPy 1.0: fundamental algorithms for scientific computing in python. Nat. Meth. **17**, 261–272 (2020)
49. Wang, Y., Huang, C., Wang, Z., Wang, Z., Zhu, Q.: Verification in the loop: correct-by-construction control learning with reach-avoid guarantees. arXiv preprint arXiv:2106.03245 (2021)
50. Xiang, W., Tran, H.D., Johnson, T.T.: Output reachable set estimation and verification for multilayer neural networks. IEEE Trans. Neural Networks Learn. Syst. **29**(11), 5777–5783 (2018)
51. Xiong, Z., Jagannathan, S.: Scalable synthesis of verified controllers in deep reinforcement learning. arXiv preprint arXiv:2104.10219 (2021)
52. Yampolskiy, R.V.: Unexplainability and incomprehensibility of AI. J. Artif. Intell. Conscious. **7**(2), 277–291 (2020)
53. Yang, Z., et al.: An iterative scheme of safe reinforcement learning for nonlinear systems via barrier certificate generation. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 467–490. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_22
54. Zhu, H., Xiong, Z., Magill, S., Jagannathan, S.: An inductive synthesis framework for verifiable reinforcement learning. In: PLDI 2019. pp. 686–701. ACM (2019)