Verifying Reinforcement Learning up to Infinity

Edoardo Bacci[1] , **Mirco Giacobbe**[2] , **David Parker**[1]
[1]University of Birmingham
[2]University of Oxford
{exb461, d.a.parker}@cs.bham.ac.uk, mirco.giacobbe@cs.ox.ac.uk

Abstract

Formally verifying that reinforcement learning systems act safely is increasingly important, but existing methods only verify over finite time. This is of limited use for dynamical systems that run indefinitely. We introduce the first method for verifying the time-unbounded safety of neural networks controlling dynamical systems. We develop a novel abstract interpretation method which, by constructing adaptable template-based polyhedra using MILP and interval arithmetic, yields sound—safe and invariant—overapproximations of the reach set. This provides stronger safety guarantees than previous time-bounded methods and shows whether the agent has generalised beyond the length of its training episodes. Our method supports ReLU activation functions and systems with linear, piecewise linear and non-linear dynamics defined with polynomial and transcendental functions. We demonstrate its efficacy on a range of benchmark control problems.

1 Introduction

Reinforcement learning (RL) has reached super-human capabilities on many challenging problems and is being increasingly applied to cyber-physical tasks, such as robot control and autonomous driving [Kendall *et al.*, 2019]. The criterion for training and evaluating RL agents is traditionally their performance, that is, how quickly and efficiently they solve their task. However, for agents that interact with critical environments, performance must meet safety: not only is it required that positive outcomes eventually happen, but also that negative ones do not [Fulton and Platzer, 2018; Luckcuck *et al.*, 2019]. Safety is subtle, because a system is truly safe only if it avoids danger regardless of how long it is left running. Determining whether an RL system is safe for *unbounded time* addresses both a formal verification question, providing stronger guarantees of correctness than bounded verification, and a machine learning question, indicating whether the learning algorithm has generalised a strategy beyond the length of the episodes used to train it. Verifying RL requires reasoning about the dynamics of the environments together with the learned agents which, in modern RL, are neural networks. For the first time, we treat the automated (and sound) time-unbounded verification of neural networks interacting with dynamical systems.

Safety analysis for neural networks has been studied before for bounded settings. One example is classification, whose well-known vulnerability to adversarial attacks has been analysed using gradient descent, mixed-integer linear programming (MILP), and satisfiability modulo theories (SMT) [Moosavi-Dezfooli *et al.*, 2016; Huang *et al.*, 2017; Ehlers, 2017; Bunel *et al.*, 2018; Katz *et al.*, 2019]. Search-based algorithms of this kind are inherently bounded, unlike *abstract interpretation* methods. Abstract interpretation computes a representation of the set of reachable states and checks whether it avoids a set of bad states. Methods for the abstract interpretation of neural networks have borrowed from the analysis of numerical programs, and have been applied to adversarial attacks [Gehr *et al.*, 2018; Singh *et al.*, 2019], output range analysis [Xiang *et al.*, 2018; Dutta *et al.*, 2018], and time-bounded verification of RL [Tran *et al.*, 2020; Bacci and Parker, 2020]. Time-unbounded verification is more difficult because it requires that the abstraction is both *safe*, i.e., disjoint from the bad states, and *invariant*, i.e., no other states are reachable from it; none of the available approaches, as is, have been demonstrated to achieve both requirements on RL problems.

We present the first technique for verifying whether a neural network controlling a dynamical system maintains the system within a safe region for unbounded time. For this purpose, we overapproximate the reach set using *template polyhedra*, i.e., polyhedra whose shape is determined by a set of directions, the template [Sankaranarayanan *et al.*, 2005]. Traditional interval and octagonal abstractions have rigid shapes which often produce abstractions that are too coarse to be safe or too tight to be invariant. By contrast—with an appropriate choice of directions—template polyhedra can be adapted to the verification problem making the abstraction tight only where necessary and thus facilitating the identification of safe invariants [Bogomolov *et al.*, 2017; Frehse *et al.*, 2018].

We formulate the problem of computing template polyhedra as an optimization problem. For this purpose, we introduce an MILP encoding for a sound abstraction of neural networks with ReLU activation functions acting over discrete-time systems. We support linear, piecewise linear and non-linear systems defined with polynomial and transcendental functions. For the latter, we combine MILP with interval arithmetic.

We propose a safety verification workflow where agents trained with any, possibly model-free, RL technique are verified against a model of the environment. Every model is accompanied with user-defined templates which, as we experimentally demonstrate, suffice to verify multiple agents. Upon every successful verification result we thus certify that an agent is safe w.r.t. a model, which determines our safety specification. Ultimately, we provide formal guarantees that are equivalent to (or stronger than) those of agents that are trained or enforced to be safe [Alshiekh *et al.*, 2018; Cheng *et al.*, 2019; Hasanbeig *et al.*, 2019; Li and Bastani, 2020],yet without imposing constraints on the agent or the RL process.

We demonstrate that our method effectively verifies agents trained over three benchmark control problems [Jaeger *et al.*, 2019; Tran *et al.*, 2020; Brockman *et al.*, 2016]. We additionally show that an alternative time-unbounded verification approach built upon range analysis fails in all cases.

In summary, we introduce the first method for the formal time-unbounded safety analysis of RL systems. We have built a software prototype and demonstrate the efficacy of our method over a range of benchmarks.

2 Safety Analysis of Reinforcement Learning

A time-invariant *controlled dynamical system* with discrete actions and over discrete time consists of an n-dimensional vector of real-valued state variables x, an m-dimensional vector of real-valued observable variables y, and a natural number Σ of input actions available to an external agent. The set A denotes the set of actions $\{a \in \mathbb{Z} \colon 1 \leq a \leq \Sigma\}$. The system dynamics are determined by a difference equation

$$x_t = f(x_{t-1}, a_t) + c_t, \qquad c_t \in C, \qquad x_0 \in X_0, \tag{1}$$

where $x_t \in \mathbb{R}^n$, $a_t \in A$, and c_t respectively denote state, input action, and control disturbance at time t. The set $C \subset \mathbb{R}^n$ is the space of control disturbances, $X_0 \subset \mathbb{R}^n$ is the space of initial conditions, and $f \colon \mathbb{R}^n \times A \to \mathbb{R}^n$ is the update function. An observation function $g \colon \mathbb{R}^n \to \mathbb{R}^m$ and a space of observation disturbances $D \subset \mathbb{R}^m$ determine the observable values at time t from a state:

$$y_t = g(x_t) + d_t, \qquad d_t \in D. \tag{2}$$

A trajectory of the system is an infinite sequence of states and actions in alternation

$$x_0 a_1 x_1 a_2 x_2 \ldots \tag{3}$$

where every state $x_1, x_2, \ldots$ is determined by Eq. (1); every action $a_1, a_2, \ldots$ is determined by an external *agent* $\sigma \colon \mathbb{R}^m \to A$ from the observation at the previous step, i.e.,

$$a_t = \sigma(y_{t-1}). \tag{4}$$

We target the *safety verification* question for controlled dynamical systems. Let $B \subset \mathbb{R}^n$ be a set of bad states. Verifying the safety of a system consists of deciding whether, for every trajectory $x_0 a_1 x_1 a_2 x_2 \ldots$, we have that $x_t \notin B$ for all $t = 0, \ldots, \infty$. Dually, it consists of determinining whether there exists a finite prefix $x_0 a_1 x_1 \ldots a_k x_k$ such that $x_k \in B$. In the former case we say that the system is safe; in the latter we say that it is unsafe.

(a) (b)

Figure 1: Adaptive cruise control: a good and a bad state.

Example 1. *Adaptive cruise control is a paradigmatic example for the safety of an RL system [Desjardins and Chaib-draa, 2011; Tran* et al.*, 2020]. In its simplest form, it consists of two vehicles, ego and lead, moving in a straight line. An agent should control ego so that it stays at some close and safe distance from lead. State variables x_v, x'_v, and x''_v resp. determine position, speed, and acceleration of each vehicle $v = ego, lead$. The observation function exposes vehicles distance $x_{lead} - x_{ego}$ and speed of ego x'_{ego}; both observables are subject to a disturbance. Lead proceeds at a constant speed of $28\,\mathrm{m\,s^{-1}}$ and, at every step, the agent can either decelerate (action 1) or accelerate ego by $1\,\mathrm{m\,s^{-2}}$ (action 2). Update and observation functions are shown in Sect. 4.2. The agent is safe only if the distance is positive along every trajectory (Fig. 1a); every other condition indicates that the vehicles have crashed (Fig. 1b). The set of bad states is thus defined by the constraint $x_{lead} \leq x_{ego}$. Trivially, an agent that always decelerates is safe; however, safety must coexist with performance, which rewards the agent for keeping ego close to lead.*

For the purpose of training an agent using RL, we augment the system with the probability distributions λ_{X_0}, λ_C, and λ_D for the sets X_0, C, and D, respectively. We require that $\mathrm{supp}(\lambda_{X_0}) = X_0$, $\mathrm{supp}(\lambda_C) = C$, and $\mathrm{supp}(\lambda_D) = D$, where $\mathrm{supp}(\lambda) = \{x \colon \lambda(x) > 0\}$ is the support of distribution λ. This induces a discrete-time *partially observable Markov decision process* (POMDP) with finite actions and possibly uncountable state space and branching. Precisely, we let $\mathbb{R}^n$ be the state space and $\mathbb{R}^m$ be the observation space, together with the σ-algebras $\mathcal{F}_X$ and $\mathcal{F}_Y$ of measurable subsets of respectively $\mathbb{R}^n$ and $\mathbb{R}^m$. The probability of beginning from a set X is given by the initial belief $\mu \colon \mathcal{F}_X \to [0, 1]$, i.e.,

$$\mu(X) = \int_X \lambda_{X_0}(x)\, dx. \tag{5}$$

The set of actions A corresponds to that of the original controlled dynamical system. The transition probability function $T \colon \mathbb{R}^n \times A \times \mathcal{F}_X \to [0, 1]$ is thus

$$T(x, a, Z) = \int_Z \lambda_C(z - f(x, a))\, dz, \tag{6}$$

which denotes the probability that, from state x and after choosing action a, the process jumps to a target set Z. The observation probability function $O \colon \mathbb{R}^n \times \mathcal{F}_Y \to [0, 1]$ is

$$O(x, Y) = \int_Y \lambda_D(y - g(x))\, dy, \tag{7}$$

that is, the probability of observing subset Y from state x. Finally, a reward function $R \colon \mathbb{R}^m \to \mathbb{R}$ maps observations to reward values. We discuss in Sect. 4 how we design rewards functions for obtaining performant and safe agents using RL.

Agents are given in the form of neural networks. We consider neural networks with ReLU activation functions, m input neurons, l hidden layers with respectively $h_1, \ldots, h_l$ neurons, and Σ output neurons. The variable vectors $z_0, \ldots, z_{l+1}$ denote the values of the neurons at each layer. The input layer z_0 is assigned from the system observation y, every hidden layer is determined according to the equation

$$z_i = \mathrm{ReLU}(W_i z_{i-1} + b_i), \qquad \text{for } i = 1, \ldots, l, \quad (8)$$

and the output layer according to $z_{l+1} = W_{l+1} z_l$. Each matrix W_i denotes the weights between any other $(i-1)$-th and i-th layers, and each vector b_i denotes the respective biases. The function ReLU applies $\max\{\cdot, 0\}$ element-wise to its h_i-dimensional argument. The output action is determined by the index of the output neuron whose value is the highest; in other words, the neural network defines the agent

$$\sigma(z_0) = \operatorname*{argmax}_{j \in A} \langle e_j, z_{l+1} \rangle, \quad (9)$$

where e_j is the j-th standard unit vector of $\mathbb{R}^\Sigma$ and $\langle \cdot, \cdot \rangle$ denotes scalar product. Altogether, the neural network acts as a classifier from observations to actions.

We train our agents over the POMDP induced by the distributions over initial and disturbance sets. Then, we verify the safety of the dynamical system controlled by the obtained network. We tackle time-unbounded safety verification; for this purpose, we introduce a technique for constructing coarse yet safe abstractions of the reach sets of these neurally controlled dynamical systems.

3 Template-based Polyhedral Abstractions for Neurally Controlled Dynamical Systems

We employ abstract interpretation for constructing a sound overapproximation of the reach set of the system. Specifically, we compute a sequence of abstract sets of states in $\mathbb{R}^n$

$$X_{t+1} = \mathsf{post}(X_t) \quad (10)$$

for increasing $t \geq 0$, where post—the *post operator*—ensures that X_{t+1} overapproximates the states that are reachable after one step from X_t. Time-unbounded safety verification succeeds if our procedure finds a finite $k \geq 1$ such that the sequence up to k is

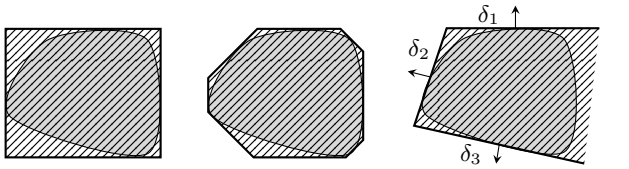
invariant $X_k \subseteq \cup\{X_0, \ldots, X_{k-1}\}$ and

safe $\cup\{X_0, \ldots, X_k\} \cap B = \emptyset$.

The procedure computes X_t iteratively for increasing t and checks both conditions at each step. If both are satisfied the procedure terminates concluding that the system is safe; if safety is violated it terminates with an inconclusive answer. This procedure may, in the worst case, not terminate. We present a post operator that computes X_t in the form of finite unions of template polyhedra; as our experiments show (Sect. 4), this yields safe and invariant abstractions in practice.

We call a finite set of directions $\Delta \subset \mathbb{R}^n$ a *template*. A Δ-polyhedron is a polyhedron whose facets are normal to the directions in Δ. The Δ-*polyhedron of* X, where X is a convex set in $\mathbb{R}^n$, is the tightest Δ-polyhedron enclosing X:

$$\cap\{\{x \colon \langle \delta, x \rangle \leq \rho_X(\delta)\} \colon \delta \in \Delta\}, \quad (11)$$



Rectangle (a) Octagon (b) $\{\delta_1, \delta_2, \delta_3\}$-polyhedron (c)

Figure 2: Template polyhedra (hatched areas) of a set (gray area).

where $\rho_X(\delta) = \sup\{\langle \delta, x \rangle \colon x \in X\}$ is the support function of X. Special cases of template polyhedra are rectangles (i.e., intervals) and octagons (Fig. 2a and b), which are determined by specific templates. In addition, by using fewer, well-chosen directions, template polyhedra let us construct sufficiently tight yet unbounded polyhedral abstractions (Fig. 2c).

We compute template polyhedra over a symbolic representation of the post. We split the post computation into a partitioning P_t (a set of sets in $\mathbb{R}^n$) that overapproximates the state that are reachable after one step from X_t, i.e.,

$$\cup P_t \supseteq \{f(x, \sigma(g(x) + d)) + c \colon c \in C, d \in D, x \in X_t\}. \quad (12)$$

As we show below, we build the partitioning from the piecewise structure of the system and represent its elements symbolically. Then, for every symbolic representation we construct a template polyhedron by optimising in the directions of Δ. Our post is the union of these template polyhedra:

$$\mathsf{post}(X_t) = \cup\{\Delta\text{-polyhedron of } \operatorname{conv} P' \colon P' \in P_t\}, \quad (13)$$

where $\operatorname{conv} P'$ denotes the convex hull of the members of $P' \subset \mathbb{R}^n$. The post produces a union of convex polyhedral overapproximations.

Neurally controlled dynamical systems often have piecewise dynamics. The discrete action space naturally induces a case split in the update function. Also, some systems may have dynamics that switch between two or more behaviours according to guard conditions over the state (see, e.g., Sect. 4.1). Formally, each case split is a partial function from a set $F \subset (\mathbb{R}^n \times A \to \mathbb{R}^n)$ s.t. $f = \cup F$ and f is total. Likewise, this case split and the encoding below also applies to the observation function g; for simplicity, we only refer to f.

We compute $\mathsf{post}(X_t)$ using optimisation. We express an encoding for every combination of action $a \in A$, case split $f' \in F$ of the update function, and convex polyhedron X' from the finite union of convex polyhedra X_t; each combination induces an element P' of P_t. For a direction $\delta \in \Delta$, we solve the following problem:

$$\begin{array}{ll} \text{maximize} & \langle \delta, p' \rangle \\ \text{subject to} & \sigma(y) = a, \\ & p' = f'(x', a) + c,\ x' \in \mathrm{dom}(f'(\cdot, a)) \\ & y = g(x') + d, \\ & c \in C, \quad d \in D, \quad x' \in X', \end{array} \quad (14)$$

over the variables $c, x', p' \in \mathbb{R}^n$ and $y, d \in \mathbb{R}^m$. The solution provides the value of $\rho_{\operatorname{conv} P'}(\delta)$ which, computed over all $\delta \in \Delta$, yields the Δ-polyhedron of P' (see Eq. (11)); in turn, the polyhedron yields an element of the post (see Eq. (13)).

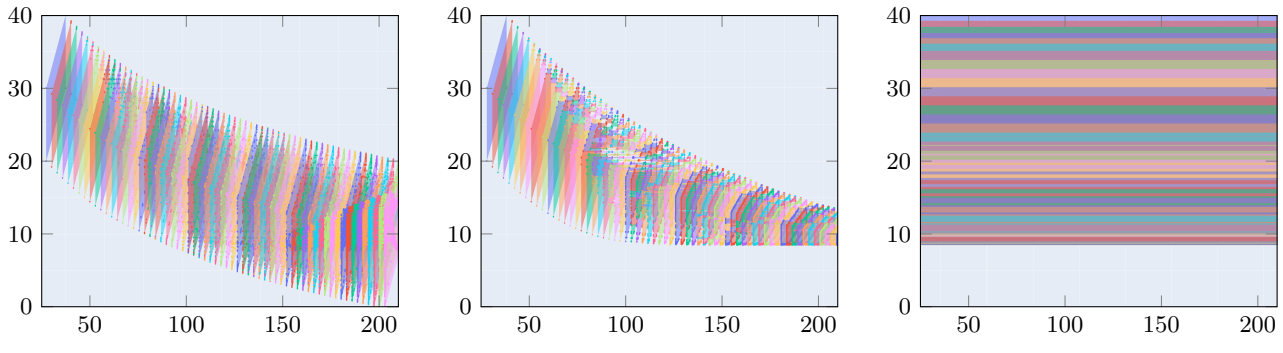


Figure 3: Abstract reach sets of a neural network for adaptive cruise control using different templates (Ex. 2). Plots are projected onto vehicles distance (y-axis) and position of lead (x-axis) and constrained within a window, as shown; different colours correspond to different time steps.

The optimisation problem consists of a linear objective function and constraints for, respectively, the action chosen by the neural network, update and observation functions, and disturbance and input sets. We assume that the disturbance sets C and D are convex polyhedra, and that the initial set X_0 is a union of convex polyhedra similarly to all other X_t for $t \geq 1$. For partial functions $f'(\cdot, a)$, we assume that the domains of definition are given as convex polyhedra. Consequently, the constraints for C, D, X', and $\mathrm{dom}(f'(\cdot, a))$ are expressed with systems of linear inequalities. Our encoding for the constraint over the neural network $\sigma(y) = a$ and our overapproximation of non-linear functions introduce integer variables, as we show below. The optimisation problem thus results in an MILP.

The network selects action a when the value of the a-th output neuron is larger than the value of all other output neurons (Eq. (9)). The constraint for $\sigma(y) = a$ is thus

$$\langle e_j - e_a, z_{l+1} \rangle \leq 0 \quad \text{for } j = 1, \ldots, \Sigma, \tag{15}$$

where $z_{l+1} \in \mathbb{R}^{\Sigma}$ is a variable for the value of the output layer. For each hidden layer i, we add to the optimisation problem a real variable $z_i \in \mathbb{R}^{h_i}$ for the values of the neurons in the layer, plus an integer variable $z'_i \in \mathbb{Z}^{h_i}$ for the activation status of the ReLU function. We encode the ReLU function using a big-M encoding [Tjeng *et al.*, 2019]. For the hidden layers, we add constraints

$$\begin{array}{ll} 0 \leq z_i - W_i z_{i-1} - b_i \leq M z'_i & \text{for } i = 1, \ldots, l, \\ 0 \leq z_i \leq M - M z'_i & \text{〃} \\ 0 \leq z'_i \leq 1 & \text{〃} \end{array} \tag{16}$$

For output and input layers, we add $z_{l+1} = W_{l+1} z_l$ and $z_0 = y$. Constant $M \in \mathbb{R}$ is an upper bound for the values a neuron can take, which we set to a sufficiently large value so as not to constrain the system's states.

Linear update (and observation) functions are encoded directly into the MILP using linear equalities. For non-linear constraints defined with polynomials or transcendental functions, we introduce an overapproximation based on interval arithmetic. Constraint $p' = f'(x', a) + c$ is an n-dimensional system of equalities. We identify the equations within the the system that are non-linear and let p'_{N}, f'_{N}, and c_{N} be the corresponding projection for resp. p', and f', and c. Moreover, we let x'_{N} be the largest subset of variables in x' that appear in these non-linear equations. The non-linear components thus form the reduced system

$$p'_{\mathsf{N}} = f'_{\mathsf{N}}(x'_{\mathsf{N}}, a) + c_{\mathsf{N}}. \tag{17}$$

We encode the remaining linear components exactly, using linear equalities, whereas we overapproximate Eq. (17). First, we construct a bounding box of X'_{N}; note that we ensure beforehand that X'_{N} is bounded with an appropriate template choice (see Sect. 4.3). Then, we partition the bounding box into a grid of intervals $[\underline{\xi}_1, \bar{\xi}_1], \ldots, [\underline{\xi}_\kappa, \bar{\xi}_\kappa]$. For every element $i = 1, \ldots, \kappa$, we compute using interval arithmetic an output interval $[\underline{\pi}_i, \bar{\pi}_i]$ for the image of $[\underline{\xi}_i, \bar{\xi}_i]$ though $f'_{\mathsf{N}}(\cdot, a)$. As a result, we obtain a lookup table that associates input intervals to output intervals. We encode this table by adding to the MILP the integer variables $\zeta_1, \ldots \zeta_\kappa \in \mathbb{Z}$, each of which represents an active interval, and the following constraints:

$$\begin{array}{l} \sum_{i=1}^{\kappa} \underline{\xi}_i - \underline{\xi}_i \cdot \zeta_i \leq x'_{\mathsf{N}} \leq \sum_{i=1}^{\kappa} \bar{\xi}_i - \bar{\xi}_i \cdot \zeta_i \\ \sum_{i=1}^{\kappa} \underline{\pi}_i - \underline{\pi}_i \cdot \zeta_i \leq p'_{\mathsf{N}} - c_{\mathsf{N}} \leq \sum_{i=1}^{\kappa} \bar{\pi}_i - \bar{\pi}_i \cdot \zeta_i \\ \sum_{i=1}^{\kappa} \zeta_i = \kappa - 1 \\ 0 \leq \zeta_i \leq 1 \qquad \text{for } i = 1, \ldots, \kappa. \end{array} \tag{18}$$

We tune the precision of the overapproximation by fixing a desired granularity for output intervals, a maximal diameter, and iteratively split the input intervals until that is attained.

Example 2. *We trained a neural network for adaptive cruise control (Ex. 1) by rewarding the agent for keeping a safety distance of* 10 m*; the vehicles start from a range of distances between* 20 *and* 40 m*. We employed our method for analysing its safety using three different abstraction templates: rectangles, octagons and a custom template designed for this system (see Sect. 4.2). Rectangles produce an excessively coarse abstraction which hit distance zero: the bad state (Fig. 3a). Unlike rectangles, octagons keep track of the vehicles' distance and thus avoid the bad state; however, their abstraction is too tight to identify an invariant, inducing an infinite sequence of polyhedra along the vehicles' position (Fig. 3b). Our custom template keeps track of vehicles' distance, while abstracting away absolute position; this yields a safe and invariant abstraction of the reach set (Fig. 3c).*

4 Experimental Evaluation

We evaluate our method over multiple agents for 3 benchmark control problems: a bouncing ball, automated cruise control, and cart-pole. We selected a range of loss functions and hyperparameters and verified, using our method, which setups produce safe behaviour. We trained RL agents using proximal policy optimisation (PPO) [Schulman *et al.*, 2017]. We used standard feed forward architectures with 2 hidden layers of size 64 (32 for the bouncing ball), and ReLU activation functions; we used a learning rate of $5e^{-4}$.

We built a prototype[1] and verified the safety of these networks with rectangular and octagonal abstractions and, when necessary, custom templates[2] which we discuss in Sect. 4.2 and 4.3. In addition, we also compared our method with an alternative approach built upon range analysis (Sect. 4.4).

We ran our experiments on a 4-core 4.2GHz with 64GB RAM. Results are shown in Tab. 1 and discussed in Sect. 4.4.

4.1 Bouncing Ball

Environment. The system consists of a ball, whose height from the ground is determined by a variable x and whose vertical velocity is determined by a variable x' [Jaeger *et al.*, 2019]. Under normal conditions, position is given by the equation $x_t = x_{t-1} + \tau \cdot x'_{t-1}$ and velocity is given by $x'_t = x'_{t-1} - \tau \cdot g$, where g denotes gravitational acceleration and $\tau = 0.1$ indicates our time step. Every time the ball hits the ground, i.e., $x_{t-1} \leq 0$, the ball bounces back after losing 10% of its energy, i.e., $x'_t = -0.9 \cdot x'_{t-1} - \tau \cdot g$ and $x_t = 0$. At every timestep, the agent can either hit the ball downward by adding $-4\,\mathrm{m\,s^{-1}}$ to its velocity, or do nothing. Overall, this results in a piecewise linear system.

Training. The goal is to ensure that the ball keeps bouncing indefinitely, while using the piston as little as possible. We reward the agent with value 1 for each time step that the ball's absolute velocity is above the minimal velocity of $1\,\mathrm{m\,s^{-1}}$. Additionally, we discourage the agent from overactivating the piston by punishing it with reward -1 every time it is activated. We trained 11 agents using different initial seeds and with episodes of at most 1000 steps, after which we forcefully terminate. We terminate training either when our agent reaches a mean reward of 900 or after 5M training steps.

Verification. As initial condition, we consider the set of initial ball heights $x_0 \in [7, 10]$ and initial velocities $x'_0 \in [0, 0.1]$. We use traditional rectangular and octagonal abstractions, that is, for rectangles we use the directions $x, -x, x', -x'$ and for octagons add the extra directions $x+x', -x+x', x-x', -x-x'$. Notably, all agents have been successfully verified with both rectangles and octagons with no notable difference in performance.

4.2 Adaptive Cruise Control

Environment. The problem consists of two vehicles, lead and ego, whose state is determined by variables x_v, x'_v and x''_v, respectively, for position, speed, and acceleration of $v =$ ego, lead (see Ex. 1). The lead car proceeds at constant speed ($28\,\mathrm{m\,s^{-1}}$), and the agent controls the acceleration ($\pm\,1\,\mathrm{m\,s^{-2}}$) of ego using either of two actions. Its dynamics are given by

$$x_{v,t} = x_{v,t-1} + \tau \cdot x'_{v,t-1} \qquad \text{for } v = \text{ego, lead}, \quad (19)$$

$$x'_{\text{ego},t} = x_{\text{ego},t-1} + \tau \cdot x''_{\text{ego},t-1} \qquad x'_{\text{lead},t} = 28, \quad (20)$$

$$x''_{\text{ego},t} = \begin{cases} -1 & \text{if } a_{t-1} = 1, \\ 1 & \text{if } a_{t-1} = 2. \end{cases} \quad (21)$$

The observation function exposes vehicle distance y_{dis} and the velocity of ego y_{vel} with an additional observation disturbance of radius ϵ, determined by the following equations:

$$y_{\text{dis},t} = x_{\text{lead},t} - x_{\text{ego},t} + d_{\text{dis},t}, d_{\text{dis},t} \in [-\epsilon, +\epsilon] \quad (22)$$

$$y_{\text{vel},t} = x'_{\text{ego},t} + d_{\text{vel},t}, d_{\text{vel},t} \in [-\epsilon, +\epsilon]. \quad (23)$$

We consider a case with $\epsilon = 0$ and another case with $\epsilon = 0.05$, and use $\tau = 0.1$. Altogether, when an action is given this is a linear system with disturbances.

Training. We train our agents using two 2 different reward functions. A "simple" function only rewards the agent for each timestep it survives without crashing, that is, $R(y_{\text{dis}}, y_{\text{vel}}) = 1$; a "complex" function additionally punishes the agent from being away from a predefined distance $y^\star_{\text{dis}}$, specifically, $R(y_{\text{dis}}, y_{\text{vel}}) = 1 - 0.02 \cdot (y_{\text{dis}} - y^\star_{\text{dis}})^2$. We cap each episode at 1000 timesteps. From the definition of the simple cost function above, we can periodically pre-test the safety of the agent by disabling the exploration and requiring an average score of 1000 before attempting the verification step. For the complex cost function it is more difficult to estimate what a safe score should be, so we empirically determined that before attempting to verify the neural network, the agent needs to reach an average score of at least -20. As an additional stopping condition we terminate the training after 20M training steps. We ran our algorithm over 22 agents trained with different initialisation seeds and two modes of input perturbation ($\epsilon = 0$ and $\epsilon = 0.05$) for up to 300 seconds.

Verification. We consider the initial region enclosed within the constraints $x_{\text{lead},0} \in [40, 50]$, $x_{\text{ego},0} \in [0, 10]$, $x'_{\text{ego},0} = 36$. Using standard rectangular or octagonal abstractions that verification procedure fails by either returning a spurious counterexamples or timing out. To effectively verify this systems, we designed a template with the following directions: x''_{ego}, $-x''_{\text{ego}}$, x'_{lead}, $-x'_{\text{lead}}$, $(x'_{\text{lead}} - x'_{\text{ego}})$, $-(x'_{\text{lead}} - x'_{\text{ego}})$, $(x_{\text{lead}} - x_{\text{ego}})$, $-(x_{\text{lead}} - x_{\text{ego}})$. This allows us to keep track of the distance between the two vehicles and easily spot if the agent encounters an unsafe state, while enabling the identification of an invariant. Agents could be proven to be safe in most but not all of the cases within our time constraints (300s), showing a higher degree of difficulty compared to the previous problem. When testing the agents on the perturbed environment, only a few of the agents that were proven safe in the previous experiment retained safety, demonstrating that the problem the agent had to solve is much harder.

4.3 Cart-pole

Environment. The cart-pole problem is a very well known control problem in the RL literature; for our experiments, we refer to the OpenAI Gym implementation of CartPole-v1 [Brockman *et al.*, 2016]. The state variables are angle θ

[1] https://github.com/phate09/SafeRL_Infinity

[2] For readability, we present direction δ by displaying $\langle \delta, x \rangle$.

and angular velocity θ' of the pole, together with horizontal position x and velocity x' of the cart. The agent has two actions for pushing the cart to either the left or the right which, together with θ and θ', internally determine horizontal and angular accelerations x'' and θ''. The values of θ'' and x'' are determined according to non-linear equations defined with transcendental functions and whose arguments are the action and variables θ and θ'. The update rule for angle θ, position x, and velocities θ' and x' follow a linear Euler integration rule with timestep τ. All variables x, x', θ, and θ' are observable.

Training. The objective for an agent is to keep the pole upright; we consider the system unsafe whenever $\theta > 12°$, according to the OpenAI Gym termination condition. We train agents using three cost functions. A "simple" version only rewards the agent for surviving; a "complex" version discourages it from having high values of θ and θ', i.e., $R(x, x', \theta, \theta') = 1 - 0.5 * \theta^2 - 0.5 * (\theta')^2 - 0.1 * (x')^2$; a third one limits the complex cost function to only giving positive rewards. For every cost function, we trained two agents using $\tau = 0.02$ and $\tau = 0.001$, thus obtaining 6 agents. We capped each episode to 8000 timesteps. For all cost functions, we terminate training when the mean reward of the last 50 episodes reaches 7950 (i.e., sufficiently close to the maximum of 8000) or after 20M training steps. We use curriculum learning to improve training: when the mean episode return reaches 6500 the initial states get sampled from bigger intervals with $\theta \in [-0.2, 0.2]$ and $\theta' \in [-0.5, 0.5]$.

Verification. We use the starting region of OpenAI Gym, i.e., all variables are initialised from the interval $[-0.05, 0.05]$. However, we remove the constraints imposed on x and let the cart-pole move freely to any position. Rectangles and octagons failed to prove safety on all instances. Thus, we designed a custom template that forms an octagon over θ and θ' only, determined by the following directions: θ, $-\theta$, θ', $-\theta'$, $\theta + \theta'$, $-\theta + \theta'$, $\theta - \theta'$, $-\theta - \theta'$. This template choice serves two purposes. First, it ignores position x thus abstracting its values away. Second, it bounds the space of θ and θ', which are the variables appearing in the non-linear equations of the system. This lets us use interval arithmetic for encoding the non-linear equations in our MILP (see Sect. 3). We verified our agents against both versions of the environment, with $\tau = 0.02$ and with $\tau = 0.001$ to test the how it would affect safety. The agents that did run on environments at $\tau = 0.001$ during the evaluation found an invariant quicker and in less timesteps.

4.4 Results

Table 1 reports number of solved instances, average timestep of invariant detection, number of template polyhedra in the abstract reach set after pruning redundant ones, and runtime.

The time required to find whether the agent is safe increases as the number of variables in the problem increases (BB has 2, ACC has 6 and CP has 4) and on the type of abstraction. Templates enable us to find invariants on problems that would not converge otherwise (ACC and CP). Once we introduce a small observation perturbation on ACC such as in adversarial examples, only a small fraction of the agents remain safe negatively impacting safety.

From our results, the cost function used does not strongly correlate with the safety of the agent hence it is omitted in the table. Conversely, shorter timesteps contribute positively to reducing the time required to verify an agent, promoting a higher chance to find a safe invariant in early timesteps.

Env.	Abs.	Safe	Avg k	Avg poly.	Avg runtime
BB	Rect	11/11	237	477	40s
BB	Oct	11/11	203	411	47s
ACC ($\epsilon = 0$)	Temp	20/22	467	610	171s
ACC ($\epsilon = .05$)	Temp	5/22	226	337	124s
CP ($\tau = .001$)	Temp	4/6	27	18	67s
CP ($\tau = .02$)	Temp	3/6	100	125	174s

Table 1: Verification results by environment, i.e, bouncing ball (BB), adaptive cruise control (ACC) and cart-pole (CP), hyperparameters ϵ and τ (where they apply), abstraction, i.e., rectangular, octagonal, or template-based, and number of agents determined to be safe within 300s. For successful outcomes, we report average timestep of fixpoint detection k, number of final template polyhedra, and runtime.

Additionally, we verified our agents using a naive time-unbounded approach (based on range analysis) that constructs, from the network in isolation, ranges of observables for which an action is enabled; then, it uses these ranges as guards for the dynamical system. This alternative approach failed on all instances by producing inconclusive answers (unsafe abstractions) or reaching time-out. Notably, existing verification methods for neural networks are incomparable as they only support time-bounded problems such as robustness to adversarial attacks or finite-horizon safety analysis of RL [Tran *et al.*, 2020; Gehr *et al.*, 2018].

5 Conclusion

We presented the first method for verifying the safety of RL agents up to infinite time. To this end, our method constructs coarse, yet precise enough, abstractions using template polyhedra. We demonstrated the efficacy of our method over multiple case studies. Our technique yields stronger formal guarantees than previous time-bounded methods, and also indicates which RL setups generalise well beyond the length of their training episodes. Our result poses the basis for future research, both in machine learning and formal verification. Our method can be used to make informed decisions about architectures and hyperparameters, and also to guide an RL procedure that trains for safety. Also, our method lends itself to extensions towards multi-agent systems, systems over continuous time, continuous actions and automated abstraction refinement.

Acknowledgments

We thank Hosein Hasanbeig and the anonymous reviewers for their suggestions. This work was in part supported by the HICLASS project (113213), a partnership between Aerospace Technology Institute (ATI), Department for Business, Energy & Industrial Strategy (BEIS) and Innovate UK, and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 834115, FUN2MODEL).

References

[Alshiekh *et al.*, 2018] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *AAAI*, pages 2669–2678. AAAI Press, 2018.

[Bacci and Parker, 2020] Edoardo Bacci and David Parker. Probabilistic guarantees for safe deep reinforcement learning. In *FORMATS*, pages 231–248. Springer, 2020.

[Bogomolov *et al.*, 2017] Sergiy Bogomolov, Goran Frehse, Mirco Giacobbe, and Thomas A. Henzinger. Counterexample-guided refinement of template polyhedra. In *TACAS (1)*, pages 589–606, 2017.

[Brockman *et al.*, 2016] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI gym. *arXiv:1606.01540*, 2016.

[Bunel *et al.*, 2018] Rudy Bunel, Ilker Turkaslan, Philip H. S. Torr, Pushmeet Kohli, and Pawan Kumar Mudigonda. A unified view of piecewise linear neural network verification. In *NeurIPS*, pages 4795–4804, 2018.

[Cheng *et al.*, 2019] Richard Cheng, Gábor Orosz, Richard M. Murray, and Joel W. Burdick. End-to-end safe reinforcement learning through barrier functions for safety-critical continuous control tasks. In *AAAI*, pages 3387–3395. AAAI Press, 2019.

[Desjardins and Chaib-draa, 2011] Charles Desjardins and Brahim Chaib-draa. Cooperative adaptive cruise control: A reinforcement learning approach. *IEEE Trans. Intell. Transp. Syst.*, 12(4):1248–1260, 2011.

[Dutta *et al.*, 2018] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Output range analysis for deep feedforward neural networks. In *NFM*, pages 121–138. Springer, 2018.

[Ehlers, 2017] Rüdiger Ehlers. Formal verification of piecewise linear feed-forward neural networks. In *ATVA*, pages 269–286. Springer, 2017.

[Frehse *et al.*, 2018] Goran Frehse, Mirco Giacobbe, and Thomas A. Henzinger. Space-time interpolants. In *CAV (1)*, pages 468–486. Springer, 2018.

[Fulton and Platzer, 2018] Nathan Fulton and André Platzer. Safe reinforcement learning via formal methods: Toward safe control through proof and learning. In *AAAI*, pages 6485–6492. AAAI Press, 2018.

[Gehr *et al.*, 2018] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. AI2: safety and robustness certification of neural networks with abstract interpretation. In *IEEE SP*, pages 3–18. IEEE Computer Society, 2018.

[Hasanbeig *et al.*, 2019] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. Logically-constrained neural fitted q-iteration. In *AAMAS*, pages 2012–2014. IFAAMAS, 2019.

[Huang *et al.*, 2017] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *CAV (1)*, pages 3–29. Springer, 2017.

[Jaeger *et al.*, 2019] Manfred Jaeger, Peter Gjøl Jensen, Kim Guldstrand Larsen, Axel Legay, Sean Sedwards, and Jakob Haahr Taankvist. Teaching stratego to play ball: Optimal synthesis for continuous space mdps. In *ATVA*, pages 81–97. Springer, 2019.

[Katz *et al.*, 2019] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. The marabou framework for verification and analysis of deep neural networks. In *CAV (1)*, pages 443–452. Springer, 2019.

[Kendall *et al.*, 2019] Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Bewley, and Amar Shah. Learning to drive in a day. In *ICRA*, pages 8248–8254. IEEE, 2019.

[Li and Bastani, 2020] Shuo Li and Osbert Bastani. Robust model predictive shielding for safe reinforcement learning with stochastic dynamics. In *ICRA*, pages 7166–7172. IEEE, 2020.

[Luckcuck *et al.*, 2019] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. Formal specification and verification of autonomous robotic systems: A survey. *ACM Comput. Surv.*, 52(5):100:1–100:41, 2019.

[Moosavi-Dezfooli *et al.*, 2016] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: A simple and accurate method to fool deep neural networks. In *CVPR*, pages 2574–2582. IEEE Computer Society, 2016.

[Sankaranarayanan *et al.*, 2005] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, pages 25–41. Springer, 2005.

[Schulman *et al.*, 2017] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv:1707.06347*, 2017.

[Singh *et al.*, 2019] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin T. Vechev. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.*, 3(POPL):41:1–41:30, 2019.

[Tjeng *et al.*, 2019] Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *ICLR (Poster)*. OpenReview.net, 2019.

[Tran *et al.*, 2020] Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T. Johnson. NNV: the neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In *CAV (1)*, pages 3–17. Springer, 2020.

[Xiang *et al.*, 2018] Weiming Xiang, Hoang-Dung Tran, and Taylor T. Johnson. Output reachable set estimation and verification for multilayer neural networks. *IEEE Trans. Neural Networks Learn. Syst.*, 29(11):5777–5783, 2018.