Polytopic Trees for Verification of Learning-Based Controllers

Osbert Bastani[1], Sadra Sadraddini[2], Shen Shen[2]

University of Pennsylvania[1], Massachusetts Institute of Technology[2],
obastani@seas.upenn.edu, sadra,shenshen,russt@mit.edu

Abstract. Reinforcement learning is increasingly used to synthesize controllers for a broad range of applications. However, formal guarantees on the behavior of learning-based controllers are elusive due to the black-box nature of machine learning models such as neural networks. In this paper, we propose an algorithm for verifying learning-based controllers—in particular, deep neural networks with ReLU activations, and decision trees with linear decisions and leaf values—for deterministic, piecewise affine (PWA) dynamical systems. In this setting, our algorithm computes the safe (resp., unsafe) region of the state space—i.e., the region of the state space on which the learned controller is guaranteed to satisfy (resp., fail to satisfy) a given reach-avoid specification. Knowing the safe and unsafe regions is substantially more informative than the boolean characterization of safety (i.e., safe or unsafe) provided by standard verification algorithms—for example, this knowledge can be used to compose controllers that are safe on different portions of the state space. At a high level, our algorithm uses convex programming to iteratively compute new regions (in the form of polytopes) that are guaranteed to be entirely safe or entirely unsafe. Then, it connects these polytopic regions together in a tree-like fashion. We conclude with an illustrative example on controlling a hybrid model of a contact-based robotics problem.

1 Introduction

Recently, there has been a great deal of success using reinforcement learning to synthesize controllers for challenging control tasks, including grasping [20], autonomous driving [24], and walking [9]. Reinforcement learning provides a number of advantages compared to traditional approaches to control—for example, it can be used to compress computationally expensive online controllers into computationally efficient control policies [21], it can be used to solve challenging nonconvex optimization problems such as grasping [4], and it can be used to adapt controllers to handle unmodeled real-world dynamics [1, 9].

Despite these successes, reinforcement learning has had limited applicability in real-world control tasks. An important obstacle is the inability to provide formal guarantees on the behavior of learned controllers. For instance, real-world control tasks are usually safety-critical in nature. Furthermore, the more general problem of computing safe regions (i.e., sets of states from which the controller is guaranteed to be safe) is also an important tool for composing controllers [32].

The challenge is that learned controllers are typically deep neural networks (DNNs), which are hard to formally analyze due to their large number of parameters and lack of internal structure [5,16]. The lack of guarantees is particularly concerning since even in the supervised learning setting, DNNs are not robust [5,11,14,16,25,29,31,34,36]—i.e., even if an input is correctly classified, a small, adversarially chosen perturbation can typically cause the input to become misclassified. For closed-loop control, the lack of robustness would imply that even if a DNN controller produces a safe trajectory from a given initial state, a small perturbation to the initial state can cause the trajectory to become unsafe. This setting is particularly challenging for formal analysis, since we must reason about repeated application of the DNN controller.

In this paper, we propose an algorithm for computing safe regions (as well as unsafe regions—i.e., for which the system is guaranteed to be unsafe) for learning-based controllers. We are interested in *reach-avoid* specifications, which are safety properties expressed as (i) a goal region that is considered to be safe, (ii) a safety constraint specifying states that are known to be unsafe. Then, our goal is to classify each state as either safe (i.e., reaches the goal region without entering the unsafe region), or unsafe (i.e., it reaches the unsafe region). For example, consider the task of stabilizing an inverted pendulum. Then, the goal region may be a small region $|\theta| \leq \theta_{\text{safe}}$ around the upright position $\theta = 0$, where the closed-loop system is known to be safe (e.g., verified using stability analysis [32]), and the safety constraint may say that the pendulum should not fall over—i.e., $|\theta| > \theta_{\text{unsafe}}$, where $\theta_{\text{unsafe}} \in \mathbb{R}$. Our goal is to compute the set of states from which the learning-based controller successfully reaches the goal region without entering the unsafe region.

We build on recent work that verify DNNs with rectified-linear units (ReLUs), which are piecewise affine [5,16]. As a consequence, for the supervised learning setting, safety properties can then be encoded as a mixed-integer linear program [33], which can be checked using standard solvers. For our setting of closed-loop control, we consider dynamical systems with piecewise affine (PWA) dynamics, where each piece is a *polytopic region* (i.e., defined by the intersection of linear half-spaces). These systems are commonly encountered in control tasks; even systems that are not PWA can typically be (locally) closely approximated by one that is PWA. For PWA systems, safety properties for the closed-loop dynamics can be formulated as a mixed-integer linear program [15]. However, this approach can only be used to verify whether a given set of initial states is a safe region, and cannot be used to compute safe and unsafe regions.

At a high level, our algorithm iteratively constructs the safe and unsafe regions as follows. On each iteration, it first samples an unclassified state x from the state space, and determines whether x is safe or unsafe using a simple forward simulation. In particular, x is safe if the feedforward simulation reaches the current safe region, and unsafe if it reaches the current unsafe region. Finally, our algorithm expands x into a polytopic region around x with the same correctness property as x (i.e., safe or unsafe). Thus, our algorithm is essentially growing two tree-like data structures (which we call *polytopic trees*)—one representing safe states one representing unsafe states. Then, our algorithm continues iteratively growing these polytopic trees until the entire state space has been classified.

The key challenge is expanding x into a polytopic region. We leverage the fact that the closed-loop dynamics is PWA—in particular, it computes a polytopic region around x such that the closed-loop dynamics is linear on that region. Then, it uses convex programming to restrict this region to a polytopic sub-region that has the same correctness property (i.e., safe or unsafe) as x. In summary, our contributions are:

– We formulate the problem of computing safe and unsafe regions for learning-based controllers (Section 2).
– We propose an algorithm for computing safe and unsafe regions (Section 5). Our algorithm uses a subroutine that computes safe and unsafe polytopic regions around a given initial state for the closed-loop dynamical system (Sections 4 & 3).
– We perform an extensive case study to evaluate our approach (Section 6).

Related work. The closest work is [30], which studies the problem of finding the safe states of a linear systems controlled with ReLU-based DNNs. Their solution is based on partitioning the work-space and computing reachable sets to obtain a finite-state abstraction. The end result is an under-approximation of the the safe region. This approach suffers from the resolution quality of the workspace partitioning, which is known to scale badly in high dimensions. Our approach does not require partitioning, which helps the algorithm scale better. Moreover, in high dimensions, it is common to only verify a desirable part of the state space, where sampling-based strategies are naturally powerful. On the other hand, fine partitioning of the whole state-space is still necessary to obtain a finite abstraction that represents the original system by the needed accuracy.

Another line of work in formal verification of learning-based controllers is searching for counterexamples within a given region. For example, [6] takes this approach to verify controllers obtained using reinforcement learning. If no counterexample is found, then the whole given region is verified as safe. The counterexample search can be formulated as an SMT problem [10]. While this approach allows for one shot verification of the specification over a given region, it does not allow for direct computation of the safe region. In many applications in robot control, counterexamples are abundant, and safe regions are complex subsets of the state space. Therefore, this approach to verification has limited applicability. Similarly, [15] provides a boolean answer to the verification problem by checking a property over a given region. The method is based on formulating the verification problem as the reachability problem of a hybrid system, which is verified using state-of-the art solvers such as dReach [17] and Flow* [8]. They also do not provide a direct way to compute safe and unsafe regions.

The authors in [37] and [38] explicitly compute the forward reachable sets of DNN-based controllers. The exact computations are formidable, so they use over-approximations. Our work also computes reachable sets, albeit in a backward manner. We efficiently grow the backward reachable set using trees, thus eliminating the need for roll-out of reachable sets for a large horizon. Moreover, we are able to provide probabilistic completeness properties, whereas methods based on over-approximations are inherently conservative.

2 Problem Statement and Approach

Notation. We denote the sets of real and non-negative real numbers by $\mathbb{R}$ and $\mathbb{R}_+$, respectively. Given a set $\mathbb{S} \subseteq \mathbb{R}^n$, we use $\text{int}(\mathbb{S})$ to denote the interior of $\mathbb{S}$. Given $\mathbb{S} \subset \mathbb{R}^n$ and $A \in \mathbb{R}^{n_A \times n}$, we use $A\mathbb{S}$ to denote the set $\{As \mid s \in \mathbb{S}\}$. For a vector $x \in \mathbb{R}^n$, we use $|x|$ to denote the cardinality of the vector, i.e., $|x| = n$. All matrix inequality relations are interpreted element-wise. A *polyhedron* $\mathbb{H} \subset \mathbb{R}^n$ is the intersection of a finite number of closed half-spaces in the form $\mathbb{H} = \{x \in \mathbb{R}^n \mid Hx \leq h\}$, where $H \in \mathbb{R}^{n_H \times n}, h \in \mathbb{R}^{n_H}$ define the hyperplanes. A bounded polyhedron is called a *polytope.* A *piecewise affine (PWA) function f* is a function $f : \mathbb{H} \to \mathbb{R}^{n_f}$, where the domain $\mathbb{H} \subseteq \mathbb{R}^n$ is a union $\mathbb{H} = \bigcup_{i=1}^{k} \mathbb{H}_i$ of disjoint polyhedra $\mathbb{H}_1, ..., \mathbb{H}_k$, and where for each i, $f(x) = A_i x + b_i$ for all $x \in \mathbb{H}_i$ for some $A_i \in \mathbb{R}^{n \times n_f}$ and $b_i \in \mathbb{R}^{n_f}$.

Problem formulation. Consider a deterministic control system

$$\begin{aligned} x_{t+1} &= F(x_t, u_t) \\ u_t &= g(x_t), \end{aligned}$$

where $x_t \in \mathbb{X} \subseteq \mathbb{R}^n$ is the state, $u_t \in \mathbb{U} \subseteq \mathbb{R}^m$ is the control input at time $t \in \mathbb{N}$, and $F : \mathbb{X} \times \mathbb{U} \to \mathbb{R}^n$ is the system dynamics, and $g : \mathbb{R}^n \to \mathbb{U}$ is the controller. We assume that F and g are both PWA functions. Note that it is possible that there exists $x \in \mathbb{X}, u \in \mathbb{U}$ such that $F(x, u) \notin \mathbb{X}$. One of the primary tasks of the controller is to keep the state within the constraint set $\mathbb{X}$. Also, we assume that $\mathbb{X}$ is bounded. Note that one or both of F and g may be machine learning components with PWA structure. Examples include feed-forward neural networks with ReLU activation functions, and decision trees with linear predicates. The closed-loop system is given as:

$$x_{t+1} = F(x_t, g(x_t)) = F_{cl}(x_t), \tag{1}$$

where F_{cl} itself is a PWA system:

$$F_{cl}(x) = A_i x + b_i, \qquad \forall x \in \mathbb{C}_i, \tag{2}$$

where $\mathbb{C}_i, i = 1, \cdots, N$, are interior-disjoint polytopes, $\bigcup_{i=1}^{N} \mathbb{C}_i = \mathbb{X}$. The number of pieces N is dependent on the structure of F and g, but upper-bounded by the number pieces in F times the number pieces in g.

Problem 1. Given closed-loop system of the form (1), and a goal region $X_G \subseteq \mathbb{X}$, compute the two followings sets:

- $X_s \subseteq \mathbb{X}$, where all trajectories originating from it reach X_G in finite time (the *safe region*).

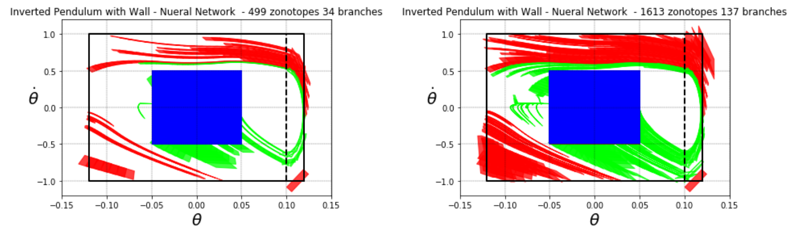
– $X_f \subseteq \mathbb{X}$, where all trajectories originating from it leave $\mathbb{X}$ in finite time (the *unsafe region*).

We assume that X and X_G are both given as a union of finite number of polytopes. This assumption is common in most control problems as any bounded set can be reasonably approximated using a finite number of polytopes.

Approach. The complete solution to Problem 1 partitions $X \setminus X_G$ into 3 regions: $X_s^{\max}$, the largest possible set of X_s, $X_f^{\max}$, the largest possible set of X_f, and $X_a^{\min} := X \setminus (X_s^{\max} \cup X_f^{\max})$, where $X_a^{\min}$ is the set of states from which originating trajectories neither reach the goal or violate the constraints in finite time. Therefore, $X_a^{\min}$ is a forward-invariant set. Although in most control tasks the forward-invariant sets around an equilibrium point or a limit cycle are desirable and the user typically designs X_G within it, it is possible that learning-based controllers inadvertently create invariant sets outside of X_G, where trajectories are trapped and thus is an undesirable behavior—examples of this behavior in DNN-based controllers is shown in Section 6.

Our solution to Problem 1 is an anytime sampling-based algorithm that gradually grows X_s and X_f, and shrinks X_a. Initially, $X_s = X_G, X_f = \emptyset$, and $X_a = X \setminus X_G$. As shown in the paper, our algorithm has probabilistic completeness: if $x \in X_s^{\max}$ or $x \in X_f^{\max}$, the probability that our solution verifies it as the algorithm runs approaches one.

Example. As an example of the safe and unsafe regions computed using our algorithm, consider the following figures:



These figures plot the safe region X_s (green), unsafe region X_f (red), and unclassified region X_a (white) for a neural network controller trained to stabilize a torque-limited inverted pendulum, after 34 iterations (left) and 137 iterations (right) of our algorithm. This task is complicated by the presence of a wall, which the controller can use to "bounce" off of to aid stabilization. The presence of the contact dynamics makes the problem challenging for traditional control synthesis algorithms. On the other hand, reinforcement learning can be directly applied to solve this problem. The blue region is the goal region X_G, where the linear controller synthesized using LQR on the linearized dynamics can provably stabilize the inverted pendulum. Reinforcement learning is used to train a neural network or decision tree controller with the goal of driving the system into the blue region X_G. The region inside the black square is the constraint set $\mathbb{X}$.

At each iteration, our algorithm grows either the safe region or the unsafe region from a randomly chosen state $x \in X_a$. Some of the regions added on each iteration can be distinguished in the figure above—they tend to be elongated subsets of the state space, since they are computed by expanding a trajectory (which has measure zero) into a small region around that trajectory (which has positive measure). Comparing the figure on the two figures, we can also see how safe and unsafe regions are added over time. As can be seen, X_s tends to grow outward from X_G as the number of iterations increases. Similarly, X_f tends to grow inward from the region $\mathbb{R}^2 \setminus \mathbb{X}$. These patterns reflect the tree-like way in which the safe and unsafe regions are grown.

3 Local Constrained Affine Dynamics

We provide a framework to characterize the local affine dynamics of (1) around a given point. Recall that (1) is PWA with affine dynamics in polytopic cells.

Definition 1. *An affine cell is defined as a tuple $\mathcal{A} = (A, b, H, h)$, where $F_{cl}(x) = Ax + b, \forall x \in \{x \in \mathbb{R}^n | Hx \leq h\}$.*

Let the set of all affine cells be defined as $\mathbb{A}$. In this section, we explain how to derive a function $\mathcal{L} : \mathbb{X} \to \mathbb{A}$, which takes a point $x \in X$, and provides the affine cell which it belongs to. The basic idea is to fix (i) the mode of the dynamical system (if it is PWA) and (ii) the activations of the machine learning structure used in the controller. In decision trees, this means the path to the leaf is fixed. In DNNs, it means all the ReLU activations remain the same. Note that there exists a finite number of affine cells in (1). However, computing all of them in advance may not be possible. For decision trees, the number of affine cells is equivalent to the number of leaves, which is often a manageable number. For neural networks, on the other hand, the number of affine cells can be as much as $2^{|\# \text{ neurons}|}$, which can be extremely large to save on a memory. Therefore, it is better that $\mathcal{L}(x)$ is computed on-the-fly, i.e. while the algorithm is running.

Decision trees. A depth d decision tree τ is a binary tree with $2^d - 1$ internal nodes and 2^d leaf nodes (we ignore leaf nodes when measuring depth). Each internal node of the tree is labeled with tuple (i, t), where $i \in \{1, ..., n\}$ (where n is the dimension of the state space) and $t \in \mathbb{R}$; we interpret this tuple as a predicate $x_i \leq t$. Each leaf node is labeled with a control input $u \in \mathbb{U}$.

We let $\mathsf{root}(\tau)$ denote the root of τ, and $\mathsf{leaves}(\tau)$ denote the leaf nodes of τ. For each leaf node ν in τ, we let $\mathsf{path}(\nu; \tau) = ((\nu_1, s_1), ..., (\nu_d, s_d))$ denote the sequence of internal nodes on the path from $\mathsf{root}(\tau)$ to ν (so $\nu_1 = \mathsf{root}(\tau)$ and ν_d is the parent of ν in τ). The variable $s_i \in \{\pm 1\}$ is $+1$ if ν_{i+1} is the left child of ν_i, and -1 if it is the right child (where we let $\nu_{d+1} = \nu$). Then, we associate each leaf node ν with the affine cell $\mathcal{A}_\nu = (A_\nu, b_\nu, H_\nu, h_\nu)$, where A_ν is the zero matrix, $b_\nu = u$ (where u is the label on ν), and (H_ν, h_ν) define the polyhedron $\mathbb{H}_\nu$ that is the intersection of the half-spaces

$$\{s \cdot x_i \leq t \mid (\nu', s) \in \mathsf{path}(\nu; \tau) \text{ has label } (i, t)\}. \tag{3}$$

It is clear that the polyhedra $\mathbb{H}_\nu$ associated with the leaf nodes $\nu \in \mathsf{leaves}(\tau)$ are disjoint and cover $\mathbb{R}^n$. Thus, we can interpret τ as a function $\tau : \mathbb{X} \to \mathbb{U}$, where

$$\tau(x) = \sum_{\nu \in \mathsf{leaves}(\tau)} (A_\nu x + b_\nu) \cdot \mathbb{I}[x \in \mathbb{H}_\nu],$$

where $\mathbb{I}$ is the indicator function. Thus, given $x \in \mathbb{X}$, we define $\mathcal{L}(x) = \mathcal{A}_\nu$, where ν is the (unique) leaf node such that $x \in \mathbb{H}_\nu$.

Neural networks. Consider a k-layer ReLU-based network. For a single layer i, let its input be x_i and output be x_{i+1}. Also, let the layer weights be $A_i \in \mathbb{R}^{|x_{i+1}| \times |x_i|}$ and the bias be $b_i \in \mathbb{R}^{|x_{i+1}|}$. The layer i input-output relationship is $x_{i+1} = \max(A_i x_i + b_i, 0)$ where $\max(\cdot)$ is taken element-wise.

To derive the polytopic cell expression, we first introduce binary vector $s_i \in \{0,1\}^{|x_{i+1}|} = S(A_i x_i + b_i)$ where the scalar function $S(\alpha) := \mathbb{I}[\alpha \geq 0]$ is applied element-wise. Therefore, we equivalently have $x_{i+1} = \hat{A}_i x_i + \hat{b}_i$ where $\hat{A}_i = s_i \odot A_i$, $\hat{b}_i = s_i \odot b_i$ and $\odot$ is the element-wise multiplication. By recursive expansion, the network final output x_{k+1} in terms of the first layer input x_1 is

$$x_{k+1} = \underbrace{\prod_{i=1}^{k} \hat{A}_i}_{A} x_1 + \underbrace{\sum_{i=1}^{k-1} \prod_{j=i+1}^{k} \hat{A}_j \hat{b}_i + \hat{b}_k}_{b}, \tag{4}$$

which is PWA with the pieces (defined by polytopes to be derived) dependent on the s_i. Also, since matrix multiplication is not commutative, it is worth pointing out that the enumeration is left-multiplied: $\prod_{i=1}^{k} \hat{A}_i = \hat{A}_k \hat{A}_{k-1} \ldots \hat{A}_1$.

To get the H-representation for the polytopes, we use one single layer with one single ReLU for illustration. In this case, x_2 can take on two possible values depending on if $s_1 = 1$, i.e., $A_1 x_1 + b_1 \geq 0$, or if $s_1 = 0$, i.e. $A_1 x_1 + b_1 < 0$. These two case conditions can be equivalently described as checking if $H_1 x_1 \leq h_1$, with $H_1 := (2s_1 - 1) \odot A_1$, and $h_1 := -(2s_1 - 1) \odot b_1$.

It is then straightforward, albeit tedious, to generalize that for any particular layer i, the hyperplanes defining a particular polytope is

$$H_i = (2s_i - e) A_i \prod_{j=1}^{i-1} \hat{A}_j, h_i = (2s_i - e) b_i + H_i (\sum_{l=1}^{i-1} \prod_{j=l+1}^{i-2} \hat{A}_j \hat{b}_l + \hat{b}_i) \tag{5}$$

where e is the all-one vector, such that $(2s_1 - e) \in \{-1, 1\}^{|x_{i+1}|}$ and easier for sign-flipping. For the entire network, the affine cell of a particular input x_1 is then $\mathcal{A}(x_1) = (A, b, H, h)$ with A, b as defined in (4), and H and h as simply the column concatenation of the individual H_i and h_i in (5).

Affine cells of the closed-loop system. Once given a state query $x_q \in X$ and obtained affine cell for the controller $g(x) = A_u x + b_u, \forall x \in \mathbb{C}_u, x_q \in \mathbb{C}_u$, we combine it with $F(x, u) = A_F x + b_F u + c_F, \forall x \in \mathbb{C}_F, x_q \in \mathbb{C}_F$ to derive the affine cell of the closed-loop dynamics:

$$F_{cl}(x) = (A_F + b_F A_u) x + (b_F b_u + c_F), \forall x \in \mathbb{C}_u \cap \mathbb{C}_F. \tag{6}$$

4 Polytopic Trajectories

In this section we introduce *polytopic trajectories*, which is central to our verification approach. Recall that by simulating the system forward we obtain a trajectory that is a region with zero measure. A key property of this trajectory is that the points in the trajectory are either all safe or all unsafe (since the dynamics and controller are deterministic). Our algorithm expands this trajectory into a region with positive measure, such that the points in the region are either all safe or all unsafe. Doing so enables our algorithm to verify a nonzero fraction of the state space at each iteration. To obtain such a region, our algorithm expands each point in this trajectory into a polytopic region, resulting in a trajectory of polytopic regions, which we refer to as a polytopic trajectory.

Parametrization. First, we define the space of all polytopic trajectories. A polytopic trajectory is a sequence of polytopes $\mathbb{P}_t, t = 0, 1, \cdots, T$, with the constraint that $\mathbb{P}_t \subseteq \mathbb{X}$ is mapped to $\mathbb{P}_{t+1} \subseteq \mathbb{X}$ by the closed-loop dynamics. As standard, the polytopes are individually parameterized as follows:

$$\mathbb{P}_t = \bar{x}_t + G_t \mathbb{P}_b, \tag{7}$$

where $\mathbb{P}_b \subseteq \mathbb{R}^q$ is a user-defined base polytope, and $\bar{x}_t \in \mathbb{R}^n, G_t \in \mathbb{R}^{n \times q}$ are parameters characterizing $\mathbb{P}_t$ as an affine transformation of $\mathbb{P}_b$. The matrix G_t is called the *generator*. When $\mathbb{P}_b$ is chosen to be the unit hypercube in $\mathbb{R}^q$, the polytopes are referred to as *zonotopes*. Due to their balance between expressiveness and computational efficiency, zonotopes are a popular way to parameterize polytopes when verifying dynamical systems [2,12]. The remaining parameter q is chosen by the user. A typical choice is $q = n$ to obtain zonotopes of order one.

Next, we describe how our algorithm enforces the constraint on $\mathbb{P}_t$ and $\mathbb{P}_{t+1}$—i.e., that the closed-loop dynamics maps $\mathbb{P}_t$ to $\mathbb{P}_{t+1}$. If we restrict $\mathbb{P}_t$ to entirely lie in an affine cell of (1), then $\mathbb{P}_t$ will be subject to an affine transformation. In this case, $\mathbb{P}_{t+1}$ will also be a polytope. However, if $\mathbb{P}_t$ has points in multiple cells of (1), then its image under the closed-loop dynamics would be a union of polytopes (but may not itself be a polytope); the number of polytopes in this union may grow exponentially in the number of time steps. Therefore, we enforce a constraint that each polytope is contained within a single affine cell. In particular, letting the affine dynamics be $x_{t+1} = A_t x_t + b_t$, we have

$$\mathbb{P}_{t+1} = A_t \bar{x}_t + b_t + A_t G_t \mathbb{P}_b. \tag{8}$$

Therefore, we obtain linear relations for the parameters of the polytopes:

$$\bar{x}_{t+1} = A_t \bar{x}_t + b_t, G_{t+1} = A_t G_t. \tag{9}$$

Finally, we remark that in many control problems, the underlying system evolves in continuous-time. Therefore, it is preferable to include the states between $\mathbb{P}_t$ and $\mathbb{P}_{t+1}$ in the polytopic region, since those states are traversed between two polytopes. One reasonable approximation is computing the convex hull of $\mathbb{P}_t$ and $\mathbb{P}_{t+1}$, but this approximation is computationally demanding. An alternative is adding $x_{t+1} - x_t$ as an additional column to G_t, which elongates the zonotope alongside the trajectory path. Due to its simplicity, we use the latter method.

Optimization. Next, we describe how our algorithm computes the parameters $\{(\bar{x}_t, G_t)\}_{t=0,\cdots,T-1}$ of a polytopic trajectory that satisfies the property that the states in the polytopic trajectory are either all safe or all unsafe. Our algorithm uses a convex program to do so. Let the sampled trajectory be $x_0, x_1, \cdots, x_T$, where x_T is already known to be safe or unsafe. Suppose that $x_T \in \mathbb{P}_{\text{target}}$, where the *target polytope* $\mathbb{P}_{\text{target}}$ is a polytope around x_T that is known to be safe or unsafe. The existence of such a polytope is guaranteed since our algorithm keeps track of safe and unsafe regions as a union of polytopes—in particular, the target polytope may be one of the following regions: X_G, $\mathbb{R}^n \setminus X_G$, or a polytope in X_s or X_f. Details are in Section 5; for now, we suppose that $\mathbb{P}_{\text{target}}$ is given.

We compute a polytopic trajectory using the following optimization problem:

$$\begin{aligned} \{(\bar{x}_t, G_t)\}_{t=0,\cdots,T-1} = \arg\min \quad & \alpha\left(\{(\bar{x}_t, G_t)\}_{t=0,\cdots,T-1}\right) \\ \text{subj. to} \quad & G_{t+1} = A_t G_t, \bar{x}_{t+1} = A_t \bar{x}_t + b_t, \bar{x}_0 = x_0, \\ & \bar{x}_t + G_t \mathbb{P}_b \subseteq \mathbb{C}_t, \bar{x}_T + G_T \mathbb{P}_b \subseteq \mathbb{P}_{\text{target}}, \\ & (A_t, b_t, \mathbb{C}_t) = \mathcal{L}(x_t), t \in \{0, \cdots, T-1\}. \end{aligned} \tag{10}$$

where $\alpha : \prod_{i=0}^{T-1} \mathbb{R}^n \times \mathbb{R}^{n \times q} \to \mathbb{R}$ is a cost function that is user-defined. The first line of constraints in (10) encode the closed-loop dynamics $\mathbb{P}_t$ to $\mathbb{P}_{t+1}$ and the initial state. The second line is polytope containment constraints that ensure that the whole $\mathbb{P}_t$ is subject to a single affine dynamics and the final polytope is contained in the target polytope $\mathbb{P}_{\text{target}}$. The details on how to encode polytope containment problems into a set of linear constraints using auxiliary variables and constraints is available in [27]. Thus, all the constraints in (10) are linear.

We wish to design α to promote larger volumes for the polytopes. The volume of zonotopes is generally a non-convex objective [18]. Thus we use heuristics for designing α. A useful, simple, form of α is

$$\alpha = \operatorname{Tr}\begin{pmatrix} G_0 & 0 \\ 0 & G_0^T \end{pmatrix}. \tag{11}$$

Note that if $\mathbb{P}_b$ is the unit hypercube, it follows from symmetry that restricting the diagonal terms in the matrix in (11) to be non-negative does not have any effect on $\mathbb{P}_0$ as zonotopes are invariant with respect to multiplication of the columns of their generator by -1. A notable difference between (10) and computing the region of attraction of LQR controllers in [32] is that in the latter, the verified regions are centered around the nominal trajectory. In contrast, there is no such restriction in (10), so the polytopic trajectory can shift around the nominal trajectory. This fact is important since constructing polytopes centered at the points of the trajectory can lead to very small polytopes if only one time point of the trajectory lies close to the boundary of its affine cell.

Computational complexity. Using (11), the optimization problem in (10) becomes a linear program which its size scales quadratically with n, the dimension of the state, and linearly in length of the trajectory. The polytope containment constraints introduce auxiliary variables whose size grows linearly with n and with the number of rows of the hyperplane, where the latter grows linearly with

Algorithm 1 Construction of Polytopic Verification Trees

```
Require: System (1) and $X_G$                                ▷ The closed loop system and the goal
Require: $i^{\max}, T^{\max}, \mathcal{L} : X \to \mathbb{L}$    ▷ The number of iterations, the maximum time of
  forward simulation, and local constrained affine system generator
  $i = 0, X_s = X_G, X_f = \emptyset$                                        ▷ Initialization
  while $i \le i^{\max}$ do
      $x_0 \leftarrow$ sample from $X \setminus (X_s \cup X_f), t = 0$
      while $t \le T^{\max}$ do
          if $x_t \in \mathbb{P}, \mathbb{P} \subseteq X_s$ then                          ▷ $\mathbb{P}$ is a polytope
              flag $\leftarrow$ safe
              Compute polytopic trajectory $\mathbb{P}_0, \mathbb{P}_1, \cdots, \mathbb{P}_{t-1}$ using (10) with $\mathbb{P}_{\text{target}} = \mathbb{P}$
              Add branch to the tree and add $\mathbb{P}_0, \mathbb{P}_1, \cdots, \mathbb{P}_{t-1}$ to $X_s$
              Break
          if $x_t \in \mathbb{P}, \mathbb{P} \subseteq X_f$ or $x_t \notin \mathbb{X}$ then              ▷ $\mathbb{P}$ is a polytope
              flag $\leftarrow$ unsafe
              Compute polytopic trajectory $\mathbb{P}_0, \mathbb{P}_1, \cdots, \mathbb{P}_{t-1}$ using (10) with $\mathbb{P}_{\text{target}} = \mathbb{P}$
              Add branch to the tree and add $\mathbb{P}_0, \mathbb{P}_1, \cdots, \mathbb{P}_{t-1}$ to $X_f$
              Break
          $x_{t+1} = F_{cl}(x_t)$                                  ▷ Simulate system forward
          $\mathbb{C}_{t+1} = \mathcal{L}(x_{t+1})$                       ▷ Compute the polytypic linear system
  return $X_s, X_f$
```

i) the depth of the decision tree, ii) the number of nodes in the DNN. Therefore, obtaining polytopic trajectories is a very efficient procedure.

5 Polytopic Trees

We describe our polytopic tree algorithm (outlined in Algorithm 1).

5.1 Sampling Unclassified States and Checking Membership

At each iteration, we first sample a point from the *unclassified states* $\mathbb{X} \setminus (X_s \cup X_f)$. A straightforward approach is to use rejection sampling—i.e., sample x from X, and reject it if it belongs to $(X_s \cup X_f)$. However, the subroutine of checking whether $x \in (X_s \cup X_f)$ can be computationally challenging as $(X_s \cup X_f)$ grows with the number of iterations. Recall that both X_s and X_f consist of a finite number of polytopes. There are several approaches to checking if $x \in (X_s \cup X_f)$.

The first approach is to check the feasibility of $x = \bar{x}_i + G_i p$, for $p \in \mathbb{P}_b$ and $i \in \{1, \cdots, N\}$, where $\bar{x}_i, G_i$ are the parameters of the ith polytope in $(X_s \cup X_f)$, and N is the total number of polytopes. This approach requires at most N linear programs, which the size of each is linear in n and q, and very small. The second approach is to precompute the hyperplanes of all polytopes; then, instead of checking feasibility of linear programs, we can evaluate the partial order relation required to check if a point is within the intersection of multiple hyperplanes. However, finding the hyperplanes may also be computationally challenging. Both methods can be greatly accelerated using binary search methods for manipulating a large number of polytopes [35].

5.2 Growing the Polytopic Tree

A *polytopic tree* is a tree-like data structure where each node is labeled with a polytope; in general, these "trees" may have multiple roots. Our algorithm represents the regions X_s and X_f as polytopic trees, where we require that the parent $\mathbb{P}'$ of a polytope $\mathbb{P}$ contains the image of $\mathbb{P}$ under the closed-loop dynamics. The regions X_s and X_f are simply the union of all polytopes in the tree. Initially, we represent $X_s = X_G$ as the "tree" where each polytope in X_G is its own root, and represent $X_f = \emptyset$ as the empty tree.

We iteratively grow the polytopic trees representing X_s and X_f by sampling an unclassified state x_0 from $\mathbb{X} \setminus (X_s \cup X_f)$, forward simulating the system from x_0 to obtain a trajectory, and then expanding the resulting trajectory into a polytopic trajectory. More precisely, during forward simulation, once we have state x_t of the trajectory, we compute $\mathbb{C}_t = \mathcal{L}(x_t)$, and check whether to stop at x_t. There are 4 cases: (i) $x_t \in X_G$ (i.e., the goal is reached), (ii) $x_t \notin \mathbb{X}$ (i.e., the state bounds are violated), (iii) $x_t \in (X_s \cup X_f)$ (i.e., the state is already known to be safe or unsafe), or (iv) $t > T^{\max}$, where $T^{\max}$ is a bound on the trajectory length specified by the user. In cases (i), (ii), and (iii), we terminate the trajectory (so $T = t$), and compute a polytopic trajectory by solving (10) with inital state x_0, sequence of affine cells $\mathbb{C}_0, \cdots, \mathbb{C}_{T-1}$, and target polytope $\mathbb{P}_T$ chosen to be the polytope encountered in X_G (case (i)), $\mathbb{R} \setminus \mathbb{X}$ (case (ii)), or $X_s \cup X_f$ (case (iii)). In case (iii), we can determine $\mathbb{P}_T$ since our algorithm represents X_s and X_f each as a finite union of polytopes. In case (iv), we assume that the trajectory belongs to the set $X_a^{\min}$, and ignore the trajectory.

Once we have computed the polytopic trajectory, we insert the polytopes $\mathbb{P}_0, ..., \mathbb{P}_{T-1}$ into the appropriate polytopic tree—i.e., the tree representing X_s if $\mathbb{P}_T$ is safe, and the tree representing X_f if $\mathbb{P}_T$ is unsafe. To do so, we simply set the parent of $\mathbb{P}_t$ to be $\mathbb{P}_{t+1}$ for each $t \in \{0, 1, ..., T-1\}$. Except in case(ii), $\mathbb{P}_T$ is already in the tree; in case (ii), if $\mathbb{P}_T$ is not yet in the polytopic tree representing X_f, then we add it as a new root node. The following result is straightforward:

Theorem 1 (Correctness). *Algorithm 1 returns X_s and X_f, from which all originating trajectories reach X_G and $\mathbb{R}^n \setminus \mathbb{X}$, respectively, in finite time.*

5.3 Probabilistic Completeness

We state the completeness result of our algorithm. We require two mild assumptions, which are similar to the assumptions made in [32].

Assumption 1 *For any measurable $\mathbb{Y} \subseteq \mathbb{X}$, the sampler chooses a point from $\mathbb{Y}$ with non-zero probability.*

Assumption 2 *Given $x_0 \in \mathrm{int}(X_s^{\max} \cup X_f^{\max})$, there is a non-zero probability that the polytopic trajectory optimization (10) provides $\mathbb{P}_0$ with non-zero volume $\mathsf{vol}(\mathbb{P}_0) \geq \lambda > 0$ for some uniform constant λ.*

Assumption 1 is not restrictive as we are often able to sample uniformly from $\mathbb{X}$. Assumption 2 relies on the heuristics used in (10). We empirically have observed that the polytopes always have non-zero volume. If a polytope with zero volume is obtained (even after elongating it across the trajectory, as explained in Section 4), we can discard it and use another heuristic. Moreover, the assumption that a uniform constant exists is not restrictive as there exists a finite number of affine cells with non-zero volume. Therefore, the assumption boils down to (10) being able to find a polytopic inner-approximation to an affine cell that covers a certain fraction of its volume if provided an appropriate cost function.

Theorem 2 (Probabilistic Completeness). *Let X_s^i and X_f^i be the the X_s and X_f computed Algorithm 1 by the i'th iteration. Then if Assumption 1 and Assumption 2 both hold, the following holds:*

$$Pr\left(\lim_{i\to\infty} \operatorname{int}(X_s^{\max} \setminus X_s^i) \cup \operatorname{int}(X_f^{\max} \setminus X_f^i) = \emptyset\right) = 1. \tag{12}$$

Proof. In appendix

6 Example

We adopt example 1 from [23] and [28]. The model represents an inverted pendulum with a spring-loaded wall on one side, as illustrated in Fig. 1 [Left]. The control input is the torque. The system is constrained to $|\theta| \leq 0.12$, $|\dot{\theta}| \leq 1$, $|u| \leq 4$, and the wall is located at $\theta = 0.1$. The problem is to steer the state toward the origin. We set $X_G = [-0.05, 0.05] \times [-0.5, 0.5]$. We know from [23] that X_G is within the region of attraction of the linear quadratic regulator (LQR) controller of the contact-free dynamics (which is Ω in Fig. 1 [Left]). Therefore, once the trajectories end in X_G, they are guaranteed to asymptotically reach the origin. The dynamics is a hybrid model with two modes associated with "contact-free" and "contact". The piecewise affine dynamics is:

$$A_1 = \begin{pmatrix} 1 & 0.01 \\ 0.1 & 1 \end{pmatrix}, A_2 = \begin{pmatrix} 1 & 0.01 \\ -9.9 & 1 \end{pmatrix}, B_1 = B_2 = \begin{pmatrix} 0 \\ 0.01 \end{pmatrix}, c_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, c_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

where mode 1 and 2 correspond to contact-free ($\theta \leq 0.1$) and contact dynamics ($\theta > 0.1$), respectively.

6.1 Controller based on Formal Synthesis

It is non-trivial to control this hybrid system. Both [23] and [28] use approximate explicit hybrid model predictive control (MPC) to compute an inner-approximation of the safe region. The method in [28] is sampling-based and it achieves probabilistic feedback coverage. As shown in Fig. 1 (right), the safe region is **not** the whole state-space and it takes a very non-trivial shape.

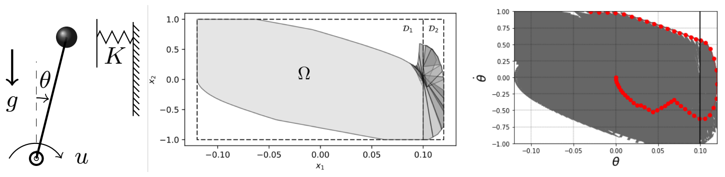


Fig. 1. Example: The inverted pendulum with wall (left). The model-based safe regions as obtained from figures in [23] (middle), and [28] (right).

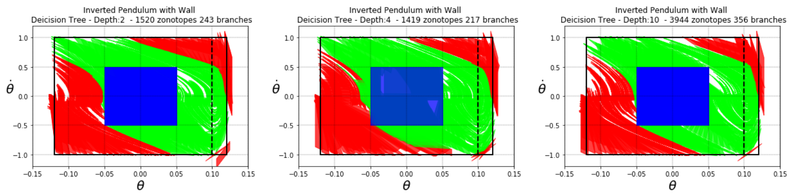


Fig. 2. Example: Verified regions for system controlled with a decision tree. X_G, X_s, X_f are shared in blue, green, and red, respectively.

6.2 Decision Tree Controllers

Training. We use the VIPER algorithm to learn decision trees [6]. Learning decision tree controllers is challenging due to their discrete structure—in particular, the gradient of the loss function for a decision tree is zero almost everywhere. To address this challenge, VIPER first learns a neural network controller using standard reinforcement learning algorithms. Then, it uses *imitation learning* to train a decision tree controller that "mimics" the neural network controller [26]. In particular, given an "oracle controller" (i.e., the neural network), imitation learning reduces the reinforcement learning problem for the decision tree controller to a supervised learning problem. Thus, we can use standard CART algorithm for learning decision trees [7]. See [6] for details.

Verification. The results are shown in Fig. 6.2. Since all decision trees are extracted from the same neural network, their performance looks quite similar, albeit the smallest decision tree with depth 2 performs slightly weaker than the others, which is interpreted by the larger unsafe region and smaller safe region. All the decision trees perform superb handling the contact, which is due to the fact that constant controller in this area of the state space is sufficient. On the other hand, the decision tree behaves very poorly with negative angle and small velocities as piecewise constant decisions are not sufficient to conduct a maneuver that brings the state to get into the goal.

6.3 Neural Network Controllers

Training. The neural network used to guide the training of the decision tree is too large for our algorithm to verify. It is known that neural networks must

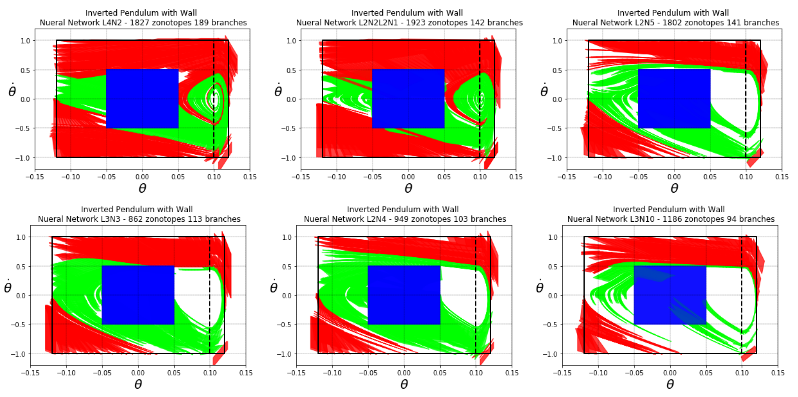


Fig. 3. Example: Verified regions for system controlled with a neural network. X_G, X_s, X_f are shared in blue, green, and red, respectively.

be vastly overparameterized to obtain good performance (otherwise, they are susceptible to local optima) [22], which explains why we can approximate a large neural network with small decision trees. Indeed, we found that training a small neural network using reinforcement learning was intractable. Instead, we used supervised learning—we run the model-based PWA controller described in [23] to generate state-control pairs, and construct ReLU-based DNN of various depth and size to minimize the mean-square-error between the network prediction and those labels. Using this approach, the neural networks still performed poorly, even though they were comparable in size to the smallest decision tree. Nevertheless, this experiment complements the decision tree experiment—the emphasis for decision trees is on the verifying "good" behaviors, whereas the emphasis here is on identifying "bad" behaviors.

Verification. Figure 6.3 shows the verified regions of various neural network controllers. We name the network by its size and depth, e.g. L4N2 means the network has four layers with two ReLUs each, and L2N2L2N1 means the first two layers have two ReLUs each and the last two layers have just one each; in addition to these ReLU-based layers, all networks also have a final affine layer.

We observe that on average, the algorithm covers a larger falsified region, which is to be expected since the network is not an ideal stabilizing controller. The plots in each column exhibit a variety of different traits. For example, the figures in the first column exhibit lack of robustness of the neural network—around the vertical strip of between $[0.05, 0.10]$, there is a slim unsafe region encompassed by a large safe region. In particular, in this portion of the state space, even a very small deviation from the state space can lead to unsafety.

The second column has an unclassified region (white). Upon manually inspecting the control behavior, we observe chattering near the point of contact with the wall—i.e., the system repeatedly hit and bounced off of the wall. This behavior showcases a limitation of learning-based controllers. The average time for each linear program in verifying the largest NN controller was less than 0.1 seconds using Gurobi [13]. The total number of linear programs to be solved is equal to the total number of branches.

7 Discussion and Future Work

Advantages. The main advantage of the method in this paper is that instead of verifying a property over a region, it is able to compute regions of safe and unsafe states, which is more relevant in problems where the set of safe states is a complex subset of the state space. Our approach is easy to deploy as it based on anytime algorithm that only requires sampling, forward simulation, and linear programming, and unlike most related verification techniques, it has the potential to be applied to large-scale problems.

Limitations. The last advantage point above is conditioned on the number of samples and polytopic branches required to provide a reasonable volume of verified regions. A major drawback of our approach stems from the fact that it relies on each polytope to be contained within a single affine cell. For many machine learning components, especially DNNs, these affine cells can be very small, thus making polytopes small hence requiring denser trees for proper coverage. This issue was alleviated for decision tree controllers, where the number of affine cells scales linearly with the number of leaves. A promising direction to resolve this issue is focusing on interpretable neural networks [3, 19], which are trained in a way that they are less fragile and typically have fewer and larger affine cells. This limitation is less severe in many applications that we are only interested in verifying regions around a nominal point or trajectory.

Another limitation of our approach is the restriction to PWA dynamical systems and controllers. While the linearity assumption around a nominal trajectory is reasonable, this restriction limits us from constructing larger verified regions that are able to incorporate nonlinearities.

Future directions. An immediate future direction is characterizing $X_a^{\max}$. A potential approach is searching for equilibria or limit cycles and applying the methods in this paper to characterize states leading to limit cycles. Moreover, we are planning to leverage the results in this paper to develop compositional methods for controller synthesis and verification. For instance, by learning a controller that steers trajectories from the unsafe region of another controller to its safe regions, we can compose controllers in a hierarchical manner. This approach has the potential to provide a framework for learning-based control synthesis in a more scalable, interpretable, and formally correct way.

References

1. Abbeel, P., Coates, A., Quigley, M., Ng, A.Y.: An application of reinforcement learning to aerobatic helicopter flight. In: Advances in neural information processing systems. pp. 1–8 (2007)
2. Althoff, M., Stursberg, O., Buss, M.: Computing reachable sets of hybrid systems using a combination of zonotopes and polytopes. Nonlinear analysis: hybrid systems 4(2), 233–249 (2010)
3. Alvarez-Melis, D., Jaakkola, T.S.: Towards robust interpretability with self-explaining neural networks. arXiv preprint arXiv:1806.07538 (2018)
4. Andrychowicz, M., Baker, B., Chociej, M., Jozefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., et al.: Learning dexterous in-hand manipulation. arXiv preprint arXiv:1808.00177 (2018)
5. Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A., Criminisi, A.: Measuring neural net robustness with constraints. In: Advances in neural information processing systems. pp. 2613–2621 (2016)
6. Bastani, O., Pu, Y., Solar-Lezama, A.: Verifiable reinforcement learning via policy extraction. arXiv preprint arXiv:1805.08328 (2018)
7. Breiman, L.: Classification and regression trees. Routledge (2017)
8. Chen, X., Abraham, E., Sankaranarayanan, S.: Taylor model flowpipe construction for non-linear hybrid systems. In: Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd. pp. 183–192. IEEE (2012)
9. Collins, S., Ruina, A., Tedrake, R., Wisse, M.: Efficient bipedal robots based on passive-dynamic walkers. Science 307(5712), 1082–1085 (2005)
10. Gao, S., Kong, S., Clarke, E.M.: dReal : An SMT Solver for Nonlinear Theories over the Reals. In: Automated Deduction–CADE-24, pp. 208–214. No. 1041377, Springer (2013)
11. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: Ai 2: Safety and robustness certification of neural networks with abstract interpretation
12. Girard, A.: Reachability of uncertain linear systems using zonotopes. In: International Workshop on Hybrid Systems: Computation and Control. pp. 291–305. Springer (2005)
13. Gurobi Optimization, I.: Gurobi optimizer reference manual (2016), http://www.gurobi.com
14. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: International Conference on Computer Aided Verification. pp. 3–29. Springer (2017)
15. Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig: verifying safety properties of hybrid systems with neural network controllers. arXiv preprint arXiv:1811.01828 (2018)
16. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient smt solver for verifying deep neural networks. In: International Conference on Computer Aided Verification. pp. 97–117. Springer (2017)
17. Kong, S., Gao, S., Chen, W., Clarke, E.: dreach: δ-reachability analysis for hybrid systems. In: International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems. pp. 200–205. Springer (2015)
18. Kopetzki, A.K., Schürmann, B., Althoff, M.: Efficient methods for order reduction of zonotopes. In: Proc. of the 56th IEEE Conference on Decision and Control (2017)

19. Lei, T., Barzilay, R., Jaakkola, T.: Rationalizing neural predictions. arXiv preprint arXiv:1606.04155 (2016)
20. Levine, S., Finn, C., Darrell, T., Abbeel, P.: End-to-end training of deep visuomotor policies. The Journal of Machine Learning Research 17(1), 1334–1373 (2016)
21. Levine, S., Koltun, V.: Guided policy search. In: International Conference on Machine Learning. pp. 1–9 (2013)
22. Li, Y., Liang, Y.: Learning overparameterized neural networks via stochastic gradient descent on structured data. In: Advances in Neural Information Processing Systems. pp. 8168–8177 (2018)
23. Marcucci, T., Deits, R., Gabiccini, M., Biechi, A., Tedrake, R.: Approximate hybrid model predictive control for multi-contact push recovery in complex environments. In: Humanoid Robotics (Humanoids), 2017 IEEE-RAS 17th International Conference on. pp. 31–38. IEEE (2017)
24. Pan, Y., Cheng, C.A., Saigol, K., Lee, K., Yan, X., Theodorou, E., Boots, B.: Learning deep neural network control policies for agile off-road autonomous driving. In: The NIPS Deep Rienforcement Learning Symposium (2017)
25. Raghunathan, A., Steinhardt, J., Liang, P.S.: Semidefinite relaxations for certifying robustness to adversarial examples. In: Advances in Neural Information Processing Systems. pp. 10900–10910 (2018)
26. Ross, S., Gordon, G., Bagnell, D.: A reduction of imitation learning and structured prediction to no-regret online learning. In: Proceedings of the fourteenth international conference on artificial intelligence and statistics. pp. 627–635 (2011)
27. Sadraddini, S., Tedrake, R.: Linear encodings for polytope containment problems. arXiv preprint arXiv:1903.05214 (2019)
28. Sadraddini, S., Tedrake, R.: Sampling-based polytopic trees for approximate optimal control of piecewise affine systems. In: International Conference on Robotics and Automation (ICRA) (2019)
29. Su, J., Vargas, D.V., Sakurai, K.: One pixel attack for fooling deep neural networks. IEEE Transactions on Evolutionary Computation (2019)
30. Sun, X., Khedr, H., Shoukry, Y.: Formal verification of neural network controlled autonomous systems. arXiv preprint arXiv:1810.13072 (2018)
31. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R.: Intriguing properties of neural networks. In: ICLR (2014)
32. Tedrake, R., Manchester, I.R., Tobenkin, M., Roberts, J.W.: Lqr-trees: Feedback motion planning via sums-of-squares verification. The International Journal of Robotics Research 29(8), 1038–1052 (2010)
33. Tjeng, V., Tedrake, R.: Verifying neural networks with mixed integer programming. arXiv preprint arXiv:1711.07356 (2017)
34. Tjeng, V., Xiao, K.Y., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming (2018)
35. Tøndel, P., Johansen, T.A., Bemporad, A.: Evaluation of piecewise affine control via binary search tree. Automatica 39(5), 945–950 (2003)
36. Wong, E., Kolter, Z.: Provable defenses against adversarial examples via the convex outer adversarial polytope. In: International Conference on Machine Learning. pp. 5283–5292 (2018)
37. Xiang, W., Lopez, D.M., Musau, P., Johnson, T.T.: Reachable set estimation and verification for neural network models of nonlinear dynamic systems. In: Safe, Autonomous and Intelligent Vehicles, pp. 123–144. Springer (2019)
38. Xiang, W., Tran, H.D., Johnson, T.T.: Specification-guided safety verification for feedforward neural networks. arXiv preprint arXiv:1812.06161 (2018)

8 Appendix

8.1 Proof of Theorem 2

Proof. Let $X_r^i = X_s^{\max} \setminus X_s^i$. We need to show that $\lim_{i\to\infty} \mathrm{int}(X_r^i) = \emptyset$ with probability one. The same argument holds for $X_f^{\max} \setminus X_f^i$. We prove by contradiction. Let $\lim_{i\to\infty} X_r^i$ be a measurable set. It follows from Assumption 1 that we obtain a sample from X_r^∞ with non-zero probability. Then we simulate the system forward from the sampled state x_0. Once the trajectory is obtained, the polytopic trajectory is computed, and by Assumption 2, we obtain a measurable $\mathbb{P}_0$, centered at $x_0 \in \mathrm{int}(X_r^\infty)$ that has non-empty intersection with X_r^∞. Therefore, by non-zero probability, the volume of X_r^i shrinks in one iteration by at least λ. Thus we reach a contradiction.