2021 IEEE International Conference on Robotics and Automation (ICRA 2021)
May 31 - June 4, 2021, Xi'an, China

# Reachable Polyhedral Marching (RPM): A Safety Verification Algorithm for Robotic Systems with Deep Neural Network Components

Joseph A. Vincent[1] and Mac Schwager[1]

2021 IEEE International Conference on Robotics and Automation (ICRA) | 978-1-7281-9077-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICRA48506.2021.9561956

***Abstract*— We present a method for computing exact reachable sets for deep neural networks with rectified linear unit (ReLU) activation. Our method is well-suited for use in rigorous safety analysis of robotic perception and control systems with deep neural network components. Our algorithm can compute both forward and backward reachable sets for a ReLU network iterated over multiple time steps, as would be found in a perception-action loop in a robotic system. Our algorithm is unique in that it builds the reachable sets by incrementally enumerating polyhedral cells in the input space, rather than iterating layer-by-layer through the network as in other methods. If an unsafe cell is found, our algorithm can return this result without completing the full reachability computation, thus giving an anytime property that accelerates safety verification. In addition, our method requires less memory during execution compared to existing methods where memory can be a limiting factor. We demonstrate our algorithm on safety verification of the ACAS Xu aircraft advisory system. We find unsafe actions many times faster than the fastest existing method and certify no unsafe actions exist in about twice the time of the existing method. We also compute forward and backward reachable sets for a learned model of pendulum dynamics over a 50 time step horizon in 87s on a laptop computer. Algorithm source code: `https://github.com/StanfordMSL/Neural-Network-Reach`.**

## I. Introduction

In this paper we present the Reachable Polyhedral Marching (RPM) algorithm for computing forward and backward reachable sets of deep neural networks with rectified linear unit (ReLU) activation. This is a critical building block to proving safety properties for autonomous systems with learned perception, dynamics, or control components in the loop. Specifically, given a set of input values, RPM will compute the set of all output values that can be obtained under the ReLU network (the forward reachable set, or image, of the input set). Similarly, given a set of output values, RPM will compute the set of all input values that can lead to those output values under the ReLU network (the backward reachable set, or preimage, of the output values). When the ReLU network is part of a dynamical process, as is common in robots with deep learned perception or control components, RPM can compute such reachable sets for multiple time steps into the future or the past without explicitly iterating over each time step. Figure 1 shows the incremental nature of how RPM enumerates input space polyhedra over which the ReLU network is affine.

This work was supported by NSF grant 1830402, DARPA grant D18AP00064, NASA grant 80NSSC20M0163, and the Dwight D. Eisenhower Transportation Fellowship. The NASA University Leadership Initiative (grant #80NSSC20M0163) provided funds to assist the authors with their research, but this article solely reflects the opinions and conclusions of its authors and not any NASA entity. We are grateful for this support.

[1]Department of Aeronautics and Astronautics, Stanford University, Stanford, CA 94305, USA, {josephav, schwager}@stanford.edu

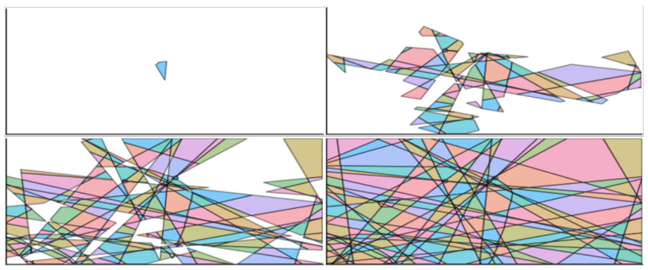


Fig. 1: Snapshots of how RPM incrementally enumerates the 2D input space polyhedra for which a random ReLU network is affine, resulting in the explicit piecewise-affine representation. Polyhedron color is random.

All existing algorithms that compute exact reachable sets iterate through the network layer-by-layer [1], [2], [3], which may become intractable if the network is applied iteratively in a feedback loop. RPM applies a fundamentally different approach to avoid layer-by-layer computation. Instead, we compute the reachable set one polyhedral cell at a time, where each cell represents a region of the input space over which the network is affine. Each cell can be computed quickly through a series of linear programs equal to the number of neurons in the network, regardless of width or depth. Our algorithm computes cells until the explored set of cells fills the desired reachable set. In this way, our method is geometrically similar to fast marching methods in optimal control [4], path planning [5], [6], and graphics [7], [8].

We demonstrate RPM in two examples related to the safety verification of dynamical systems with learned components. First, we compute forward reachable and backward reachable sets for a learned dynamical model of a pendulum over 50 time steps in the future and past, respectively. Furthermore, we compare with a state of the art reachability method [3] in a safety verification problem involving ACAS Xu, a neural network policy for aircraft collision avoidance. Our algorithm finds unsafe inputs for this network many times faster than [3] and when no unsafe input exists, our algorithm certifies safety approximately 2 times slower.

The paper is organized as follows. We give related work in Section II and give background and state the problem in Section III. Section IV presents our main algorithm, and explains its derivation. In Section V we describe how the RPM algorithm can be used to perform forward and backward reachability over multiple time steps. In Section VI we present the aforementioned examples of RPM.

## II. Related Work

Though the analysis of neural networks is a young field, a broad literature has emerged to address varied questions related to interpretability, verification, and safety. Much work has been dedicated to characterizing the expressive potential of ReLU networks by studying how the number of affine regions scales with network depth and width [9], [10], [11]. Other research includes encoding piecewise-affine (PWA) functions as ReLU networks [12], [13], learning deep signed distance functions and extracting level sets [8], and learning neural network dynamics or controllers that satisfy stability conditions [14], [15], which may be more broadly grouped with correct-by-construction training approaches [16], [17].

Spurred on by pioneering methods such as Reluplex [18], the field of neural network verification has emerged to address the problem of analyzing properties of neural networks over continuous input sets. A survey of the neural network verification literature is given by [19]. Reachability approaches are subset of this literature and are especially useful for analysis of learned dynamical systems. Reachability methods can be categorized into overapproximate and exact methods. Overapproximate methods often compute neuron-wise bounds either from interval arithmetic or symbolically [20], [21], [22], [23], [24]. Optimization based approaches such as (mixed-integer) linear programming are also used to solve for bounds on a reachable output set in [25], [26], [27], [28]. Other approaches include modeling the network as a hybrid system [29], abstracting the domain [30], and performing layer-by-layer operations on zonotopes [31]. Further, [32] demonstrated how [18], [25], [22], [29] could be used to perform closed-loop reachability of a dynamical system given a neural network controller.

Exact reachability methods have also been proposed, although to a lesser degree [1], [2], [3]. These methods generally perform the set operations of affine transformation, intersection/division, and projection layer-by-layer through a ReLU network. Layer-by-layer approaches have also been proposed to solve for the explicit PWA representation of a ReLU network [33], [34]. Exact methods, unlike overapproximate methods, can compute backward reachable sets.

Our RPM algorithm inherits all advantages of exact methods, but is unique in that it does not iterate layer-by-layer. This can lead to faster verification decisions when finding unsafe inputs. Layer-by-layer methods must compute the entire reachable set, regardless of whether a network violates a safety property. In contrast, if our algorithm encounters a cell with an unsafe input, it can return that result immediately without computing the entire reachable set. Finally, all intermediate polyhedra and affine map matrices of layer-by-layer methods must be stored in memory until the algorithm terminates (since intermediate polyhedra may be split by later neurons). Conversely, since our method computes the true PWA function incrementally, once a polyhedron-affine map pair is computed it can be sent to external memory and only a binary vector (of the neuron activations) needs to be stored to continue the algorithm. This is especially useful because the number of affine regions is hard to estimate before runtime.

## III. Background and Problem Statement

An $n$-layer feedforward neural network implements a function $F(\mathbf{x}) : \mathbb{R}^{k_0} \to \mathbb{R}^{k_n}$ to give a map from inputs $\mathbf{x}$ to outputs $\mathbf{y} = F(\mathbf{x})$. Each layer in $F$ is a function $F_i(\mathbf{z}_{i-1}) : \mathbb{R}^{k_{i-1}} \to \mathbb{R}^{k_i}$ where $\mathbf{z}_i \in \mathbb{R}^{k_i}$ is the hidden layer variable for layer $i$. We assume $\mathbf{z}_0 = \mathbf{x}$ and $\mathbf{z}_n = \mathbf{y}$. The function for layer $i$ is

$$F_i(\mathbf{z}_{i-1}) = \mathbf{z}_i = \sigma_i(\hat{\mathbf{z}}_i) = \sigma_i(\mathbf{W}_i\mathbf{z}_{i-1} + \mathbf{b}_i), \tag{1}$$

where $\sigma_i$ is the activation function, $\hat{\mathbf{z}}_i$ is the preactivation value, and $\mathbf{W}_i$ and $\mathbf{b}_i$ are the weights and biases for layer $i$. We assume $\sigma_n$ is an identity map and all hidden layer activations are the rectified linear unit (ReLU)

$$\sigma_i(\hat{\mathbf{z}}_i) = [max(0, \hat{z}_{i,1}), ..., max(0, \hat{z}_{i,k_i})]^\top. \tag{2}$$

From the above definition we can augment the weights to construct an equivalent neural network that has no bias terms. This equivalent formulation is such that $\tilde{F}(\tilde{\mathbf{x}}) : \mathbb{R}^{k_0+1} \to \mathbb{R}^{k_n}$ and $\tilde{\mathbf{x}} = [x_1, x_2, ..., x_n, 1]^\top$. We then have

$$\tilde{F}_i(\tilde{\mathbf{z}}_{i-1}) = \tilde{\mathbf{z}}_i = \sigma_i(\tilde{\mathbf{W}}_i\tilde{\mathbf{z}}_{i-1}) \tag{3a}$$

$$\tilde{\mathbf{W}}_i = \begin{bmatrix} \mathbf{W}_i & \mathbf{b}_i \\ \mathbf{0}^\top & 1 \end{bmatrix} \tag{3b}$$

$$\tilde{\mathbf{W}}_n = \begin{bmatrix} \mathbf{W}_n & \mathbf{b}_n \end{bmatrix}. \tag{3c}$$

It is well-known that the function implemented by a ReLU network is continuous and PWA [9], [10], [11]. We mathematically characterize PWA functions below.

*Definition 1 (Polyhedral Complex):* A polyhedral complex $\mathscr{C}$ is a finite set of convex polyhedra where every polyhedron contains its faces and the intersection between two polyhedra is either a face of both $c_1$ and $c_2$ or empty. We henceforth refer to convex polyhedra simply as polyhedra.

A PWA function is a function $M(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^m$ where

$$M(\mathbf{x}) = \mathbf{C}_i\mathbf{x} + \mathbf{d}_i \quad \forall \mathbf{x} \in c_i \quad \forall c_i \in \mathscr{C} \tag{4}$$

for $\mathbf{C}_i \in \mathbb{R}^{m\times n}$, $\mathbf{d}_i \in \mathbb{R}^m$, and $c_i$ elements of a polyhedral complex $\mathscr{C}$. Here we consider only continuous PWA functions. Equation (4) is the *explicit* PWA representation.

The forward reachability problem is to solve for the image of a specified input set under the neural network map

$$\mathscr{Y} = \{F(\mathbf{x}) | \mathbf{x} \in \mathscr{X}\}. \tag{5}$$

Likewise, the backward reachability problem is to solve for the preimage of a specified output set

$$\mathscr{X} = \{x | F(\mathbf{x}) \in \mathscr{Y}\}. \tag{6}$$

Our algorithm transforms a ReLU network into an explicit PWA representation by computing a set of polyhedra ($c_i$) and associated affine maps ($\mathbf{C}_i, \mathbf{d}_i$). There exist efficient methods for computing forward and backward reachable sets of polyhedral sets under PWA maps. Accordingly, we restrict our focus to these problems.

## IV. Explicit PWA Representation

First, we seek to construct the explicit PWA representation of a ReLU network. Our method constructs a PWA function for a ReLU network by enumerating each polyhedral cell and its associated affine map. In Sec. IV-A we show how cells and affine maps are computed from the *activation pattern* of a ReLU network. In Sec. IV-B we show how cell representations are reduced to a minimal form, which is used in Sec. IV-C to determine neighboring polyhedra given a current polyhedron. This leads to a recursive procedure in which we explore an expanding front of polyhedra, ultimately giving the explicit PWA representation, as explained in Sec. IV-D.

## A. Determining Polyhedral Cells from Activation Patterns

For a given input to a ReLU network, every hidden neuron has an associated binary neuron activation of zero or one corresponding to whether the preactivation value was nonpositive or positive, respectively[1]. We refer to the vector of all neuron activations as the *activation pattern* of the network for a given input. For neuron $j$ in layer $i$,

$$AP_{ij}(\mathbf{x}) = \begin{cases} 1 & \hat{z}_{ij} > 0 \\ 0 & \hat{z}_{ij} \leq 0 \end{cases} \tag{7a}$$

$$AP_i(\mathbf{x}) = [AP_{i,1}(\mathbf{x}), ..., AP_{i,k_i}(\mathbf{x})]^\top \tag{7b}$$

$$AP(\mathbf{x}) = (AP_1(\mathbf{x}), ..., AP_{n-1}(\mathbf{x})). \tag{7c}$$

For a network with $N$ hidden neurons there are $2^N$ possible combinations of neuron activations, however not all are realizable by the network. Practically, the number of activation patterns is better approximated by $N^{k_0}$ [35].

To construct an explicit PWA representation we first want to characterize the set of inputs $c$ for which the network reduces to a single affine map. For a fixed activation pattern the ReLU network simplifies to the affine map $\mathbf{Cx}+\mathbf{d}$ where

$$\begin{bmatrix}\mathbf{C} & \mathbf{d}\end{bmatrix} = \tilde{\mathbf{W}}_n \prod_{i=1}^{n-1} \tilde{\mathbf{W}}_i^c \tag{8a}$$

$$\tilde{\mathbf{W}}_i^c(\mathbf{x}) = diag(AP_i(\mathbf{x}))\tilde{\mathbf{W}}_i, \tag{8b}$$

and $diag(\cdot)$ diagonalizes a vector to a matrix. Equation (8b) equivalently sets row $j$ of $\tilde{\mathbf{W}}_i$ to zero if $AP_{ij} = 0$.

The activation pattern only changes if some $\hat{z}_{ij}$ switches from positive to nonpositive (or vice-versa), $c$ is thus given by a set of linear constraints, one for each neuron in the network. The constraint for neuron $j$ in layer $i$ is

$$\begin{cases} \mathbf{a}_{ij}^\top \mathbf{x} \geq b_{ij} & \text{if } AP_{ij} = 1 \\ \mathbf{a}_{ij}^\top \mathbf{x} \leq b_{ij} & \text{if } AP_{ij} = 0 \end{cases} \tag{9}$$

where

$$\begin{bmatrix}\mathbf{a}_{ij} & -b_{ij}\end{bmatrix} = \tilde{\mathbf{W}}_i[j,:] \prod_{l=1}^{i-1} \tilde{\mathbf{W}}_l, \tag{10}$$

and $\tilde{\mathbf{W}}_i[j,:]$ denotes the $j^{th}$ row of $\tilde{\mathbf{W}}_i$. Equation (10) is similar to (8a) but instead simply gives the affine map parameters for a single neuron output. Strict inequalities are not present in (9) because the affine map $\mathbf{Cx}+\mathbf{d}$ holds over the interior and boundary of the set where the activation pattern is constant. Since $c$ is an intersection of halfspaces, the set is a polyhedron. A similar formulation is given in [8] for finding the surface of an object for graphics rendering. Note that it is not uncommon for neurons to have the degenerate linear constraint $\mathbf{0}^\top \mathbf{x} \leq 0$ which is satisfied for all $\mathbf{x} \in c$, and also for multiple different neurons in the network to have equivalent constraints (perhaps scaled arbitrarily). To address these edge cases we introduce Definitions 2, 3, 4 before describing our fast marching method below.

*Definition 2 (H-representation):* The halfspace representation (H-representation) of a polyhedron is

$$P_H = \{\mathbf{x} \in \mathbb{R}^d | \mathbf{Ax} \leq \mathbf{b}\} \tag{11}$$

[1]This choice is arbitrary and some others use the opposite convention.

where $\mathbf{A}$ an $m \times d$ matrix and $\mathbf{b}$ an $m$-dimensional vector. Equality constraints may also be included if the set $P_H$ is of lower dimension than the ambient dimension.

*Definition 3 (Duplicate Constraints):* A constraint $\mathbf{a}_j^\top \mathbf{x} \leq b_j$ is duplicate if there exists a scalar $\alpha$ and a prior constraint $\mathbf{a}_i^\top \mathbf{x} \leq b_i$ where $i < j$ such that $\alpha[\mathbf{a}_j \; b_j] = [\mathbf{a}_i \; b_i]$ where $\alpha > 0$.

*Definition 4 (Redundant & Essential Constraints):* A constraint $\mathbf{a}_i^\top \mathbf{x} \leq b_i$ is redundant if the feasible set does not change upon its removal or if it is a duplicate constraint. An essential constraint is one which is not redundant.

We use (9) to construct an H-representation of a cell $c$ by enumerating all constraints associated with hidden layer neurons. The resulting $\mathbf{a}_{ij}$ vectors and $b_{ij}$ scalars are collected into ($\mathbf{A}$, $\mathbf{b}$) to give the desired H-representation from (11). For an arbitrary ReLU network in an arbitrary activation pattern, we find that the resulting H-representation often has redundant constraints, which need to be removed to give a minimal representation of the polyhedron.

## B. Finding Essential Constraints

We first normalize all constraints, remove any duplicates, and consider the resulting H-representation $\mathbf{Ax} \leq \mathbf{b}$. To determine if the $i$th constraint is essential or redundant, define a new set of constraints with the $i$th constraint removed,

$$\tilde{\mathbf{A}} = [\mathbf{a}_1 \; ... \; \mathbf{a}_{i-1} \; \mathbf{a}_{i+1} \; ... \; \mathbf{a}_m]^\top \tag{12a}$$

$$\tilde{\mathbf{b}} = [b_1 \; ... \; b_{i-1} \; b_{i+1} \; ... \; b_m]^\top, \tag{12b}$$

and solve the linear program

$$\max_{\mathbf{x}} \quad \mathbf{a}_i^\top \mathbf{x} \tag{13a}$$

$$\text{subject to} \quad \tilde{\mathbf{A}}\mathbf{x} \leq \tilde{\mathbf{b}}. \tag{13b}$$

If the optimal objective value is less than or equal to $b_i$, constraint $i$ is redundant. In the worst case, a single LP must be solved for each constraint to determine whether it is essential or redundant. However, heuristics exist to avoid this worst case complexity. We use the bounding box heuristic to quickly find a subset of redundant constraints [36]. Empirically, this results in identifying about 90% of the redundant constraints. We find that other heuristics beyond the bounding box method do not improve performance. Our implementation uses the GLPK open-source LP solver and the JuMP optimization package [37].

## C. Determining Neighboring Activation Patterns

Given a cell $c$ in the input space of a ReLU network, we are interested in finding the activation pattern corresponding to a neighboring cell, $c'$. We can then determine the H-representation of $c'$ using the procedure outlined above. The challenge in finding the activation pattern for $c'$ is in determining which individual neuron activations ($AP_{ij}$ for some $ij$) must be flipped when $\mathbf{x}$ transitions from $c$ to $c'$.

The boundary between $c$ and $c'$ ($c \cap c'$) is defined by a linear constraint of $c$. We refer to this as the *neighbor constraint*. Intuitively, to generate $AP^{c'}$ one ought to simply flip the activation of the neuron defining this neighbor constraint (and all neurons defining duplicate constraints). However, this intuitive procedure is incomplete because it does not correctly deal with degenerate constraints $\mathbf{0}^\top \mathbf{x} \leq 0 \; \forall \mathbf{x} \in c$, which are common in practice. A degenerate constraint holds everywhere over $c$, but it may turn into a non-degenerate constraint for $c'$ due to a neuron activation in an earlier layer being flipped when moving to cell $c'$. Conversely, a non-degenerate constraint in $c$ may turn into a degenerate one when passing to $c'$. To define a correct procedure for flipping neuron activations we introduce a taxonomy for classifying neurons based on their associated linear constraints.

*Definition 5 (Type 1, 2, 3 neurons):* When identifying a neighboring cell $c'$ from a current cell $c$, neurons whose linear constraints in $c$ define $c \cap c'$ are called *Type 1.* neurons with degenerate linear constraints in $c$, $\mathbf{0}^\top \mathbf{x} \leq 0 \ \forall \mathbf{x} \in c$, are called *Type 2*. All other neurons are *Type 3*.
Constraints labeled by type are illustrated in Figure 2a.

Though not stated explicitly, the linear constraint parameters from (9) are (nonlinear) functions of the input. Given Definition 5, for inputs $\mathbf{x} \in c \cap c'$,

$$\mathbf{a}_{ij}^\top(\mathbf{x})\mathbf{x} = b_{ij}(\mathbf{x}) \quad \text{if } ij \in \text{Type 1} \quad (14a)$$

$$\mathbf{a}_{ij}^\top(\mathbf{x})\mathbf{x} = b_{ij}(\mathbf{x}) \quad \text{if } ij \in \text{Type 2} \quad (14b)$$

$$\begin{cases} \mathbf{a}_{ij}^\top(\mathbf{x})\mathbf{x} > b_{ij}(\mathbf{x}) & \text{if } ij \in \text{Type 3 and } AP_{ij} = 1 \\ \mathbf{a}_{ij}^\top(\mathbf{x})\mathbf{x} < b_{ij}(\mathbf{x}) & \text{if } ij \in \text{Type 3 and } AP_{ij} = 0 \end{cases} . \quad (14c)$$

The function $\mathbf{a}_{ij}^\top(\mathbf{x})\mathbf{x} - b_{ij}(\mathbf{x})$ is continuous because it is simply the function for computing the preactivation value $\hat{z}_{ij}$. From this continuity, we know that the strict inequality of (14c) holds for all inputs in $c'$ as well. Therefore, the activation of a Type 3 neuron is not flipped.

Conversely, inputs in $c'$ may be either feasible or infeasible for Type 1 and Type 2 constraints. Figure 2b-c illustrates the possible feasible and infeasible regions of a Type 1 constraint. Figure 2d-e illustrates the possible feasible and infeasible regions of a Type 2 constraint. It follows that the procedure to generate a neighboring activation pattern is to apply (9) layer-by-layer updating the activation pattern in place, where Type 1 and Type 2 neuron activations are set to zero if their new linear constraints are $\mathbf{0}^\top \mathbf{x} \leq 0$, otherwise, they are flipped. This procedure is formally defined in Algorithm 1. We note that applying this neuron flipping algorithm to identify a neighboring cell $c'$ is exceedingly fast, even for networks with 10,000s of neurons, as we only require a single logical check at each neuron.

## D. Reachable Polyhedral Marching

From (9) and Algorithm 1, we define our main algorithm, Reachable Polyhedral Marching (RPM) in Algorithm 2 for explicitly enumerating all polyhedral cells in the input space of a ReLU network, resulting in the explicit PWA representation. First, we start with an initial point in the input space and evaluate the network to find the activation pattern. From this the H-representation is found using (9). Essential constraints are then identified as described in Section IV-B. For each essential constraint we generate a neighboring activation pattern using Algorithm 1. The process then repeats with each new neighboring activation pattern being added to a working set. A neighbor activation pattern only gets added to the working set if it has not already been visited and it is not already in the working set. For a given starting cell each neighbor cell is enumerated, and since each is connected to another, Algorithm 2 is guaranteed to enumerate every cell in the input space. Figure 1 shows the result of applying Algorithm 2 to a randomly initialized ReLU network.

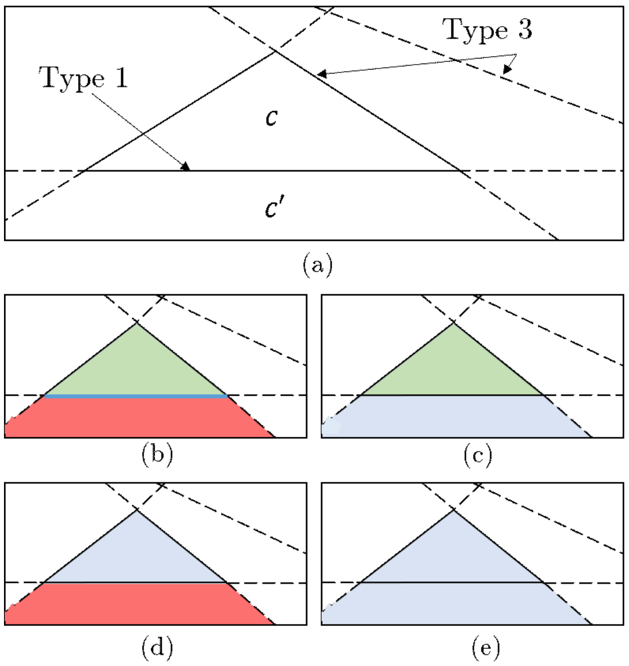


Fig. 2: (a) Example $c$ and $c'$ with Type 1 and Type 3 constraints labeled. For (b)-(e), regions in green denote where the nonlinear constraint is strictly satisfied, regions in blue denote where the nonlinear constraint function evaluates to zero, and regions in red denote where the nonlinear constraint is violated. (b) Type 1 constraint where the neuron activation must flip when going from $c$ to $c'$. (c) Type 1 constraint where the neuron activation is set to zero when going from $c$ to $c'$. (d) Type 2 constraint where the neuron activation must flip when going from $c$ to $c'$. (e) Type 2 constraint where the neuron activation remains zero when going from $c$ to $c'$.

**Algorithm 1:** Neighboring Activation Pattern

```
Input: $AP^c$, $Type\ 1$, $Type\ 2$, $(W_0, \dots, W_N)$
Output: $AP^{c'}$
1  $AP^{c'} \leftarrow AP^c$                         ▷ Initialize new AP as old AP
2  for $i \in [1,\dots,n-1]$, $j \in [1,\dots k_i]$ do ▷ For each neuron
3      if $ij \in Type\ 1$ or $ij \in Type\ 2$ then
4          $\mathbf{a}_{ij}$, $b_{ij} \leftarrow$ Equation 10      ▷ $AP_{ij}^{c'}$ hyperplane
5          if $[\boldsymbol{a}_{ij}\ b_{ij}] = \boldsymbol{0}$ then
6              $AP_{ij}^{c'} \leftarrow 0$
7          else
8              $AP_{ij}^{c'} \leftarrow \neg AP_{ij}^{c'}$
9          end
10     end
11 end
12 return $AP^{c'}$
```

**Algorithm 2:** Reachable Polyhedral Marching

```
Input: $AP^{c_0}$, $(W_0,\dots,W_N)$
Output: Explicit PWA representation
1  input cells = $\emptyset$; visited cells = $\emptyset$
2  working set = $\{AP^{c_0}\}$
3  while working set $\neq \emptyset$ do
4      $AP^c \leftarrow$ pop element off of working set
5      $\mathbf{C}^c$, $\mathbf{d}^c \leftarrow$ Equation 8a        ▷ Retrieve affine map
6      $H^c_{rep} \leftarrow$ Equation 9     ▷ Retrieve H-representation
7      $H^c_{rep} \leftarrow$ remove redundant $H^c_{rep}$ constraints
8      push $(\mathbf{C}^c$, $\mathbf{d}^c$, $H^c_{rep})$ onto input cells
9      for $\boldsymbol{a}_k, b_k \in H^c_{rep}$ do
10         $AP^{c'} \leftarrow$ Algorithm 1
11         if $AP^{c'} \notin$ visited cells $\cup$ working set then
12             push $AP^{c'}$ onto working set
13         end
14     end
15     push $AP^c$ onto visited cells
16 end
17 return input cells
```

## V. Reachability

### A. Forward Reachability

The forward reachable set of a PWA function over some input set is simply the union of forward reachable sets of each individual polyhedron under its associated affine map. The image of a polyhedron under an affine map is

$$P_{out} = \{\mathbf{y} | \mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{d}, \mathbf{A}\mathbf{x} \leq \mathbf{b}\}. \tag{15}$$

For $\mathbf{C}$ invertible, the H-representation of the image is

$$P_{out} = \{\mathbf{y} | \mathbf{A}\mathbf{C}^{-1}\mathbf{y} \leq \mathbf{b} + \mathbf{A}\mathbf{C}^{-1}\mathbf{d}\}. \tag{16}$$

In the case the affine map is not invertible, more general polyhedral projection methods such as block elimination, Fourier-Motzkin elimination, or parametric linear programming can compute the H-representation of the image [38], [39]. Our implementation uses the block elimination projection in the case of a non-invertible affine map [40].

Our RPM algorithm is used to perform forward reachability as follows. We first specify a polyhedral input set whose image through the ReLU network we want to compute. This is a set over which to perform the RPM algorithm. For each activation pattern $AP^c$ we also compute the image of $H^c_{rep}$ under the map $\mathbf{C}^c\mathbf{x} + \mathbf{d}^c$. To do this, an additional line is introduced between lines 9 and 10 of Algorithm 2

$$H^c_{rep,\ forward} \leftarrow project(H^c_{rep}, \mathbf{C}^c, \mathbf{d}^c) \tag{17}$$

where $project(\cdot,\cdot,\cdot)$ applies (16) if $\mathbf{C}^c$ invertible and block elimination otherwise.

### B. Backward Reachability

The preimage of a polyhedron under an affine map is

$$P_{in} = \{\mathbf{x} | \mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{d}, \mathbf{A}\mathbf{y} \leq \mathbf{b}\} \tag{18a}$$

$$P_{in} = \{\mathbf{x} | \mathbf{A}\mathbf{C}\mathbf{x} \leq \mathbf{b} - \mathbf{A}\mathbf{d}\}. \tag{18b}$$

Like forward reachability, performing backward reachability only requires a small modification to Algorithm 2. We first specify a polyhedral output set whose preimage we would like to compute. For each activation pattern $AP^c$, we also compute the intersection of $H^c_{rep}$ with the preimage of the given output set under the map $\mathbf{C}^c\mathbf{x} + \mathbf{d}^c$. Two additional lines are thus introduced between lines 9 and 10 of Algorithm 2

$$P^c_{in} \leftarrow \text{Equation } 18b \tag{19a}$$

$$H^c_{rep,\ backward} \leftarrow H^c_{rep} \cap P^c_{in}. \tag{19b}$$

Multiple backward reachable sets can be solved for simultaneously at the added cost of repeating (19a) and (19b) for each output set argument to Algorithm 2.

Finally, we address the issue of finding forward and backward reachable sets iterated over multiple time steps. For this, we note that a ReLU network that is applied iteratively over $T$ timesteps, $\mathbf{x}_{t+1} = F(\mathbf{x}_t)$ for $t = 0,\dots,T-1$, is mathematically equivalent to a single ReLU network consisting of $T$ copies of the original network concatenated end to end, $\mathbf{x}_T = F_T(\mathbf{x}_0) := F \circ \cdots \circ F(\mathbf{x}_0)$. If the original network $F(\mathbf{x})$ has $N$ neurons and $L$ layers, the equivalent multi-time step network $F_T(\mathbf{x})$ has $TN$ neurons and $TL$ layers. We simply perform RPM for forward or backward reachability on the equivalent multi-time step network $F_T(\mathbf{x})$.

## VI. Examples

All examples are run on a 2013 Dell Latitude E6430s laptop with Intel Core i7 3GHz processor and 16GB of RAM. The cell coloring in plots is random.

### A. Damped Pendulum Example

The forward and backward reachability algorithms can be applied to discrete-time dynamical systems represented as ReLU networks. In this example we analyze a ReLU network that approximates the discrete-time dynamics of a damped pendulum. The function learned by the network is

$$\boldsymbol{x}_{t+1} = f_{nn}(\boldsymbol{x}_t) \tag{20}$$

where $\boldsymbol{x} = [\theta, \dot{\theta}]^\top$. The time-step used is 0.1 seconds. The learned dynamics function $f_{nn}$ is a ReLU network with a single hidden layer of 12 neurons. An example trajectory of the learned system is shown in Figure 3. We can compose multiple copies of $f_{nn}$ together to get a neural network which outputs an arbitrary future state. For instance, composing the network with itself 50 times results in a final network with 600 neurons and output $\boldsymbol{x}_{t+50}$.

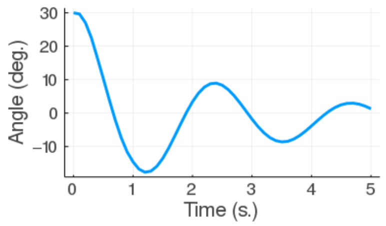
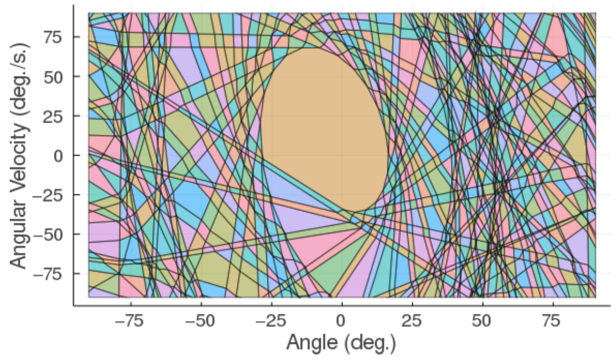
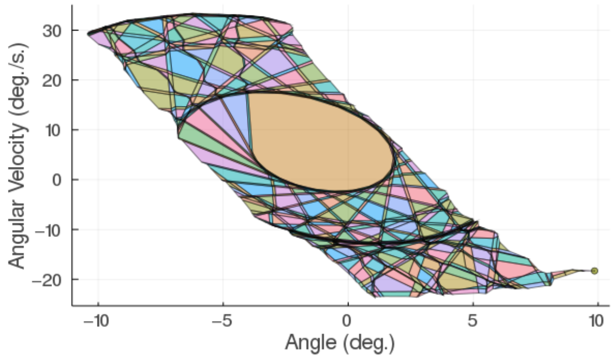


Fig. 3: 50 time-step trajectory from $\boldsymbol{x}_0 = [30,0]^\top$

Figure 3 suggests that states of the learned system tend toward the origin. We can use the forward reachability algorithm to determine where *all* states within some set of initial conditions lead to over some time interval. In Figure



(a) Input cells.



(b) Output cells.

Fig. 4: (a) Input cells of the 50 time-step pendulum network. (b) image of input cells under the neural network map.

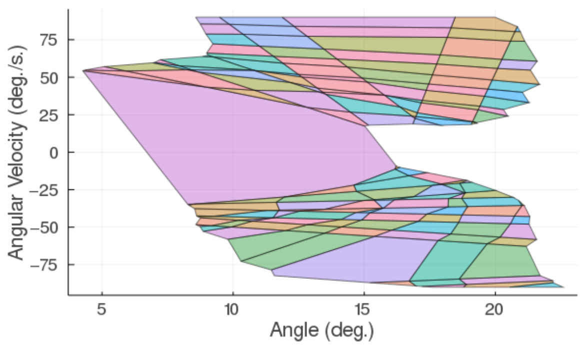


Fig. 5: Set of initial states that lead to a neighborhood around the origin after 50 time steps for the pendulum network.

4a we show the result of applying the cell enumeration algorithm and the forward reachability algorithm to the 50 time step system for initial conditions $-90^\circ \leq \theta \leq 90^\circ$ and $-90^\circ/s \leq \dot{\theta} \leq 90^\circ/s$. The resulting number of regions is 2781 and the forward reachability version of the RPM algorithm took 87s to complete. It is clear from Figure 4a that all initial conditions are mapped to a subset of the initial input set, proving that the original input set is forward invariant over 50 time step increments, and indicating that trajectories tend to approach the origin over time.

Further, we perform backward reachability to find the set of inputs that map to a small neighborhood around the origin after 50 time steps. Figure 5 shows the inputs that map to the output set $-5^\circ \leq \theta \leq 5^\circ$ and $-2^\circ/s \leq \dot{\theta} \leq 2^\circ/s$ after 50 time steps. Interestingly, only initial conditions with positive angles map to the target set. In the true dynamics, the initial angles would be symmetric about zero. This analysis can thus help us identify undesirable modeling artifacts to inform retraining or control of the model.

## B. Aircraft Collision Avoidance Example

The ACAS Xu networks are 45 distinct policy networks designed to issue advisory warnings to avoid mid-air collisions for unmanned aircarft. Each network has five inputs, five outputs (advisories), and 300 neurons. Inputs are relative distance, angles, and speeds. See [41] for more details.

The appendix of [18] lists ten safety properties the networks should satisfy. Here we consider Property 3. This property is satisfied if the network never outputs a “Clear of Conflict” advisory when the intruder is directly ahead and moving towards the ownship. We use the backward reachability algorithm to verify whether Property 3 is satisfied. A nonempty backward reachable set implies that some allowable inputs will map to unsafe outputs and the safety property is not satisfied. Table I shows the results of verifying Property 3 for four of the networks. A comparison is given against the fastest existing exact reachability method based on the face lattice representation [3] as well as the Reluplex and Marabou verification algorithms [18], [42]. Each algorithm was run on a single core. We note that a parallelized algorithm is also provided for the face lattice approach [3], and our RPM algorithm is also amenable to a parallelized implementation to be explored in future work.

TABLE I: ACAS Xu Property 3 Verification Results

| Network | Result | Time (s) RPM | Time (s) Face Lattice | Time (s) Reluplex | Time (s) Marabou |
|---|---|---|---|---|---|
| $N_{1,7}$ | Unsafe | **0.01** | 6.66 | 2.15 | 0.70 |
| $N_{1,8}$ | Unsafe | **0.01** | 5.45 | 4.32 | 1.49 |
| $N_{3,8}$ | Safe | 19.72 | **9.35** | 231.28 | 40.13 |
| $N_{5,6}$ | Safe | 31.92 | **17.28** | 366.39 | 54.07 |

The results in Table I show the advantage of our proposed method when verifying network properties that are found to not hold. Our RPM algorithm is anytime, in the sense that once an unsafe input polyhedron is found, the algorithm can terminate without completing the full reachability computation. This is in contrast to all existing exact reachability methods that proceed layer-by-layer through the network. They must solve the entire backward reachability problem to conclude any verification result, whether safe or unsafe.

## VII. Conclusion

We proposed the Reachable Polyhedral Marching (RPM) algorithm to efficiently construct the exact PWA representation of a ReLU network. RPM computes an explicit PWA function representation for a given ReLU network. This PWA function can then be used to quickly find forward and backward reachable sets over multiple time steps. Solving for multiple backward reachable sets can be done simultaneously at little added cost. RPM is shown to be especially fast when searching for the existence of unsafe inputs during verification. In the future, we will investigate parallel implementations of RPM to further improve computational speed.

## REFERENCES

[1] W. Xiang, H.-D. Tran, and T. T. Johnson, "Reachable set computation and safety verification for neural networks with relu activations," 2017.

[2] H.-D. Tran, D. M. Lopez, P. Musau, X. Yang, L. V. Nguyen, W. Xiang, and T. T. Johnson, "Star-based reachability analysis of deep neural networks," in *International Symposium on Formal Methods*. Springer, 2019, pp. 670–686.

[3] X. Yang, H.-D. Tran, W. Xiang, and T. Johnson, "Reachability analysis for feed-forward neural networks using face lattices," 2020.

[4] J. A. Sethian, "A fast marching level set method for monotonically advancing fronts," *Proceedings of the National Academy of Sciences*, vol. 93, no. 4, pp. 1591–1595, 1996.

[5] S. Garrido, L. Moreno, M. Abderrahim, and F. Martin, "Path planning for mobile robot navigation using voronoi diagram and fast marching," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2006, pp. 2376–2381.

[6] L. Janson, E. Schmerling, A. Clark, and M. Pavone, "Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions," *The International journal of robotics research*, vol. 34, no. 7, pp. 883–921, 2015.

[7] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *ACM siggraph computer graphics*, vol. 21, no. 4, pp. 163–169, 1987.

[8] J. Lei and K. Jia, "Analytic marching: An analytic meshing solution from deep implicit surface networks," 2020.

[9] G. F. Montufar, R. Pascanu, K. Cho, and Y. Bengio, "On the number of linear regions of deep neural networks," in *Advances in neural information processing systems*, 2014, pp. 2924–2932.

[10] B. Hanin, "Universal function approximation by deep neural nets with bounded width and relu activations," *Mathematics*, vol. 7, no. 10, p. 992, 2019.

[11] R. Arora, A. Basu, P. Mianjy, and A. Mukherjee, "Understanding deep neural networks with rectified linear units," in *International Conference on Learning Representations*, 2018.

[12] J. He, L. Li, J. Xu, and C. Zheng, "Relu deep neural networks and linear finite elements," *Journal of Computational Mathematics*, 2019.

[13] D. Rolnick and K. P. Kording, "Reverse-engineering deep relu networks," 2020.

[14] J. Z. Kolter and G. Manek, "Learning stable deep dynamics models," in *Advances in Neural Information Processing Systems*, 2019, pp. 11 128–11 136.

[15] W. Jin, Z. Wang, Z. Yang, and S. Mou, "Neural certificates for safe control policies," 2020.

[16] X. Lin, H. Zhu, R. Samanta, and S. Jagannathan, "Art: Abstraction refinement-guided training for provably correct neural networks," 2020.

[17] S. Mell, O. Brown, J. Goodwin, and S.-H. Son, "Safe predictors for enforcing input-output specifications," 2020.

[18] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient smt solver for verifying deep neural networks," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 97–117.

[19] C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. J. Kochenderfer, "Algorithms for verifying deep neural networks," 2019.

[20] W. Xiang, H.-D. Tran, and T. T. Johnson, "Output reachable set estimation and verification for multilayer neural networks," *IEEE transactions on neural networks and learning systems*, vol. 29, no. 11, pp. 5777–5783, 2018.

[21] T.-W. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, L. Daniel, and I. Dhillon, "Towards fast computation of certified robustness for relu networks," in *International Conference on Machine Learning (ICML)*, 2018.

[22] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal security analysis of neural networks using symbolic intervals," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1599–1614. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi

[23] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Efficient formal safety analysis of neural networks," in *Advances in Neural Information Processing Systems*, 2018, pp. 6367–6377.

[24] H. Zhang, T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, and L. Daniel, "Efficient neural network robustness certification with general activation functions," in *Advances in neural information processing systems*, 2018, pp. 4939–4948.

[25] S. Dutta, S. Jha, S. Sanakaranarayanan, and A. Tiwari, "Output range analysis for deep neural networks," 2017.

[26] P. Kouvaros and A. Lomuscio, "Formal verification of cnn-based perception systems," 2018.

[27] A. Lomuscio and L. Maganti, "An approach to reachability analysis for feed-forward relu neural networks," 2017.

[28] W. Ruan, X. Huang, and M. Kwiatkowska, "Reachability analysis of deep neural networks with provable guarantees," *arXiv preprint arXiv:1805.02242*, 2018.

[29] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Verisig: verifying safety properties of hybrid systems with neural network controllers," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, 2019, pp. 169–178.

[30] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–30, 01 2019.

[31] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Ai2: Safety and robustness certification of neural networks with abstract interpretation," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 3–18.

[32] K. D. Julian and M. J. Kochenderfer, "A reachability method for verifying dynamical systems with deep neural network controllers," *arXiv preprint arXiv:1903.00520*, 2019.

[33] H. Robinson, A. Rasheed, and O. San, "Dissecting deep neural networks," 2020.

[34] B. Hanin and D. Rolnick, "Complexity of linear regions in deep networks," in *International Conference on Machine Learning*, 2019, pp. 2596–2604.

[35] B. Hanin and D. Rolnick, "Deep relu networks have surprisingly few activation patterns," in *Advances in Neural Information Processing Systems*, 2019, pp. 361–370.

[36] R. Suard, J. Lofberg, P. Grieder, M. Kvasnica, and M. Morari, "Efficient computation of controller partitions in multi-parametric programming," in *2004 43rd IEEE Conference on Decision and Control (CDC) (IEEE Cat. No.04CH37601)*, vol. 4, 2004, pp. 3643–3648 Vol.4.

[37] I. Dunning, J. Huchette, and M. Lubin, "Jump: A modeling language for mathematical optimization," *SIAM Review*, vol. 59, no. 2, pp. 295–320, 2017.

[38] K. Fukuda, "Cddlib reference manual," *Report version 093a, McGill University, Montréal, Quebec, Canada*, 2003.

[39] A. Maréchal, D. Monniaux, and M. Périn, "Scalable minimizing-operators on polyhedra via parametric linear programming," in *International Static Analysis Symposium*. Springer, 2017, pp. 212–231.

[40] B. Legat, R. Deits, M. Forets, D. Oyama, S. Timme, F. Pacaud, S. Guadalupe, M. Besançon, J. TagBot, and E. Saba, "Juliapolyhedra/cddlib.jl: v0.6.1," Mar. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3733590

[41] K. D. Julian, J. Lopez, J. S. Brush, M. P. Owen, and M. J. Kochenderfer, "Policy compression for aircraft collision avoidance systems," in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. IEEE, 2016, pp. 1–10.

[42] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, *et al.*, "The marabou framework for verification and analysis of deep neural networks," in *International Conference on Computer Aided Verification*. Springer, 2019, pp. 443–452.