

# Algorithms for Verifying Deep Neural Networks

Changliu Liu

Carnegie Mellon University, Pittsburgh, PA 15213

CLIU6@ANDREW.CMU.EDU

Tomer Arnon

TARNON@STANFORD.EDU

Christopher Lazarus

CLAZARUS@STANFORD.EDU

Clark Barrett

BARRETT@CS.STANFORD.EDU

Mykel J. Kochenderfer

MYKEL@STANFORD.EDU

Stanford University, Stanford, CA 94305

Deep neural networks are widely used for nonlinear function approximation with applications ranging from computer vision to control. Although these networks involve the composition of simple arithmetic operations, it can be very challenging to verify whether a particular network satisfies certain input-output properties. This article surveys methods that have emerged recently for soundly verifying such properties. These methods borrow insights from reachability analysis, optimization, and search. We discuss fundamental differences and connections between existing algorithms. In addition, we provide pedagogical implementations of existing methods and compare them on a set of benchmark problems.

## 1 Introduction

Neural networks [15] have been widely used in many applications, such as image classification and understanding [17], language processing [24], and control of autonomous systems [26]. These networks represent functions that map inputs to outputs through a sequence of layers. At each layer, the input to that layer undergoes an affine transformation followed by a simple nonlinear transformation before being passed to the next layer. These nonlinear transformations are often called *activation functions*, and a common example is the *rectified linear unit* (ReLU), which transforms the input by setting any negative values to zero. Although the computation involved in a neural network is quite simple, these networks can represent complex nonlinear functions by appropriately choosing the matrices that define the affine transformations. The matrices are often learned from data using a form of stochastic gradient descent.

Neural networks are being used for increasingly important tasks, and in some cases, incorrect outputs can lead to costly consequences. Traditionally, validation of neural networks has largely focused on evaluating the network on a large collection of points in the input space and determining whether the outputs are as desired. However, since the input space is effectively infinite in cardinality, it is not feasible to check all possible inputs. Even networks that perform well on a large sample of inputs may not correctly generalize to new situations and may be vulnerable to adversarial attacks [29].

This article surveys a class of methods that are capable of formally verifying properties of deep neural networks over the full input space. A property can be formulated as a statement that if the input belongs to some set  $\mathcal{X}$ , then the output will belong to some set  $\mathcal{Y}$ . To illustrate, in classification problems, it can be useful to verify that points near a training example belong to the same class as that example. In the control of physical problems, it can be useful to verify that the outputs from a network satisfy hard safety constraints.

The verification algorithms that we survey are *sound*, meaning that they will only report that a property holds if the property actually holds. Some of the algorithms that we discuss are also *complete*, meaning that whenever the property holds, the algorithm will correctly

state that it holds. However, some of the algorithms compromise completeness in their use of approximations to improve computational efficiency.

The algorithms may be classified based on whether they draw insights from these three categories of analysis:

1. *Reachability*. These methods use layer-by-layer reachability analysis of the network. Representative methods are ExactReach [46], MaxSens [44], and Ai2 [14]. Some other approaches also use reachability methods (such as interval arithmetic) to compute the bounds on the values of the nodes.
2. *Optimization*. These methods use optimization to falsify the assertion. The function represented by the neural network is a constraint to be considered in the optimization. As a result, the optimization problem is not convex. In *primal optimization*, different methods are developed to encode the nonlinear activation functions as linear constraints. Examples include NSVerify [23], MIPVerify [38], and ILP [4]. The constraints can also be simplified through *dual optimization*. Representative methods for dual optimization are Duality [12], ConvDual [43], and Certify [31].
3. *Search*. These methods search for a case to falsify the assertion. Search is usually combined with either reachability or optimization, as the latter two methods provide possible search directions. Representative methods for *search and reachability* are ReluVal [40], DLV [18], FastLin [41], and Fast-Lip [41]. Representative methods for *search and optimization* are Reluplex [20], Planet [13], BaB [7], and Sherlock [10]. Some of these methods call Boolean satisfiability (SAT) or satisfiability modulo theories (SMT) solvers [3] to verify networks with only ReLU activations.

**Scope of this article.** This article introduces a unified mathematical framework for verifying neural networks, classifies existing methods under this framework, provides pedagogical implementations of existing methods,<sup>1</sup> and compares those methods on a set of benchmark problems.<sup>2</sup>

The following topics are not included in the discussion:

- neural network testing methods that generate better test cases [16], [30], [36], [37];
- white box approaches that build meaningful mappings from network parameters to some functional description [28];
- verification of binarized neural networks [8], [9], [27];
- closed-loop safety, stability and robustness by executing control policies defined by neural networks [11], [45], or verification of recurrent neural networks [2];
- training or retraining methods to make a network satisfy a property [25], [31], [43];
- robustness of the verification algorithm under floating point arithmetic [33].

Section 2 discusses the mathematical problem for verification. Section 3 gives an overview of the categories of methods that we will consider. Section 4 introduces preliminary and background mathematics. Section 5 discusses reachability methods. Section 6 discusses methods for primal optimization. Section 7 discusses methods for dual optimization. Section 8 discusses methods for search and reachability. Section 9 discusses methods for search and optimization. Section 10 compares those methods. Section 11 concludes the article.

<sup>1</sup> Our implementation is provided in the Julia programming language. We have found the language to be ideal for specifying in human readable form [5]. The full implementation may be found at <https://github.com/sisl/NeuralVerification.jl>.

<sup>2</sup> There have been other reviews of methods for verifying neural networks. Leofante et al. review primal optimization methods that encode ReLU networks as mixed integer programming problems together with search and optimization under the framework of Boolean satisfiability and SMT [22]. Xiang et al. review a broader range of verification techniques in addition to safe control and learning [48]. Salman et al. review and compare methods that use convex relaxations to compute robustness bounds of ReLU networks [32].

## 2 Problem Formulation

We first review feedforward neural networks and introduce the mathematical formulation of the verification problem. We will then discuss the results provided by various algorithms along with the properties of soundness and completeness. In our discussion, we will use lowercase letters in italics for scalars and scalar functions ( $x$ ), lowercase letters in upright bold for vectors and vector functions ( $\mathbf{x}$ ), uppercase letters in upright bold for matrices and matrix functions ( $\mathbf{X}$ ), and calligraphic uppercase letters for sets and set functions ( $\mathcal{X}$ ).

### 2.1 Feedforward Neural Network

Consider an  $n$ -layer *feedforward neural network* that represents a function  $\mathbf{f}$  with input  $\mathbf{x} \in \mathcal{D}_{\mathbf{x}} \subseteq \mathbb{R}^{k_0}$  and output  $\mathbf{y} \in \mathcal{D}_{\mathbf{y}} \subseteq \mathbb{R}^{k_n}$ , i.e.,  $\mathbf{y} = \mathbf{f}(\mathbf{x})$ , where  $k_0$  is the input dimension and  $k_n$  is the output dimension. All non-vector inputs or outputs are reshaped to vectors. Each layer in  $\mathbf{f}$  corresponds to a function  $\mathbf{f}_i : \mathbb{R}^{k_{i-1}} \rightarrow \mathbb{R}^{k_i}$ , where  $k_i$  is the dimension of the hidden variable  $\mathbf{z}_i$  in layer  $i$ . Moreover, we set  $\mathbf{z}_0 = \mathbf{x}$  and  $\mathbf{z}_n = \mathbf{y}$ . Hence, the network can be represented by

$$\mathbf{f} = \mathbf{f}_n \circ \mathbf{f}_{n-1} \circ \cdots \circ \mathbf{f}_1, \quad (1)$$

where  $\circ$  means function composition. The function at layer  $i$  is

$$\mathbf{z}_i = \mathbf{f}_i(\mathbf{z}_{i-1}) = \sigma_i(\mathbf{W}_i \mathbf{z}_{i-1} + \mathbf{b}_i), \quad (2)$$

which consists of a linear transformation defined by a weight matrix  $\mathbf{W}_i \in \mathbb{R}^{k_i \times k_{i-1}}$  and a bias vector  $\mathbf{b}_i \in \mathbb{R}^{k_i}$ , and an activation function  $\sigma_i : \mathbb{R}^{k_i} \rightarrow \mathbb{R}^{k_i}$ . All activation functions are assumed to be monotone and non-decreasing. For simplicity, let  $\hat{\mathbf{z}}_i := \mathbf{W}_i \mathbf{z}_{i-1} + \mathbf{b}_i$  denote the node value before activation. Let  $z_{i,j}$  be the value of the  $j$ th node in the  $i$ th layer,  $\mathbf{w}_{i,j} \in \mathbb{R}^{1 \times k_{i-1}}$  be the  $j$ th row in  $\mathbf{W}_i$ ,  $w_{i,j,k}$  be the  $k$ th entry in  $\mathbf{w}_{i,j}$ ,  $b_{i,j}$  be the  $j$ th entry in  $\mathbf{b}_i$ . In the case that the activation function is node-wise, we denote the activation for the  $j$ th node as  $\sigma_{i,j}$ . Then

$$z_{i,j} = \sigma_{i,j}(\mathbf{w}_{i,j} \mathbf{z}_{i-1} + b_{i,j}) = \sigma_{i,j} \left( \sum_k w_{i,j,k} z_{i-1,k} + b_{i,j} \right) = \sigma_{i,j}(\hat{z}_{i,j}). \quad (3)$$

Figure 1 shows the structure of a feedforward neural network.

A network is defined in algorithm 2.1. The definitions of different activation functions are listed in algorithm 2.2.

### 2.2 Verification Problem

Verification involves checking whether input-output relationships of a function hold. The input constraint is imposed by a set  $\mathcal{X} \subseteq \mathcal{D}_{\mathbf{x}}$ . The corresponding output constraint is imposed by a set  $\mathcal{Y} \subseteq \mathcal{D}_{\mathbf{y}}$ . In the following discussion, we call the sets  $\mathcal{X}$  and  $\mathcal{Y}$  *constraints*. Solving the *verification problem* requires showing that the following assertion holds:

$$\mathbf{x} \in \mathcal{X} \Rightarrow \mathbf{y} = \mathbf{f}(\mathbf{x}) \in \mathcal{Y}. \quad (4)$$

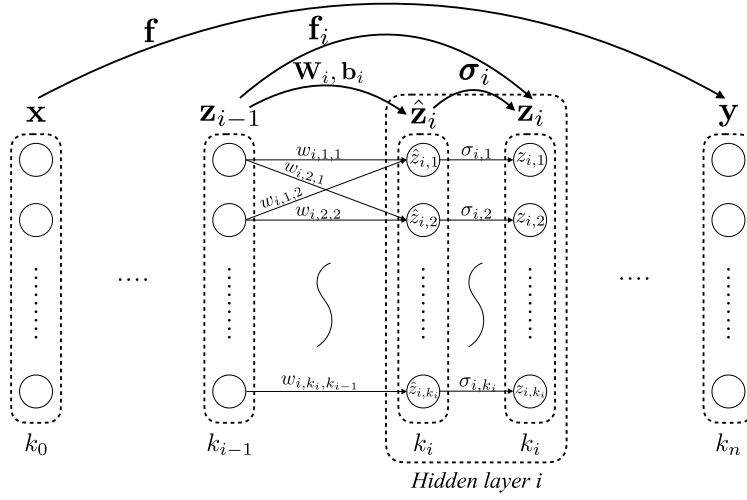


Figure 1. Illustration of a feedforward neural network and the notations used in this article.

```

abstract type ActivationFunction end

struct Layer{F<:ActivationFunction, N<:Number}
  weights::Matrix{N}
  bias::Vector{N}
  activation::F
end

struct Network
  layers::Vector{Layer}
end

```

Algorithm 2.1. Network structure. A network is a list of layers. Each layer consists of weights  $\mathbf{W}$ , bias  $\mathbf{b}$ , and activation  $\sigma$ .

```

struct GeneralAct <: ActivationFunction end
struct ReLU <: ActivationFunction end
struct Id <: ActivationFunction end
(f::GeneralAct)(x) = f(x)
(f::ReLU)(x) = max.(x, 0)
(f::Id)(x) = x

```

Algorithm 2.2. Activation functions. This paper mostly focuses on networks with ReLU activations. Other piece-wise linear activations can be encoded by multiple ReLU activations. For example, the Max activation function can be encoded using the following relationship:  $\text{Max}(x, y) = \text{ReLU}(x - y) + y$ . Composing multiple Max functions can enable the encoding of the Max pooling operator.

For example, to verify robustness in a classification network,<sup>3</sup> we need to ensure that all samples in the neighborhood of a given input  $\mathbf{x}_0$  are classified with the same label. Suppose the desired label is  $i^* \in \{1, \dots, k_n\}$ . We need to ensure that  $y_{i^*} > y_j$  for all  $j \neq i^*$ . The input and output constraints are

$$\mathcal{X} = \{\mathbf{x} : \|\mathbf{x} - \mathbf{x}_0\|_p \leq \epsilon\}, \quad (5a)$$

$$\mathcal{Y} = \{\mathbf{y} : y_{i^*} > y_j, \forall j \neq i^*\}, \quad (5b)$$

where  $\epsilon$  is the maximum allowable disturbance in the input space. The metric to measure disturbance can be any  $\ell_p$  norm, though the  $\ell_\infty$  or the  $\ell_1$  norms are common because they lead to linear constraints.

Our formulation is broader than classification problems. In general, the input set  $\mathcal{X}$  and the output set  $\mathcal{Y}$  can have any geometry. For simplicity, we assume that  $\mathcal{X}$  is a polytope, and  $\mathcal{Y}$  is either a polytope or the complement of a polytope. A *polytope* is a generalization of the three-dimensional polyhedron and is defined as the intersection of a set of half-spaces.<sup>4</sup> Since any compact domain can be approximated by a finite set of polytopes for any required accuracy, this formulation can be easily extended to arbitrary geometries. Moreover, the complement of a polytope allows the encoding of unbounded sets. By default, a polytope is a closed set, while its complement is an open set.

<sup>3</sup> Given an input, a classification network outputs weights over several labels. The input is assigned the label with the highest weight.

<sup>4</sup> This is the definition of a *convex polytope*; alternative definitions exist, but this is the one we will use here.

<sup>5</sup> In our implementations, we use the definitions in `LazySets.jl`, which is a Julia package for calculus with convex sets [6]. The implementation can be found at <https://github.com/JuliaReach/LazySets.jl>.

Our discussion will focus on the following four subclasses of polytopes:<sup>5</sup>

- *Halfspace-polytope* (or *H-polytope*), which represents polytopes using a set of linear inequality constraints

$$\mathbf{C}\mathbf{x} \leq \mathbf{d}, \quad (6)$$

where  $\mathbf{C} \in \mathbb{R}^{k \times k_0}$ ,  $\mathbf{d} \in \mathbb{R}^k$ , and  $k$  is the number of inequality constraints that are defining the polytope. A point  $\mathbf{x}$  is in the polytope if and only if  $\mathbf{C}\mathbf{x} \leq \mathbf{d}$  is satisfied.

- *Vertex-polytope* (or *V-polytope*), which represents polytopes using a set of vertices. Mathematically, it is described by a concatenation of all vertices  $\mathbf{v}_i$  for  $i \in \{1, \dots, k\}$ ,

$$[\mathbf{v}_1 \quad \mathbf{v}_2 \quad \dots \quad \mathbf{v}_k], \quad (7)$$

where  $k$  is the number of vertices. A point  $\mathbf{x}$  is in the polytope if and only if  $\mathbf{x}$  is in the convex hull of the vertices.

- *Hyperrectangle*, which corresponds to a high-dimensional rectangle, defined by

$$|\mathbf{x} - \mathbf{c}| \leq \mathbf{r}, \quad (8)$$

where  $\mathbf{c} \in \mathbb{R}^{k_0}$  is the center of the hyperrectangle and  $\mathbf{r} \in \mathbb{R}^{k_0}$  is the radius of the hyperrectangle.

- *Halfspace*, which is represented by a single linear inequality constraint

$$\mathbf{c}^\top \mathbf{x} \leq d, \quad (9)$$

where  $\mathbf{c} \in \mathbb{R}^{k_0}$  and  $d \in \mathbb{R}$ .

In our discussion, we may refer to hyperrectangles as *intervals*. A hyperrectangle that has uniform side lengths is called a *hypercube*. The set in (5b) corresponds to a halfspace-polytope. When  $p = \infty$ , the set in (5a) corresponds to a hyperrectangle centered at  $\mathbf{x}_0$  with uniform radius  $\epsilon$ .

The verification problem is defined in algorithm 2.3.

```

struct Problem{P, Q}
  network::Network
  input::P
  output::Q
end

```

Algorithm 2.3. Problem definition. It consists of a network to be verified, an input set constraint, and an output set constraint. The types P and Q can be any sets that match the requirements of the algorithm.

### 2.3 Results

Verification algorithms attempt to identify whether (4) holds. In some cases, algorithms may return *unknown* if no conclusion can be drawn. Different algorithms output different types of results as listed below and illustrated in figure 2.

- *Counter example result*, which is a counter example  $x^* \in \mathcal{X}$  with

$$f(x^*) \notin \mathcal{Y}. \quad (10)$$

The property (4) is violated if such a counter example is found.

- *Adversarial result*, which is the maximum allowable disturbance with respect to an  $\ell_p$  norm while maintaining  $f(x) \in \mathcal{Y}$ :

$$\epsilon(x_0, f, \mathcal{Y}, p) := \min_{x, \text{ s.t. } f(x) \in \mathcal{Y}} \|x - x_0\|_p. \quad (11)$$

The property (4) is violated if the input set  $\mathcal{X}$  exceeds the maximum allowable disturbance,

$$\epsilon(x_0, f, \mathcal{Y}, p) < \max_{x \in \mathcal{X}} \|x - x_0\|_p. \quad (12)$$

- *Reachability result*, which is the output reachable set:

$$\mathcal{R}(\mathcal{X}, f) := \{y : y = f(x), \forall x \in \mathcal{X}\}. \quad (13)$$

The property (4) is violated if the reachable set does not belong to the output set  $\mathcal{Y}$ ,

$$\mathcal{R}(\mathcal{X}, f) \not\subseteq \mathcal{Y}. \quad (14)$$

Algorithm 2.4 provides definitions of these result types used in our implementation. The status may be `:holds`, `:violated`, or `:Unknown`.

### 2.4 Soundness and Completeness

The result returned by a particular solver may not always be correct. A specific instance of (4) can either *hold* or be *violated*. The status from a solver can be holds, violated, or unknown. Ideally, a solver only outputs holds or violated to match the actual status of a given problem. However, some algorithms make approximations that can result in a mismatch. For example, the computed reachable set (denoted  $\tilde{\mathcal{R}}$ ) may be an over-approximation of  $\mathcal{R}$  in (13). Then, even if  $\tilde{\mathcal{R}} \not\subseteq \mathcal{Y}$ , i.e., the solver returns violated, it is possible that  $\mathcal{R} \subset \mathcal{Y}$ , i.e., the property actually holds.

We use the following definitions to categorize solvers:

- *Soundness*, which requires that when the solver returns holds, the property actually holds.

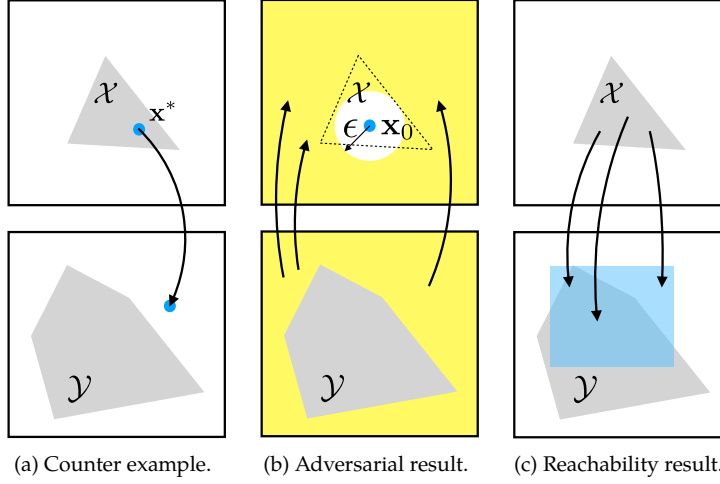


Figure 2. Illustration of different results. The upper square represents the input domain. The lower square represents the output domain. The input set  $\mathcal{X}$  and the output set  $\mathcal{Y}$  are shown as gray polygons. In (a), a counter example is found. In (b), the input set exceeds the maximum allowable disturbance (white circle in the input domain). In (c), the output reachable set (blue set in the output domain) does not belong to the output set.

```

abstract type Result end

struct BasicResult <: Result
  status::Symbol
end

struct CounterExampleResult <: Result
  status::Symbol
  counter_example::Vector{Float64}
end

struct AdversarialResult <: Result
  status::Symbol
  max_disturbance::Float64
end

struct ReachabilityResult <: Result
  status::Symbol
  reachable::Vector{<:AbstractPolytope}
end

```

Algorithm 2.4. Result types. `BasicResult` is for satisfiability only. `CounterExampleResult` also outputs a counter example if the problem is not satisfied. `AdversarialResult` outputs the maximum allowable disturbance. `ReachabilityResult` outputs the reachable set.

- *Completeness*, which requires that (i) the solver never returns unknown; and (ii) if the solver returns violated, the property is actually violated.
- *Termination*, which requires that the solver always finishes after a finite number of steps.

A method that is sound, complete, and terminating always outputs the correct result with no unknowns. All methods discussed in this survey are sound and terminating, but not all of them are complete. Some methods use over-approximations to speed up computation and result in incompleteness. Our implementation may be slightly different from the original implementation of some methods. For example, the original implementation of DLV is always complete and is sound under the minimality assumption of the search tree. Our implementation is sound but not complete. The differences will not affect the key concepts of the methods. And we point out the differences in the detailed discussion of the methods.

Table 1 summarizes the characteristics of all methods considered in this survey. The input and output sets may include hyperrectangles (HR), halfspaces (HS), halfspace-polytopes (HP), vertex-polytopes (VP), and polytope complements (PC). Additionally, the superscripts in the table indicate the following constraints on the output sets:

1. The polytope has to be bounded, this restriction is not due to a theoretical limitation, but rather to our implementation and will eventually be relaxed.
2. Polytope complements encode unbounded sets. They are used in optimization-based methods, to be explained in section 6. These methods usually encode the complement of the output set as a constraint and require the constraint to be convex. The complement of a polytope complement is a convex polytope, hence satisfies the requirement.
3. The output set must be 1-dimensional.

Method Name	Activation	Approach	Input/Output	Completeness
ExactReach [46]	ReLU	Exact Reachability	HP/HP(bounded) <sup>1</sup>	✓
AI2 [14]	Piecewise Linear	Split and Join	HP/HP(bounded) <sup>1</sup>	×
MaxSens [44]	Any	Interval Arithmetic	HP/HP(bounded) <sup>1</sup>	×
NSVerify [23]	ReLU	Naive MILP	HR/PC <sup>2</sup>	✓
MIPVerify [38]	ReLU and Max	MILP with bounds	HR/PC <sup>2</sup>	✓
ILP [4]	ReLU	Iterative LP	HR/PC <sup>2</sup>	×
Duality [12]	Any	Lagrangian Relaxation	HR(uniform)/HS	×
ConvDual [43]	ReLU	Convex Relaxation	HR(uniform)/HS	×
Certify [31]	Differentiable	Semidefinite Relaxation	HR/HS	×
Fast-Lin [41]	ReLU	Network Relaxation	HR/HS	×
Fast-Lip [41]	ReLU	Lipschitz Estimation	HR/HS	×
ReluVal [40]	ReLU	Symbolic Interval	HR/HR	✓
DLV [18]	Any	Search in Hidden Layers	HR/HR(1-D) <sup>3</sup>	✓*
Sherlock [10]	ReLU	Local and Global Search	HR/HR(1-D) <sup>3</sup>	×
BaB [7]	Piecewise Linear	Branch and Bound	HR/HR(1-D) <sup>3</sup>	×
Planet [13]	Piecewise Linear	Satisfiability (SAT)	HR/PC <sup>2</sup>	✓
Reluplex [20]	ReLU	Simplex	HR/PC <sup>2</sup>	✓

Table 1. List of existing methods. We name the method if it does not have a name. The entries under “activation” show the type of activations supported in the methods. The entries under “approach” summarize the key ideas of the methods. All the methods presented in this paper are sound and the methods that are complete as defined in section 2.4 are marked in the “completeness” column. For DLV, the original implementation is complete but may not be sound, while our implementation is sound but not complete.



### 3 Overview of Methods

This section overviews existing methods that are studied in this survey. Their common components will be outlined. As mentioned earlier, there are three basic verification methods, *i.e.*, reachability, optimization, and search. Regarding the three basic methods, we categorize those methods into the following five categories as shown in figure 3. The methods are also summarized in table 1.

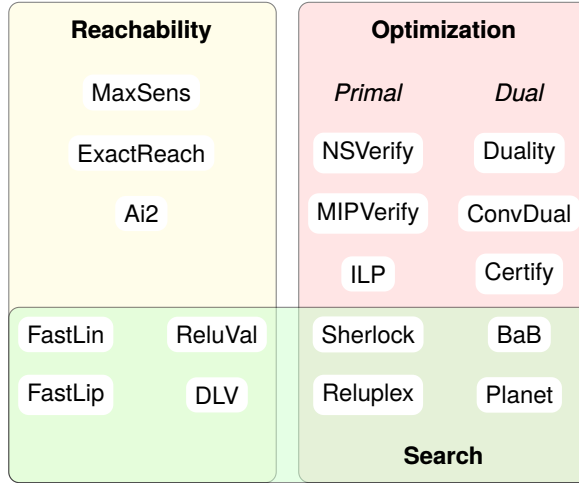


Figure 3. Overview of all methods in the survey. Given the three basic methods: reachability, optimization, and search, existing methods are divided into five categories: reachability, primal optimization, dual optimization, search and reachability, and search and optimization. Relationships among different methods are shown in the figure.

#### 3.1 Reachability

These methods perform layer-by-layer reachability analysis to compute the reachable set  $\mathcal{R}(\mathcal{X}, \mathbf{f})$ . They usually generate `ReachabilityResult`.

ExactReach [46] computes the exact reachable set for networks with only ReLU activations. The key insight is that if the input set to a ReLU function is a union of polytopes, then the output reachable set is also a union of polytopes. As there is no over-approximation, this method is sound and complete. However, because the number of polytopes grows exponentially with each layer, this method does not scale.

Ai2 [14] uses representations that over-approximate the reachable set. It trades precision of the reachable set for scalability of the algorithm. It works for any piecewise linear activation functions, such as ReLU and max pooling. Due to its approximation, the number of geometric objects to be traced during layer-by-layer analysis is greatly reduced. Though Ai2 is not complete, it scales well.

MaxSens [44] partitions the input domain into small grid cells, and loosely approximates the reachable set for each grid cell considering the maximum sensitivity of the network at each grid cell. Sensitivity of a function is equivalent to the Lipschitz constant of the function. The union of those reachable sets is the output reachable set. The finer the partition, the tighter the output reachable set. MaxSens is not complete. It works for any activation function and scales well.

### 3.2 Primal Optimization

Primal optimization methods try to falsify assertion (4). The network structure is a constraint to be considered in the optimization. Existing methods only work for ReLU activations. Different methods are developed to encode the network as a set of linear constraints or mixed integer linear constraints by exploiting the piecewise linearity in the ReLU. The encoding methods will be discussed in section 6.1. The return type can be either `CounterExampleResult` or `AdversarialResult`, depending on the objective of the optimization.

NSVerify [1], [23] encodes the network as a set of mixed integer linear constraints. It solves a feasibility problem without an objective function. It tries to find a counter example for the verification problem.<sup>6</sup> This method is sound and complete.

MIPVerify [38] also encodes the network as a set of mixed integer linear constraints. There are two differences between MIPVerify and NSVerify. First, MIPVerify determines the bounds on the nodes to tighten the constraints. Second, MIPVerify solves an adversarial problem that tries to estimate the maximum allowable disturbance on the input side. This method is also complete.

ILP (iterative linear programming) [4] encodes the network as a set of linear constraints by linearizing the network at a reference point. The optimization problem in ILP is an adversarial problem that tries to estimate the maximum allowable disturbance on the input side. It iteratively solves the optimization. This method is not complete as it only considers one linear segment of the network.

### 3.3 Dual Optimization

In primal optimization methods, different methods are developed to encode the constraints imposed by the network. We can also use dual optimization to simplify the constraints. In dual optimization, the constraints are much simpler than those in primal optimization. On the other hand, objectives in dual optimization, which correspond to the constraints in primal optimization, are much more complicated than those in the primal problem. Relaxations are usually involved during the construction of the dual problem. Due to relaxation, these approaches are incomplete. The return type is `BasicResult`.

Duality [12] solves a Lagrangian relaxation of the optimization problem to obtain bounds on the output. The dual problem is formulated as an unconstrained convex optimization problem, which can be computed efficiently. Duality works for any activation function.

ConvDual [43] also uses a dual approach to estimate the bounds on the output. It obtains a simplified dual problem by first making a convex relaxation of the network in the primal optimization. The bounds are heuristically computed by choosing a fixed, dual feasible solution, without any explicit optimization. In this way, ConvDual is more computationally efficient than Duality. In the original ConvDual approach, the bounds are then used to robustly train the network. This survey focuses on the method to compute the bounds.

Certify [31] uses a semidefinite relaxation to compute over approximated certificates (*i.e.*, bounds). It only works for neural networks with only one hidden layer. It works for any activation function that is differentiable almost everywhere.<sup>7</sup> In the original Certify approach, the certificates are then optimized jointly with network parameters to provide an adaptive regularizer that improves robustness of the network. This survey focuses on the method to obtain the certificates.

<sup>6</sup> NSVerify [1] is developed for verification of a closed-loop system that has neural network components. It has been extended to verify recurrent neural networks [2]. We only review the method used to verify non-recurrent neural networks, which was first discussed in [23].

<sup>7</sup> Differentiable a.e. means that the function is differentiable everywhere except for countably many points. Piecewise linear activation functions are all differentiable a.e.

### 3.4 Search and Reachability

Reachability methods need to balance computational efficiency and precision of the approximation. When reachability is combined with search, it is possible to improve both efficiency and accuracy. These methods usually search in the input or the hidden spaces for a counter example. However, due to over-approximation in reachability analysis, these methods are sound but incomplete.

ReluVal [39], [40] uses symbolic interval analysis along with search to minimize over-approximations of the output sets. During the search process, ReluVal iteratively bisects its input range. This process is called iterative interval refinement, which is also used in BaB [7].

Fast-Lin [41] computes a certified lower bound on the allowable input disturbance for ReLU networks using a layer-by-layer approach and binary search in the input domain.

Fast-Lip [41] depends on Fast-Lin to compute the bounds on the activation functions, and further estimates the local Lipchitz constant of the network. In general, Fast-Lin is more scalable than Fast-Lip, while Fast-Lip provides better solutions for  $\ell_1$  bounds.

DLV [18] searches for adversarial inputs layer by layer in the hidden layers. This is the only approach that searches in the hidden spaces we have seen so far. DLV supports any activation function.

### 3.5 Search and Optimization

Search can also be combined with optimization. We can either search in the input space, or search in the function space. Searching in the function space is done by exploring possible activation patterns. An activation pattern is an assignment (*e.g.*, on or off for ReLU) to each activation function in the network. These methods may use SAT or SMT.

Sherlock [10] estimates the output range using a combination of local search and global search. Local search solves a linear program to find local optima. Global search solves a mixed integer linear program to escape local optima, which is similar to the method in NSVerify and MIPVerify. Sherlock is incomplete.

BaB [7] uses branch and bound to compute the output bounds of a network. It has a modularized design that can serve as a unified framework that can support other methods such as Reluplex and Planet.

Planet [13] integrates with a SAT solver for tree search in the function space. The objective of the search is to find an activation pattern of ReLU networks that maps an input in  $\mathcal{X}$  to an output not in  $\mathcal{Y}$ . It combines optimization-based filtering and pruning in the search process. Planet is complete.

Reluplex [20] performs tree search in the function space. It extends the simplex algorithm, a standard algorithm for solving linear programming (LP) instances, to support ReLU networks. The algorithm is called Reluplex, for ReLU with the simplex method. Reluplex is complete.

## 4 Preliminaries

This section introduces additional notation and operations that will be used in different verification algorithms to be discussed in the following sections.<sup>8</sup> Section 4.1 discusses interval arithmetic to compute node-wise bounds given an input set. Such node-wise bounds are needed in many methods, such as MIPVerify, Duality, ConvDual, Planet, and Reluplex. Section 4.2 discusses interval refinement, which is used in ReluVal and BaB. Section 4.3 discusses methods

<sup>8</sup> First-time readers may skip this section and refer back when going into the details of the algorithms.

to compute network gradient and bounds on the gradient given a non-trivial input set. The bounds on the gradient are used in ReluVal and FastLin. Section 4.4 introduces specific notation for ReLU activations.

We use  $[a]_+ := \max\{a, 0\}$  and  $[a]_- = \min\{a, 0\}$  to represent the positive and negative parts of a scalar variable  $a$ . For a vector  $\mathbf{a}$  or a matrix  $\mathbf{A}$ ,  $[\cdot]_+$  and  $[\cdot]_-$  take element-wise max and min, respectively.

#### 4.1 Bounds

The lower and upper bounds on  $z_{i,j}$ , i.e., the value of node  $j$  at layer  $i$  after activation, are denoted  $\ell_{i,j}$  and  $u_{i,j}$ . The lower and upper bounds on  $\hat{z}_{i,j}$ , i.e., the value of node  $j$  at layer  $i$  before activation, are denoted  $\hat{\ell}_{i,j}$  and  $\hat{u}_{i,j}$ . The after-activation bounds for the whole layer  $i$  are denoted  $\ell_i$  and  $\mathbf{u}_i$ . The before-activation bounds for the whole layer  $i$  are denoted  $\hat{\ell}_i$  and  $\hat{\mathbf{u}}_i$ . The bounds can be computed using different methods. For example, MaxSens uses interval arithmetic, which will be discussed in section 5.4. Planet uses optimization to compute tight bounds, which will be discussed in section 9.3. Because the optimization is difficult to solve, ConvDual and FastLin relax the network constraints and analytically compute the bounds using dynamic programming, which will be discussed in section 7.3 and section 8.2. Here we introduce interval arithmetic.

*Interval arithmetic* By interval arithmetic, given the bounds at layer  $i - 1$ , the bounds at layer  $i$  satisfy

$$\hat{\ell}_{i,j} = \min_{\mathbf{z}_{i-1} \in [\ell_{i-1}, \mathbf{u}_{i-1}]} \mathbf{w}_{i,j} \mathbf{z}_{i-1} + b_{i,j} = [\mathbf{w}_{i,j}]_+ \ell_{i-1} + [\mathbf{w}_{i,j}]_- \mathbf{u}_{i-1} + b_{i,j}, \quad (15a)$$

$$\hat{u}_{i,j} = \max_{\mathbf{z}_{i-1} \in [\ell_{i-1}, \mathbf{u}_{i-1}]} \mathbf{w}_{i,j} \mathbf{z}_{i-1} + b_{i,j} = [\mathbf{w}_{i,j}]_+ \mathbf{u}_{i-1} + [\mathbf{w}_{i,j}]_- \ell_{i-1} + b_{i,j}, \quad (15b)$$

$$\ell_{i,j} = \min_{\hat{z}_{i,j} \in [\hat{\ell}_{i,j}, \hat{u}_{i,j}]} \sigma_{i,j}(\hat{z}_{i,j}) = \sigma_{i,j}(\hat{\ell}_{i,j}), \quad (15c)$$

$$u_{i,j} = \max_{\hat{z}_{i,j} \in [\hat{\ell}_{i,j}, \hat{u}_{i,j}]} \sigma_{i,j}(\hat{z}_{i,j}) = \sigma_{i,j}(\hat{u}_{i,j}), \quad (15d)$$

where the implicit assumption for the last two equalities is that the activation  $\sigma_{i,j}$  is non-decreasing. In the implementation, the bounds are usually stored as a list of hyperrectangles. The bounds with respect to the input constraint  $\mathcal{X}$  can be computed layer-by-layer using interval arithmetic (15), to be discussed in algorithm 5.5. In the following discussion, we write interval arithmetic with respect to linear mappings compactly as  $\otimes$  where

$$\mathbf{W} \otimes [\ell, \mathbf{u}] := [[\mathbf{W}]_+ \ell + [\mathbf{W}]_- \mathbf{u}, [\mathbf{W}]_+ \mathbf{u} + [\mathbf{W}]_- \ell], \quad (16)$$

where  $\ell$  and  $\mathbf{u}$  can also be replaced with matrices. The function (16) is implemented in algorithm 4.1. Hence,  $[\hat{\ell}_i, \hat{\mathbf{u}}_i] = \mathbf{W}_i \otimes [\ell_{i-1}, \mathbf{u}_{i-1}] + [\mathbf{b}_i, \mathbf{b}_i]$ .

```
function interval_map(W, l, u)
    l_new = max.(W, 0) * l + min.(W, 0) * u
    u_new = max.(W, 0) * u + min.(W, 0) * l
    return (l_new, u_new)
end
```

Algorithm 4.1. Function to compute linear mapping on intervals which corresponds to equation (16).

## 4.2 Interval Refinement

Interval refinement is used in many methods including ReluVal and BaB. In those cases, a high dimensional interval  $[\ell, \mathbf{u}]$  is split in two at index  $i^*$ :

$$[\ell, \mathbf{u}^*] \text{ and } [\ell^*, \mathbf{u}], \quad (17)$$

where  $\mathbf{u}^* = \mathbf{u} - r_{i^*} \mathbf{e}_{i^*}$ ,  $\ell^* = \ell + r_{i^*} \mathbf{e}_{i^*}$ ,  $r_{i^*} = \frac{1}{2}(u_{i^*} - \ell_{i^*})$ , and  $\mathbf{e}_i$  is a unit vector whose entries are all zero except at the  $i$ th entry. The index  $i^*$  is determined using different methods. For example, in BaB, the index  $i^*$  is chosen to be the longest dimension, *i.e.*,  $i^* = \arg \max_i (u_i - \ell_i)$ . The function to perform the split is implemented in algorithm 4.2. Figure 4 illustrates the split of a two dimensional interval.



Figure 4. Illustration of interval split. The high dimensional interval on the left is split into two high dimensional intervals on the right, with respect to the horizontal axis.

```

function split_interval(dom::Hyperrectangle, i::Int64)
    input_lower, input_upper = low(dom), high(dom)
    input_upper[i] = dom.center[i]
    input_split_left = Hyperrectangle(low = input_lower, high = input_upper)
    input_lower[i] = dom.center[i]
    input_upper[i] = dom.center[i] + dom.radius[i]
    input_split_right = Hyperrectangle(low = input_lower, high = input_upper)
    return (input_split_left, input_split_right)
end

```

Algorithm 4.2. Function to split interval at a specified index. The argument `dom` is the interval to be split. The argument `index` is the index  $i^*$  where the interval should be split at. The function returns the two hyperrectangles after the split, which correspond to equation (17).

## 4.3 Gradient

The gradient of a neural network (of the output with respect to the input) satisfies the chain rule,

$$\nabla \mathbf{f} := \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \hat{\mathbf{z}}_n} \frac{\partial \hat{\mathbf{z}}_n}{\partial \mathbf{z}_{n-1}} \cdots \frac{\partial \mathbf{z}_1}{\partial \hat{\mathbf{z}}_1} \frac{\partial \hat{\mathbf{z}}_1}{\partial \mathbf{x}} = \nabla \sigma_n \mathbf{W}_n \cdots \nabla \sigma_1 \mathbf{W}_1, \quad (18)$$

where  $\nabla \sigma_i := \frac{\partial \mathbf{z}_i}{\partial \hat{\mathbf{z}}_i} \in \mathbb{R}^{k_i \times k_i}$ .

In some cases, we evaluate the point-wise gradient  $\nabla \mathbf{f}(\mathbf{x}_0)$  for some  $\mathbf{x}_0$ . The point-wise gradient is easy to compute by following the chain rule as shown in the first function in algorithm 4.3. In other cases, *e.g.*, in ReluVal and FastLip, we need to compute the maximum gradient given an input set  $\mathcal{X}$ , *i.e.*,  $\max_{\mathbf{x} \in \mathcal{X}} \nabla \mathbf{f}(\mathbf{x})$ . The maximum over a vector is taken point-wise. The maximum gradient can be computed using interval arithmetic.

Denote the lower and upper bounds of  $\nabla \sigma_i$  with respect to the input set  $\mathcal{X}$  as  $\underline{\mathbf{A}}_i \in \mathbb{R}^{k_i \times k_i}$  and  $\overline{\mathbf{A}}_i \in \mathbb{R}^{k_i \times k_i}$ . The matrices  $\underline{\mathbf{A}}_i$  and  $\overline{\mathbf{A}}_i$  are diagonal. Due to the monotonicity assumption on  $\sigma_i$ ,

$$\overline{\mathbf{A}}_i \geq \underline{\mathbf{A}}_i \geq \mathbf{0}, \quad (19)$$

where the inequalities are interpreted point-wise.

Define  $\mathbf{G}_i := \frac{\partial \mathbf{z}_i}{\partial \mathbf{x}}$  and  $\hat{\mathbf{G}}_i := \mathbf{W}_i \mathbf{G}_{i-1}$ . Denote the lower and upper bounds of  $\mathbf{G}_i$  as  $\underline{\mathbf{G}}_i, \overline{\mathbf{G}}_i \in \mathbb{R}^{k_i \times k_0}$ , and the lower and upper bounds of  $\hat{\mathbf{G}}_i$  as  $\underline{\hat{\mathbf{G}}}_i, \overline{\hat{\mathbf{G}}}_i \in \mathbb{R}^{k_i \times k_0}$ . The bounds on the gradients are initialized as  $\underline{\mathbf{G}}_0 = \overline{\mathbf{G}}_0 = \mathbf{I}$ , and can be updated inductively using interval arithmetic by forward propagation,<sup>9</sup>

$$\mathbf{G}_i = \nabla \sigma_i \hat{\mathbf{G}}_i = \nabla \sigma_i \mathbf{W}_i \mathbf{G}_{i-1}. \quad (20)$$

Given  $\underline{\mathbf{G}}_{i-1}$  and  $\overline{\mathbf{G}}_{i-1}$ ,

$$[\underline{\hat{\mathbf{G}}}_i, \overline{\hat{\mathbf{G}}}_i] = \mathbf{W}_i \otimes [\underline{\mathbf{G}}_{i-1}, \overline{\mathbf{G}}_{i-1}]. \quad (21)$$

<sup>9</sup> We can also use backward propagation in the chain rule to compute  $\nabla \mathbf{f}$ . Then the update equation becomes  $\frac{\partial \mathbf{y}}{\partial \mathbf{z}_i} = \frac{\partial \mathbf{y}}{\partial \mathbf{z}_{i+1}} \nabla \sigma_{i+1} \mathbf{W}_{i+1}$ .

The lower bound  $\underline{\mathbf{G}}_i$  on the gradient  $\mathbf{G}_i$  is the minimum of  $\nabla \sigma_i \hat{\mathbf{G}}_i$ , where  $\nabla \sigma_i \in [\underline{\Lambda}_i, \bar{\Lambda}_i]$  and  $\hat{\mathbf{G}}_i \in [\hat{\underline{\mathbf{G}}}_i, \hat{\bar{\mathbf{G}}}_i]$ . Since  $\nabla \sigma_i \geq \mathbf{0}$  according to (19), the minimum of  $\nabla \sigma_i \hat{\mathbf{G}}_i$  is achieved on the lower bound of  $\hat{\mathbf{G}}_i$ . Hence,

$$\underline{\mathbf{G}}_i = \min\{\underline{\Lambda}_i \hat{\underline{\mathbf{G}}}_i, \bar{\Lambda}_i \hat{\underline{\mathbf{G}}}_i\} = \underline{\Lambda}_i [\hat{\underline{\mathbf{G}}}_i]_+ + \bar{\Lambda}_i [\hat{\underline{\mathbf{G}}}_i]_-.$$
 (22)

Similarly, the upper bound on the gradient  $\frac{\partial z_i}{\partial \mathbf{x}}$  satisfies

$$\bar{\mathbf{G}}_i = \underline{\Lambda}_i [\hat{\bar{\mathbf{G}}}_i]_- + \bar{\Lambda}_i [\hat{\bar{\mathbf{G}}}_i]_+.$$
 (23)

The function to compute the bounds on the gradient given a non-trivial input set is implemented in algorithm 4.3. There is a solver, called RecurJac [50], that can efficiently compute the maximum gradient in a recursive manner.<sup>10</sup>

<sup>10</sup> <https://github.com/huanzhang12/RecurJac-Jacobian-Bounds>

Algorithm 4.3. Functions to compute gradient. The first function computes point-wise gradient, where `act_gradient` (not shown) computes the gradient of the activation function. The second and third functions compute the bounds on the gradient given a non-trivial input set. The input to the second function is the input set. It calls `act_gradient_bounds` (not shown) to compute  $\underline{\Lambda}_i$  and  $\bar{\Lambda}_i$ . The third function directly takes the bounds on the gradient of activation functions.

```
function get_gradient(nnet::Network, x::Vector)
    z = x
    gradient = Matrix{1.0I, length(x), length(x)}
    for (i, layer) in enumerate(nnet.layers)
        z_hat = affine_map(layer, z)
        o_gradient = act_gradient(layer.activation, z_hat)
        gradient = Diagonal(o_gradient) * layer.weights * gradient
        z = layer.activation(z_hat)
    end
    return gradient
end

function get_gradient(nnet::Network, input::AbstractPolytope)
    LA, UA = act_gradient_bounds(nnet, input)
    return get_gradient(nnet, LA, UA)
end

function get_gradient(nnet::Network, LA::Vector{Matrix}, UA::Vector{Matrix})
    n_input = size(nnet.layers[1].weights, 2)
    LG = Matrix{1.0I, n_input, n_input}
    UG = Matrix{1.0I, n_input, n_input}
    for (i, layer) in enumerate(nnet.layers)
        LG_hat, UG_hat = interval_map(layer.weights, LG, UG)
        LG = LA[i] * max.(LG_hat, 0) + UA[i] * min.(LG_hat, 0)
        UG = LA[i] * min.(UG_hat, 0) + UA[i] * max.(UG_hat, 0)
    end
    return (LG, UG)
end
```

#### 4.4 ReLU Activation

If  $\sigma_i$  is a ReLU activation function, i.e.,  $\sigma_i(\hat{\mathbf{z}}_i) = [\mathbf{z}_i]_+$ , we associate a binary vector  $\delta_i \in \{0, 1\}^{k_i}$  for  $i \in \{1, \dots, n\}$  to specify activation status (off or on) of the nodes. At layer  $i$ , given the bounds on the values of the nodes, we define the set of nodes that are activated  $\Gamma_i^+$ , not activated  $\Gamma_i^-$ , and undetermined  $\Gamma_i$  as

$$\Gamma_i^+ = \{j : \hat{\ell}_{i,j} \geq 0\},$$
 (24a)

$$\Gamma_i^- = \{j : \hat{u}_{i,j} \leq 0\},$$
 (24b)

$$\Gamma_i = \{j : \hat{\ell}_{i,j} < 0 < \hat{u}_{i,j}\}.$$
 (24c)

The diagonal entries  $\underline{\lambda}_{i,j}$  and  $\bar{\lambda}_{i,j}$  of the bounds  $\underline{\Lambda}_i, \bar{\Lambda}_i \in \mathbb{R}^{k_i \times k_i}$  on the gradient  $\nabla \sigma_i$  satisfy that

$$\bar{\lambda}_{i,j} = \begin{cases} 1 & j \in \Gamma_i^+ \\ 0 & j \in \Gamma_i^- \\ 1 & j \in \Gamma_i \end{cases}, \quad \underline{\lambda}_{i,j} = \begin{cases} 1 & j \in \Gamma_i^+ \\ 0 & j \in \Gamma_i^- \\ 0 & j \in \Gamma_i \end{cases}. \quad (25)$$

## 5 Reachability

Reachability methods compute the output reachable set  $\mathcal{R}(\mathcal{X}, \mathbf{f})$  to verify the problem through layer-by-layer analysis. This section first reviews the general methodology, and then discusses the specific methods.

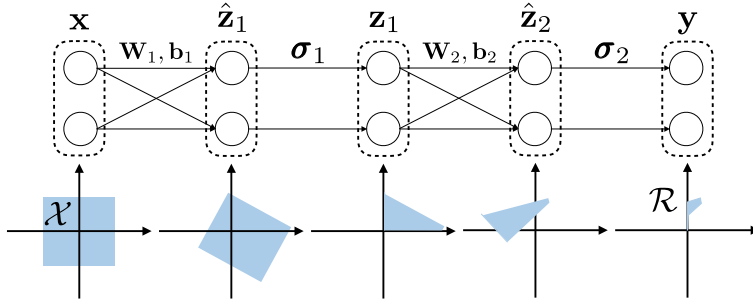


Figure 5. Illustration of reachability methods. The network in the illustration only contains one hidden layer. The input set  $\mathcal{X}$  is first passed through the linear mapping defined by  $\mathbf{W}_1$  and  $\mathbf{b}_1$ . Then it goes through the nonlinear mapping defined by  $\sigma_1$  (ReLU is considered). The corresponding reachable sets are illustrated in the shaded area. The process is repeated for the next layer. And the output reachable set is then obtained.

### 5.1 Overview

The layer-by-layer propagation in reachability methods is illustrated in figure 5 and implemented in algorithm 5.1. The function `forward_network` computes  $\mathcal{R}$ . It calls `forward_layer` to perform layer-by-layer forward propagation. Once it computes the reachable set, `check_inclusion` verifies whether  $\mathcal{R}(\mathcal{X}, \mathbf{f}) \subset \mathcal{Y}$ .

```

function solve(solver, problem::Problem)
    reach = forward_network(solver, problem.network, problem.input)
    return check_inclusion(reach, problem.output)
end

function forward_network(solver, nnet::Network, input::AbstractPolytope)
    reach = input
    for layer in nnet.layers
        reach = forward_layer(solver, layer, reach)
    end
    return reach
end

function check_inclusion(reach::Vector{<:AbstractPolytope}, output)
    for poly in reach
        issubset(poly, output) || return ReachabilityResult(:violated, reach)
    end
    return ReachabilityResult(:holds, similar(reach, 0))
end

```

Algorithm 5.1. General structure of reachability methods. The problem is solved by first performing layer-by-layer reachability analysis as specified in `forward_network`, then checking if the output reachable set belongs to the output constraint using `check_inclusion`. Different methods have different implementations of `forward_layer`. The outer loop `solve` can also vary between methods.



The function `forward_layer` involves the mapping  $\mathbf{z}_{i-1} \mapsto \sigma_i(\mathbf{W}_i \mathbf{z}_{i-1} + \mathbf{b}_i)$ . The linear mapping defined by  $\mathbf{z}_{i-1} \mapsto \mathbf{W}_i \mathbf{z}_{i-1} + \mathbf{b}_i$  is relatively easy to handle. The nonlinear mapping  $\hat{\mathbf{z}}_i \mapsto \sigma_i(\hat{\mathbf{z}}_i)$  is non-trivial. Different methods introduce different ways to handle nonlinear mappings. Hence, the implementation of `forward_layer` varies across different methods. There are at least five different approaches in the literature:

- *Exact reachability* (for piecewise linear networks), which computes the reachable set for every linear segment of the network and keeps track of all sets. This is done by ExactReach [46].
- *Split-and-join* (for piecewise linear networks), which computes the reachable set for every linear segment of the network and joins those sets by over-approximation. This is done by Ai2 [14].
- *Interval arithmetic* (for networks with monotone activation functions), which computes the bounds for each node separately by interval arithmetic. This is done by MaxSens [44].
- *Symbolic propagation* (for piecewise linear networks), which also computes the node-wise bounds using interval arithmetic but keeps track of dependencies among nodes using symbolic representations. This is done by ReluVal [40].
- *Network relaxation* (for piecewise linear networks), which computes symbolic lower and upper bounds based on a linear approximation of the network. This is done by FastLin [41].

The first method is exact without approximation. The next three methods only approximate the geometric objects passed through different layers. The last method approximates the network. The bounds can be computed using dynamic programming.

Figure 6 illustrates the difference between exact reachability and split-and-join in the case that the activation function is ReLU. Two nodes are considered, *i.e.*,  $\mathbf{z}_i \in \mathbb{R}^2$ . Hence, there are four piecewise linear components in the nonlinear mapping  $\hat{\mathbf{z}}_i \mapsto \sigma_i(\hat{\mathbf{z}}_i)$ , which correspond to the four quadrants shown in the left plots of figure 6. Under ReLU, the set in the first quadrant (*i.e.*, both values are greater than zero) is kept the same after the mapping. The sets in the second and the fourth quadrants (*i.e.*, only one value is greater than zero) are mapped to line segments. The set in the third quadrant (*i.e.*, both values are smaller than zero) is mapped to the origin. Exact reachability keeps track of all geometric objects after the mapping. In figure 6a, one input set generates four geometric objects to represent the reachable set. The number of geometric objects grows exponentially with the number of nodes at each layer. On the other hand, to improve scalability of the method, split-and-join methods merge all geometric objects together as shown in figure 6b. Mathematical derivations will be introduced in section 5.2 for exact reachability and in section 5.3 for split-and-join.

Exact reachability and split-and-join methods do not distinguish individual nodes, but consider all nodes at a layer as a whole. Though these methods make it easy to track dependencies between nodes, the resulting high dimensional geometric objects (usually polytopes) can be inefficient to manipulate. Interval arithmetic considers node-wise reachability by computing reachable intervals for all nodes separately as shown in figure 7a. Consequently, the reachable set is a hyperrectangle, which is easy to manipulate. However, as the dependencies among nodes are removed, interval arithmetic may result in significant over-approximation. Symbolic propagation then introduces symbolic intervals to track those dependencies as shown in figure 7b. As illustrated in figure 7, symbolic propagation can generate much tighter bounds.<sup>11</sup>

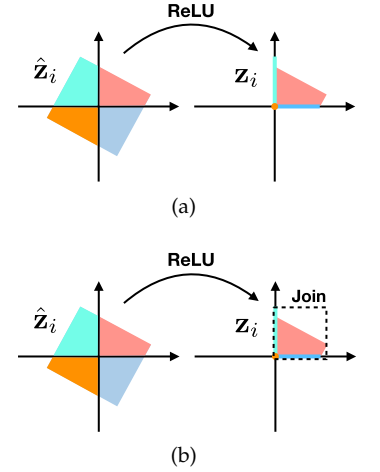


Figure 6. Illustration of different approaches to handle the nonlinear mapping  $\hat{\mathbf{z}}_i \mapsto \sigma_i(\hat{\mathbf{z}}_i)$ . (a) Exact reachability. (b) Split-and-join.

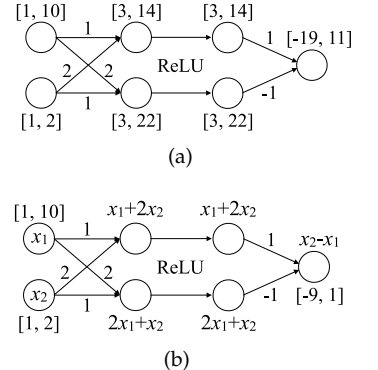


Figure 7. Illustration of the difference between interval arithmetic and symbolic propagation. (a) Interval arithmetic. (b) Symbolic propagation.

<sup>11</sup> To tighten the bounds in interval arithmetic, in addition to do symbolic propagation, we can also divide the input space into smaller intervals, and compute the reachable output intervals for those small intervals. This approach is used in MaxSens and ReluVal.



Rigorous mathematical derivations will be introduced in section 5.4 for interval arithmetic and section 8.1 for symbolic propagation.

For different methods, their outer loops, *e.g.*, the `solve` function, can also be different. The simplest outer loop is shown in algorithm 5.1, which works for ExactReach and Ai2. In some methods, the inputs are split into smaller segments to minimize over-approximation as illustrated in figure 8.<sup>12</sup> The reachable set for each segment is computed and then joined together. MaxSens uses this approach. However, blindly splitting the input set may not be efficient. By combining with search, we can look for the most influential inputs to split. ReluVal uses this approach. We may also use binary search to adjust the radius of the segment, which is adopted in FastLin.

This section discusses ExactReach, Ai2, and MaxSens. ReluVal and FastLin will be discussed in section 8.

## 5.2 ExactReach

ExactReach [46] performs exact reachability analysis for networks with linear or ReLU activations. For any ReLU function, if the input set is a union of polytopes, then the output reachable set is also a union of polytopes as shown in figure 6a. In the implementation, the input set and output set are both set to `HPolytope`. The function to compute the reachable set for a single layer is shown in algorithm 5.2, which is called in the main loop of algorithm 5.1.

The input set to layer  $i$  consists of a list of H-polytopes. One input H-polytope parameterized by  $\mathbf{C} \in \mathbb{R}^{k \times k_{i-1}}$  and  $\mathbf{d} \in \mathbb{R}^k$  defines a set

$$\mathcal{I} = \{\mathbf{z}_{i-1} : \mathbf{C}\mathbf{z}_{i-1} \leq \mathbf{d}\}, \quad (26)$$

where  $k$  is the number of constraints. After the linear mapping  $\mathbf{z}_{i-1} \mapsto \mathbf{W}_i\mathbf{z}_{i-1} + \mathbf{b}_i$ , the set before activation is denoted

$$\hat{\mathcal{I}} = \{\hat{\mathbf{z}}_i : \hat{\mathbf{C}}\hat{\mathbf{z}}_i \leq \hat{\mathbf{d}}\}, \quad (27)$$

where  $\hat{\mathbf{C}} \in \mathbb{R}^{k \times k_i}$  and  $\hat{\mathbf{d}} \in \mathbb{R}^k$ . The number of inequality constraints  $k$  in (27) may be different from that in (26). The set  $\hat{\mathcal{I}}$  can be computed by calling `linear_transform`, which converts  $\mathcal{I}$  into a V-polytope, applies the linear map to all vertices, then converts it back to an H-polytope.

The set  $\hat{\mathcal{I}}$  can be partitioned into several non-intersecting subsets according to different activation patterns. The activation status for  $\mathbf{z}_i$  is denoted  $\delta_i \in \{0, 1\}^{k_i}$ . Since the entries in  $\delta_i$  are binary, there is a bijection between  $\delta_i$  and an integer  $h \in \{0, 1, \dots, 2^{k_i} - 1\}$ .<sup>13</sup> Define a diagonal matrix  $\mathbf{P}_h \in \mathbb{R}^{k_i \times k_i}$ , whose diagonal entries are the entries in the binary vector  $\delta_i$ . Hence, there is a correspondence between  $\mathbf{P}_h$  and the integer  $h$ . For a given activation  $\delta_i$ , the before activation node  $\hat{\mathbf{z}}_i$  needs to satisfy that if  $\delta_{i,j} = 1$ , then  $\hat{z}_{i,j} \geq 0$ ; and if  $\delta_{i,j} = 0$ , then  $\hat{z}_{i,j} \leq 0$ . The subset of  $\hat{\mathcal{I}}$  that corresponds to the  $h$ th activation pattern is

$$\hat{\mathcal{I}}_h = \{\hat{\mathbf{z}}_i : \mathbf{P}_h\hat{\mathbf{z}}_i \geq \mathbf{0}, (\mathbf{I} - \mathbf{P}_h)\hat{\mathbf{z}}_i \leq \mathbf{0}, \hat{\mathbf{C}}\hat{\mathbf{z}}_i \leq \hat{\mathbf{d}}\}. \quad (28)$$

In a compact form, the constraint on  $\hat{\mathbf{z}}_i$  is

$$\begin{bmatrix} \hat{\mathbf{C}} \\ \mathbf{I} - 2\mathbf{P}_h \end{bmatrix} \hat{\mathbf{z}}_i \leq \begin{bmatrix} \hat{\mathbf{d}} \\ \mathbf{0} \end{bmatrix}. \quad (29)$$

The expression  $\mathbf{I} - 2\mathbf{P}_h$  is a diagonal matrix whose entries are either 1 (for inactive nodes) or  $-1$  (for active nodes). The constraint  $(\mathbf{I} - 2\mathbf{P}_h)\hat{\mathbf{z}}_i \leq \mathbf{0}$  is a combination of the two constraints  $\mathbf{P}_h\hat{\mathbf{z}}_i \geq \mathbf{0}$  and  $(\mathbf{I} - \mathbf{P}_h)\hat{\mathbf{z}}_i \leq \mathbf{0}$ .

<sup>12</sup> Recall `split_interval`, first introduced in section 4.

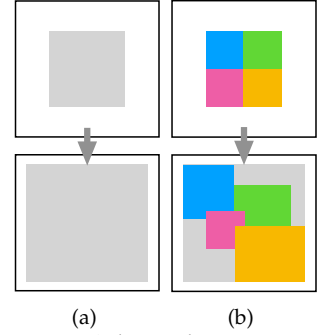


Figure 8. Splitting the input set to minimize over-approximation. The upper square is the input domain and the lower square is the output domain. (a) Over-approximated reachable set (gray square in the output domain) without partition. (b) Over-approximated reachable sets for different segments of the input set. Colors show correspondence.

<sup>13</sup> When  $h = 0$ , all nodes are inactive. When  $h = 2^{k_i} - 1$ , all nodes are active.

For  $\hat{\mathbf{z}}_i \in \hat{\mathcal{I}}_h$ , the after activation nodes satisfy that  $\mathbf{z}_i = \mathbf{P}_h \hat{\mathbf{z}}_i$ . Hence, the reachable set  $\mathcal{O}_h$  for  $\hat{\mathcal{I}}_h$  is a linear transform  $\hat{\mathbf{z}}_i \rightarrow \mathbf{P}_h \hat{\mathbf{z}}_i$  of  $\hat{\mathcal{I}}_h$  defined by (29). We write the linear transformation as

$$\mathcal{O}_h = \mathbf{P}_h \circ \hat{\mathcal{I}}_h. \quad (30)$$

The above process is implemented in `forward_partition`.

Finally, the output reachable set for  $\mathcal{I}$  is the union of all  $\mathcal{O}_h$ ,

$$\mathcal{O} = \bigcup_{h=0}^{2^{k_i}-1} \mathcal{O}_h. \quad (31)$$

It has been shown that the output set  $\mathcal{O}$  is *tight* [46], meaning that it is not an over-approximation in the sense that for any point  $\mathbf{z}_i$  in  $\mathcal{O}$ , there is a point  $\mathbf{z}_{i-1}$  in  $\mathcal{I}$  satisfying  $\mathbf{z}_i = \mathbf{f}_i(\mathbf{z}_{i-1})$ .

For one input polytope, the output for one layer generates  $2^{k_i}$  polytopes. Hence, the number of polytopes grows exponentially with the depth. Though the empty sets can be pruned out in the process, it is still inefficient to keep track of the exact reachable set for large neural networks. Xiang, Tran, and Johnson developed a toolbox, called NNV,<sup>14</sup> for efficient reachability analysis using parallel computation.

<sup>14</sup> <https://github.com/verivital/nnv>.

```

struct ExactReach end

function forward_layer(solver::ExactReach, layer::Layer, input::HPolytope)
    input = affine_map(layer, input)
    return forward_partition(layer.activation, input)
end

function forward_partition(act::ReLU, input::HPolytope)
    n = dim(input)
    output = Vector{HPolytope}(undef, 0)
    C, d = tosimplehrep(input)
    dh = [d; zeros(n)]
    for h in 0:(2^n)-1
        P = getP(h, n)
        Ch = [C; I - 2P]
        input_h = HPolytope(Ch, dh)
        if !isempty(input_h)
            push!(output, linear_map(Matrix{Float64}(P), input_h))
        end
    end
    return output
end

```

Algorithm 5.2. ExactReach. The main process follows from algorithm 5.1. In the `forward_layer` function, each input set  $\mathcal{I}$  first goes through an affine map to  $\hat{\mathcal{I}}$ . Then the function `forward_partition` partitions  $\hat{\mathcal{I}}$  into  $\hat{\mathcal{I}}_h$  for the  $h$ th activation pattern. The output set  $\mathcal{O}_h$  for each  $\hat{\mathcal{I}}_h$  is computed using linear transformation and the reachable set is a union of all these  $\mathcal{O}_h$ .

### 5.3 Ai2

In many cases, the exact reachable set is intractable. In Ai2 [14], an estimate  $\tilde{\mathcal{R}}(\mathcal{X}, \mathbf{f})$  of the reachable set is obtained such that  $\mathcal{R}(\mathcal{X}, \mathbf{f}) \subseteq \tilde{\mathcal{R}}(\mathcal{X}, \mathbf{f})$ .

Ai2 uses an *abstract domain* to approximate the reachable set at each layer, which is represented by a set of logical formulas that capture certain geometric shapes, such as the geometries and their formulas introduced in section 2.2.<sup>15</sup> The choice of abstract domain needs to balance between precision and scalability. For example, the polytopes used in ExactReach are precise but not scalable, while the hyperrectangles used in MaxSens are scalable but too loose. The original implementation of Ai2 uses zonotopes, center-symmetric convex closed polytopes, which are more scalable than polytopes and tighter than hyperrectangles. Due to lack of tools to manipulate geometries defined by zonotopes, we use polytopes in our implementation.

Ai2 works for piecewise linear activation functions, *e.g.*, ReLU and max pooling. Any piecewise linear activation function can be described as one conditional affine transformation (CAT), which consists of a set of linear conditions and a set of affine mappings corresponding to the linear conditions. For example, for a ReLU activation,

$$\sigma_i(\hat{\mathbf{z}}_i) = \begin{cases} \mathbf{P}_0 \hat{\mathbf{z}}_i & \text{if } (\mathbf{I} - 2\mathbf{P}_0) \hat{\mathbf{z}}_i \leq \mathbf{0} \\ \mathbf{P}_1 \hat{\mathbf{z}}_i & \text{if } (\mathbf{I} - 2\mathbf{P}_1) \hat{\mathbf{z}}_i \leq \mathbf{0} \\ \vdots & \\ \mathbf{P}_{2^{k_i}-1} \hat{\mathbf{z}}_i & \text{if } (\mathbf{I} - 2\mathbf{P}_{2^{k_i}-1}) \hat{\mathbf{z}}_i \leq \mathbf{0} \end{cases}, \quad (32)$$

where  $\mathbf{P}_h$  for different  $h$  is defined in section 5.2. Each condition corresponds to one specific activation pattern.

To propagate an abstract domain through a CAT, Ai2 introduces two basic operations: *meet* ( $\sqcap$ ) and *join* ( $\sqcup$ ). The meet operation splits an abstract domain into different subdomains that correspond to different conditions of the CAT. Due to the restriction of the abstract domain, those subdomains may be over-approximated and overlap with each other. For example, if the abstract domain is chosen to be hyperrectangles, then a subdomain corresponding to the condition  $(\mathbf{I} - 2\mathbf{P}_h) \hat{\mathbf{z}}_i \leq \mathbf{0}$  is the smallest hyperrectangle that includes all points that satisfy the condition. After the meet operation, the reachable sets of those subdomains are computed with respect to the linear maps under the corresponding conditions. The join operation then uses one instance of the abstract domain to cover and approximate all the reachable sets. The implementation of the meet and join operations is deeply related to the chosen abstract domain. If the abstract domain is the polytope domain, ExactReach also splits the input set according to different cases and computes  $\mathcal{O}_h$  in different cases. However, ExactReach does not have a join operation. Instead, it keeps track of all sets in (31).

Our implementation is shown in algorithm 5.3. For simplicity, we only consider ReLU activations. The input and output sets are both polytopes.

An input set  $\mathcal{I}$  at layer  $i$  goes through the linear map defined by  $\mathbf{W}_i$  and  $\mathbf{b}_i$ . The matrix  $\mathbf{W}_i$  rotates the set  $\mathcal{I}$  and  $\mathbf{b}_i$  shifts the center of the set. The set after those linear maps is denoted  $\hat{\mathcal{I}}$ . Then the output set is

$$\hat{\mathcal{I}}_h = \hat{\mathcal{I}} \sqcap \{\hat{\mathbf{z}}_i : (\mathbf{I} - 2\mathbf{P}_h) \hat{\mathbf{z}}_i \leq \mathbf{0}\}, \quad (33a)$$

$$\mathcal{O}_h = \mathbf{P}_h \circ \hat{\mathcal{I}}_h, \quad (33b)$$

$$\mathcal{O} = \sqcup_{h=0}^{2^{k_i}-1} \mathcal{O}_h. \quad (33c)$$

where (33a) is the meet step, (33b) the linear mapping, and (33c) the join step. (33a) is the same as in (30). The meet function adds more constraints to the set. Since the mapping in (33b) is linear, we can efficiently compute the output set by simply mapping all vertices. Then all those sets are joined using a convex hull.<sup>16</sup>

<sup>15</sup> A detailed discussion of an abstract domain for certifying neural networks can be found in [34]. An extended version of Ai2, called ERAN, which is robust to floating point arithmetic [34], can be found in <https://github.com/eth-sri/eran>. An approach that combines Ai2 with mixed integer optimization is discussed in Singh et al. [35].

<sup>16</sup> The original implementation of Ai2 uses a zonotope to represent the abstract domain. The meet and join operations are implemented differently and are customized for zonotopes.

For one input polytope, the output reachable set is still one polytope. Hence, Ai2 is much more scalable than ExactReach, though the over-approximation results in incompleteness. Following Ai2, the authors introduced DeepZ [33], which can scale to other activation functions, and is robust to floating point operations. Recently, Yang et al. introduce a method combining abstract domain with symbolic propagation to further improve scalability and precision of the verification algorithm [49].

```

struct Ai2 end

function forward_layer(solver::Ai2, layer::Layer, input::AbstractPolytope)
    outlinear = affine_map(layer, input)
    relued_subsets = forward_partition(layer.activation, outlinear)
    return convex_hull(relued_subsets)
end

```

#### 5.4 MaxSens

MaxSens [44] is also a reachability method that uses over-approximation. It works for networks with monotone activation functions and low-dimensional input and output spaces. The key idea of MaxSens is to grid the input space and compute the reachable set for each grid cell. The finer the grid cells, the smaller the over-approximation. Computing the reachable sets for different cells can be done in parallel.

Algorithm 5.4 provides an implementation of MaxSens. The input set is a hyperrectangle. The output set can be any abstract polytope. There is one more step in the main loop than in the general `solve` function in algorithm 5.1, which is to partition the input set into several grid cells. The `forward_layer` function takes a hyperrectangle input set and outputs the reachable hyperrectangle.

Suppose the input set at layer  $i$  is

$$\mathcal{I} = \{\mathbf{z}_{i-1} : |\mathbf{z}_{i-1} - \mathbf{c}_{i-1}| \leq \mathbf{r}_{i-1}\}, \quad (34)$$

where  $\mathbf{c}_{i-1} \in \mathbb{R}^{k_{i-1}}$  is the center of the hyperrectangle and  $\mathbf{r}_{i-1} \in \mathbb{R}^{k_{i-1}}$  is the radius of the hyperrectangle. The output reachable set is over-approximated by a hyperrectangle

$$\mathcal{O} = \{\mathbf{z}_i : |\mathbf{z}_i - \mathbf{c}_i| \leq \mathbf{r}_i\}, \quad (35)$$

where  $\mathbf{c}_i, \mathbf{r}_i \in \mathbb{R}^{k_i}$ .

For simplicity, define the following node-wise values for layer  $i$ ,

$$\beta_j = \sigma_{i,j}(\mathbf{w}_{i,j}\mathbf{c}_{i-1} + b_{i,j}), \quad (36a)$$

$$\beta_j^{\max} = \sigma_{i,j}(\mathbf{w}_{i,j}\mathbf{c}_{i-1} + |\mathbf{w}_{i,j}|\mathbf{r}_{i-1} + b_{i,j}), \quad (36b)$$

$$\beta_j^{\min} = \sigma_{i,j}(\mathbf{w}_{i,j}\mathbf{c}_{i-1} - |\mathbf{w}_{i,j}|\mathbf{r}_{i-1} + b_{i,j}), \quad (36c)$$

where  $\sigma_{i,j}$  is the activation function for the  $j$ th node, and  $\mathbf{w}_{i,j}$  is the  $j$ th row of  $\mathbf{W}_i$ . Due to monotonicity of  $\sigma_{i,j}$ , we have

$$z_{i,j} = \sigma_{i,j}(\mathbf{w}_{i,j}\mathbf{z}_{i-1} + b_{i,j}) \in [\beta_j^{\min}, \beta_j^{\max}], \forall \mathbf{z}_{i-1} \in \mathcal{I}. \quad (37)$$

There are three different ways to define the over-approximated set  $\mathcal{O}$  as illustrated in figure 9.

Algorithm 5.3. Ai2. Every set goes through meet and join as defined in equation (33). Note that in the case of H-polytope input sets, meet and join are equivalently replaced with `forward_partition` (defined in algorithm 5.2) and `convex_hull`, respectively. The meet operation computes the part of input that satisfies a given activation pattern by adding more linear constraints to the input `HPolytope` according to equation (33a).

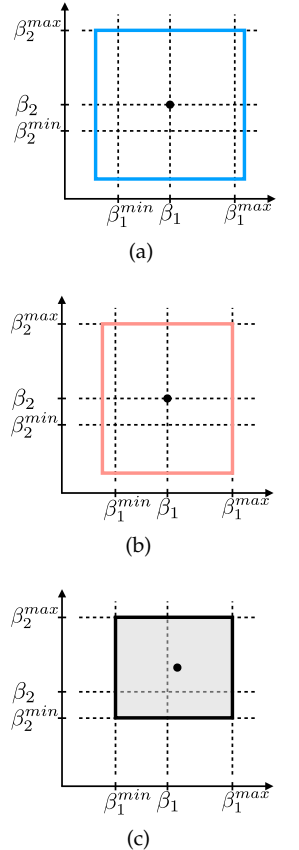


Figure 9. Illustration of different approximations of output reachable sets. (a) Center-aligned set with uniform radius. (b) Center-aligned set with non uniform radius. (c) Tight set.

- Center-aligned set with uniform radius.

As illustrated in figure 9a, the output set  $\mathcal{O}$  is constructed to be a hypercube by setting all entries of  $\mathbf{r}_i$  to be equal to its maximum element. Though the reachable set becomes looser, this method requires less memory to store intermediate results because we only need to store a scalar bound instead of a vector of bounds. The uniform bound indeed represents the “maximum sensitivity” of the network given the input set. However, since we use `Hyperrectangle` objects to store intermediate results anyway, a uniform bound does not significantly enhance efficiency. Hence, our implementation uses a non-uniform bound for both  $\mathcal{I}$  and  $\mathcal{O}$ .

- Center-aligned set with non-uniform radius.

As illustrated in figure 9b, the output set  $\mathcal{O}$  is no longer required to have uniform radius. By requiring that the center of the input hyperrectangle  $\mathcal{I}$  maps to the center of the output hyperrectangle  $\mathcal{O}$ , the center and radius of the  $\mathcal{O}$  can be obtained, for all  $j \in \{1, \dots, k_i\}$ ,

$$c_{i,j} = \beta_j, \quad (38a)$$

$$r_{i,j} = \max_{\mathbf{z}_{i-1} \in \mathcal{I}} |\sigma_{i,j}(\mathbf{w}_{i,j}\mathbf{z}_{i-1} + b_{i,j}) - c_{i,j}| \quad (38b)$$

$$= \max\{\beta_j^{max} - \beta_j, \beta_j - \beta_j^{min}\}. \quad (38c)$$

where  $r_{i,j}$  and  $c_{i,j}$  are the  $j$ th entry in  $\mathbf{r}_i$  and  $\mathbf{c}_i$ . The last equality is due to the monotonicity of  $\sigma_{i,j}$ .

- Tight set.

A even tighter result can be obtained by directly using  $\beta_j^{min}$  and  $\beta_j^{max}$  as bounds and not aligning the centers of  $\mathcal{I}$  and  $\mathcal{O}$ . As illustrated in figure 9c, the center and radius of the tight set can be defined as

$$c_{i,j} = \frac{\beta_j^{min} + \beta_j^{max}}{2}, \quad (39a)$$

$$r_{i,j} = \frac{\beta_j^{max} - \beta_j^{min}}{2}. \quad (39b)$$

Equivalently, the output set can be represented as

$$\mathcal{O}^* = \{\mathbf{z}_i : \beta_j^{min} \leq z_{i,j} \leq \beta_j^{max}, \forall j\}. \quad (40)$$

It is easy to show that the resulting node-wise bounds are the same as the bounds computed in (15) by interval arithmetic.

The last two cases are implemented in `forward_node`. In algorithm 5.4, the solver has a boolean field `tight`. When `tight` is set to be true, it computes the set in (39). Otherwise, it computes the center-aligned set in (38). The center-aligned set is desired if we want to estimate the maximum sensitivity (or maximum gradient) of the network  $\mathbf{f}$  at the center of the input set.

The advantage of MaxSens over other reachability methods is that the number of geometric objects does not grow during the layer-by-layer propagation. The total number of hyperrectangles only depends on the the initial partition. Though the number of hyperrectangles will not grow during the layer-by-layer propagation, the error of over-approximation will accumulate quickly with respect to the number of layers. For networks with many input nodes, the number of hyperrectangles on the initial partition can be prohibitively large for a tight estimation. Otherwise, the computation with a sparse partition will be overly conservative [47]. To improve the initial partition, the authors developed a specification-guided method [47] that can adaptively choose the partition resolution according to the problem specification.

Some of the methods discussed later need to compute the bounds of each node. We implement the `get_bounds` function in algorithm 5.5 based on MaxSens, but with the tighter output reachable set (39), which is equivalent to the interval arithmetic introduced in (15) in section 4.

```

struct MaxSens
    resolution::Float64 = 1.0
    tight::Bool         = false
end

function solve(solver::MaxSens, problem::Problem)
    inputs = partition(problem.input, solver.resolution)
    f_n(x) = forward_network(solver, problem.network, x)
    outputs = map(f_n, inputs)
    return check_inclusion(outputs, problem.output)
end

function forward_layer(solver::MaxSens, L::Layer, input::Hyperrectangle)
    (W, b, act) = (L.weights, L.bias, L.activation)
    center = zeros(size(W, 1))
    gamma = zeros(size(W, 1))
    for j in 1:size(W, 1)
        node = Node(W[j,:], b[j], act)
        center[j], gamma[j] = forward_node(solver, node, input)
    end
    return Hyperrectangle(center, gamma)
end

function forward_node(solver::MaxSens, node::Node, input::Hyperrectangle)
    output = node.w' * input.center + node.b
    deviation = sum(abs.(node.w) .* input.radius)
    β = node.act(output)
    βmax = node.act(output + deviation)
    βmin = node.act(output - deviation)
    if solver.tight
        return ((βmax + βmin)/2, (βmax - βmin)/2)
    else
        return (β, max(abs(βmax - β), abs(βmin - β)))
    end
end

```

Algorithm 5.4. MaxSens. The main `solve` function is slightly different from the general reachability method in algorithm 5.1 by adding a partition step. In `forward_layer`, the bounds are computed for each node using a method similar to interval arithmetic.

```

function get_bounds(problem::Problem)
    solver = MaxSens(1.0, true)
    bounds = Vector{Hyperrectangle}(length(nnet.layers) + 1)
    bounds[1] = input
    for (i, layer) in enumerate(nnet.layers)
        bounds[i+1] = forward_layer(solver, layer, bounds[i])
    end
    return bounds
end

```

Algorithm 5.5. Function to compute node-wise bounds. It outputs the tight bounds in MaxSens.

## 6 Primal Optimization

There can be many different designs of the optimization problem to verify (4). A common structure is

$$\min_{\mathbf{x}, \mathbf{y}} o(\mathbf{x}, \mathbf{y}, \mathcal{X}, \mathcal{Y}), \quad (41a)$$

$$\text{s.t. } \mathbf{x} \in \mathcal{X}, \mathbf{y} \notin \mathcal{Y}, \mathbf{y} = \mathbf{f}(\mathbf{x}), \quad (41b)$$

where  $o(\mathbf{x}, \mathbf{y}, \mathcal{X}, \mathcal{Y})$  is an objective function, which may depend on the input  $\mathbf{x}$ , the output  $\mathbf{y}$  and their domains  $\mathcal{X}$  and  $\mathcal{Y}$ . We can either minimize or maximize the objective function. The major difficulty in solving (41) is the nonlinear and non-convex constraint imposed by the network  $\mathbf{f}$ .

NSVerify [23] and MIPVerify [38] reformulate the problem (41) into a mixed integer linear programming (MILP). Iterative LP (ILP) [4] approximates the problem (41) as a linear programming and solves it by iteratively adding the constraints.

This section first provides an overview of common methods to simplify the primal optimization problem, then discusses the three methods (*i.e.*, NSVerify, MIPVerify, ILP) in detail. For simplicity, we only consider ReLU activations. There are different ways to encode ReLU networks as linear constraints. The following two types of variables are usually used as decision variables in the optimization.<sup>17</sup>

- Variables for all nodes, *i.e.*,  $\mathbf{z}_i \in \mathbb{R}^{k_i}$  for  $i \in \{0, 1, \dots, n\}$ .
- Variables for all activations, *i.e.*,  $\delta_i \in \{0, 1\}^{k_i}$  for  $i \in \{1, \dots, n\}$ .

Our implementation uses JuMP.jl<sup>18</sup> to model and solve the optimizations. In the following discussion, `Model` refers to a jump model and `solve(:Model)` calls the JuMP solver to solve the optimization problem encoded in the model.

### 6.1 Encoding Network as Constraints

Primal optimization needs to deal with the constraint imposed by the network. The constraint  $\mathbf{y} = \mathbf{f}(\mathbf{x})$  is equivalent to

$$z_{i,j} = [\mathbf{w}_{i,j}\mathbf{z}_{i-1} + b_{i,j}]_+, \forall i \in \{1, \dots, n\}, \forall j \in \{1, \dots, k_i\}. \quad (42)$$

There are different ways to encode (42). For given activations  $\delta_i$ , equation (42) can be encoded as linear constraints as shown in algorithms 6.1 to 6.3. For given bounds  $\ell_i$  and  $\mathbf{u}_i$ , equation (42) can be encoded as linear constraints via triangle relaxation as shown in algorithm 6.4. Or (42) can be encoded as mixed integer linear constraints as shown in algorithms 6.5 and 6.6. In the following discussion, we use  $\hat{z}_{i,j}$  for  $\mathbf{w}_{i,j}\mathbf{z}_{i-1} + b_{i,j}$ , though  $\hat{z}_{i,j}$  is not a variable to be considered in the optimization.

<sup>17</sup> NSVerify, MIPVerify, and ILP use these two kinds of variables. Other methods may use different sets of variables. For example, Reluplex uses  $\mathbf{z}_i$  and  $\hat{\mathbf{z}}_i$  as decision variables.

<sup>18</sup> <https://github.com/jump.jl>

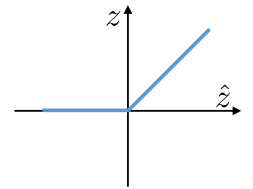


Figure 10. Illustration of ReLU activation function.

*Linear constraints for given  $\delta_i$*  For a given activation  $\delta_i$  for any  $i$ , the network can be encoded as a set of linear constraints for  $j \in \{1, \dots, k_i\}$ :

$$\delta_{i,j} = 1 \Rightarrow z_{i,j} = \hat{z}_{i,j} \geq 0, \quad (43a)$$

$$\delta_{i,j} = 0 \Rightarrow z_{i,j} = 0, \hat{z}_{i,j} \leq 0. \quad (43b)$$

The function is shown in algorithm 6.1. These constraints represent a subset of the original constraint (42), since the activation pattern is determined. This function is called in local search in Sherlock.

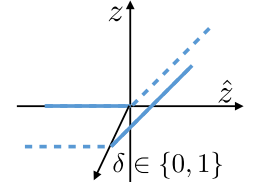


Figure 11. Illustration of linear encoding for given  $\delta$ . If  $\delta$  is unknown, it corresponds to mixed integer encoding. The  $\delta$  axis is perpendicular to the  $z - \hat{z}$  plane in 3D.

Algorithm 6.1. Encoding the network as linear constraints for given  $\delta_i$ . The function `encode_newtork!` encodes the constraints layer by layer by calling `encode_layer!`. The encoding method is specified in the abstract type `AbstractLinearProgram`.

```

abstract type AbstractLinearProgram end
struct StandardLP <: AbstractLinearProgram end

function encode_network!(model::Model, network, z,  $\delta$ ,
                        encoding::AbstractLinearProgram)
    for (i, layer) in enumerate(network.layers)
        encode_layer!(encoding, model, layer, z[i], z[i+1],  $\delta[i]$ )
    end
    return encoding
end

function encode_layer! (::StandardLP, model, layer, zi, zi+1,  $\delta_{i+1}$ )
     $\hat{z}$  = affine_map(layer, zi)
    for j in 1:length(layer.bias)
        if  $\delta_{i+1}[j]$ 
            @constraint(model,  $\hat{z}[j] \geq 0.0$ )
            @constraint(model, zi+1[j] ==  $\hat{z}[j]$ )
        else
            @constraint(model,  $\hat{z}[j] \leq 0.0$ )
            @constraint(model, zi+1[j] == 0.0)
        end
    end
end

```

*Relaxed linear constraints for given  $\delta_i$*  In some cases, we drop the inequalities in (43) to get a relaxed encoding to ensure that the optimization problem is still feasible for infeasible activation  $\delta_i$ 's. The relaxed encoding is

$$\delta_{i,j} = 1 \Rightarrow z_{i,j} = \hat{z}_{i,j}, \quad (44a)$$

$$\delta_{i,j} = 0 \Rightarrow z_{i,j} = 0. \quad (44b)$$

The function is shown in algorithm 6.2 and illustrated in figure 12. This function is used in ILP.

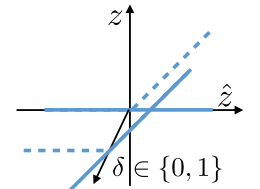


Figure 12. Illustration of relaxed linear encoding for given  $\delta$ .

*Slack linear constraints for given  $\delta_i$*  The previous relaxed linear encoding allows violation of the constraints imposed by ReLU's. We can use slack variables  $s_{i,j}$ 's to estimate the violations in the linear constraints (43):

$$\delta_{i,j} = 1 \Rightarrow \hat{z}_{i,j} + s_{i,j} \geq 0, \quad (45a)$$

$$\delta_{i,j} = 0 \Rightarrow \hat{z}_{i,j} - s_{i,j} \leq 0. \quad (45b)$$

The function is shown in algorithm 6.3. It is used in Planet in order to detect conflicts among different  $\delta$ 's.



```

struct LinearRelaxedLP <: AbstractLinearProgram end

function encode_layer!(::LinearRelaxedLP, model, layer, zi, zi+1, δi+1)
     $\hat{z}$  = affine_map(layer, zi)
    for j in 1:length(layer.bias)
        if δi+1[j]
            @constraint(model, zi+1[j] ==  $\hat{z}$ [j])
        else
            @constraint(model, zi+1[j] == 0.0)
        end
    end
end

```

Algorithm 6.2. Encoding the network as relaxed linear constraints for given  $\delta_i$ . Only the function `encode_layer!` is shown. The function `encode_network!` is the same as in algorithm 6.1.

```

struct SlackLP <: AbstractLinearProgram
    slack::Vector{Vector{VariableRef}}
end

function encode_layer!(SLP::SlackLP, model, layer, zi, zi+1, δi+1)
     $\hat{z}$  = affine_map(layer, zi)
    slack_vars = @variable(model, [1:length(layer.bias)])
    for j in 1:length(layer.bias)
        if δi+1[j]
            @constraint(model, zi+1[j] ==  $\hat{z}$ [j] + slack_vars[j])
            @constraint(model,  $\hat{z}$ [j] + slack_vars[j] >= 0.0)
        else
            @constraint(model, zi+1[j] == slack_vars[j])
            @constraint(model, 0.0 >=  $\hat{z}$ [j] - slack_vars[j])
        end
    end
    push!(SLP.slack, slack_vars)
end

```

Algorithm 6.3. Encoding the network as slack linear constraints for given  $\delta_i$ . Only the function `encode_layer!` is shown. The function `encode_network!` is the same as in algorithm 6.1.

*Triangle relaxation for given  $\ell_i$  and  $\mathbf{u}_i$*  When we can bound the value of the nodes, we can use  $\Delta$ -relaxation to encode the constraint as

$$j \in \Gamma_i^+ \Rightarrow z_{i,j} = \hat{z}_{i,j}, \hat{z}_{i,j} \geq 0, \quad (46a)$$

$$j \in \Gamma_i^- \Rightarrow z_{i,j} = 0, \hat{z}_{i,j} \leq 0, \quad (46b)$$

$$j \in \Gamma_i \Rightarrow z_{i,j} \geq \hat{z}_{i,j}, z_{i,j} \geq 0, z_{i,j} \leq \frac{\hat{u}_{i,j}(\hat{z}_{i,j} - \hat{\ell}_{i,j})}{\hat{u}_{i,j} - \hat{\ell}_{i,j}}, \quad (46c)$$

where activated nodes  $\Gamma_i^+$ , unactivated nodes  $\Gamma_i^-$ , and undetermined nodes  $\Gamma_i$  are introduced in (24). The function for  $\Delta$ -relaxation is shown in algorithm 6.4. The function `linear_transform` corresponds to (15).  $\Delta$ -relaxation is used in ConvDual and Planet.

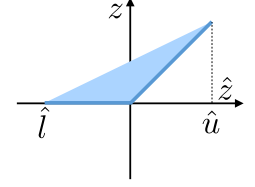


Figure 13. Illustration of triangle relaxation.

Algorithm 6.4. Encoding the network as linear constraints for given  $\ell_i$  and  $\mathbf{u}_i$  via  $\Delta$ -relaxation. Only the function `encode_layer!` is shown. The function `encode_network!` is the same as in algorithm 6.1.

```

struct TriangularRelaxedLP <: AbstractLinearProgram end

function encode_layer! (::TriangularRelaxedLP, model, layer, z_i, z_{i+1}, bounds)
     $\hat{z}$  = affine_map(layer, z_i)
     $\hat{z\_bound}$  = approximate_affine_map(layer, bounds)
     $\hat{\ell}$ ,  $\hat{u}$  = low( $\hat{z\_bound}$ ), high( $\hat{z\_bound}$ )
    for j in 1:length(layer.bias)
        if  $\hat{\ell}[j] > 0.0$ 
            @constraint(model, z_{i+1}[j] ==  $\hat{z}[j]$ )
        elseif  $\hat{u}[j] < 0.0$ 
            @constraint(model, z_{i+1}[j] == 0.0)
        else
            slope =  $\hat{u}[j] / (\hat{u}[j] - \hat{\ell}[j])$ 
            @constraints(model, begin
                z_{i+1}[j] >=  $\hat{z}[j]$ 
                z_{i+1}[j] <= slope * ( $\hat{z}[j] - \hat{\ell}[j]$ )
                z_{i+1}[j] >= 0.0
            end)
        end
    end
end

```

*Parallel relaxation for given  $\ell_i$  and  $\mathbf{u}_i$*  Parallel relaxation is very similar to  $\Delta$ -relaxation, except that (46c) becomes

$$j \in \Gamma_i \Rightarrow \frac{\hat{u}_{i,j}}{\hat{u}_{i,j} - \hat{\ell}_{i,j}} \hat{z}_{i,j} \leq z_{i,j} \leq \frac{\hat{u}_{i,j}}{\hat{u}_{i,j} - \hat{\ell}_{i,j}} (\hat{z}_{i,j} - \hat{\ell}_{i,j}). \quad (47)$$

Parallel relaxation is used in FastLin for network relaxation before applying reachability methods. To the best of our knowledge, it has not been used directly in any optimization method yet. The implementation of constraint encoding under parallel relaxation is not provided in the paper.

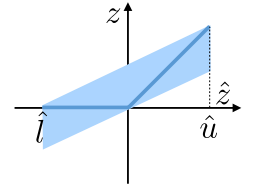


Figure 14. Illustration of parallel relaxation.

*Naive mixed integer linear constraints* The nonlinear constraint (42) can be formulated as a set of linear constraints:

$$z_{i,j} \geq \hat{z}_{i,j}, \quad (48a)$$

$$z_{i,j} \geq 0, \quad (48b)$$

$$z_{i,j} \leq \hat{z}_{i,j} + m\delta_{i,j}, \quad (48c)$$

$$z_{i,j} \leq m(1 - \delta_{i,j}), \quad (48d)$$

where  $m$  should be sufficiently large. If  $m$  is not large enough, the encoding may lead to error. The function is shown in algorithm 6.5. NSVerify calls this function.

```

struct MixedIntegerLP <: AbstractLinearProgram
  m::Float64
end

function encode_layer!(MIP::MixedIntegerLP, model, layer, z_i, z_{i+1}, \delta_{i+1})
  m = MIP.m
  \hat{z} = affine_map(layer, z_i)
  for j in 1:length(layer.bias)
    @constraints(model, begin
      z_{i+1}[j] >= \hat{z}[j]
      z_{i+1}[j] >= 0.0
      z_{i+1}[j] <= \hat{z}[j] + m * \delta_{i+1}[j]
      z_{i+1}[j] <= m - m * \delta_{i+1}[j]
    end)
  end
end

```

Algorithm 6.5. Encoding as mixed integer linear constraints using a sufficiently large number  $m$ . Only the function `encode_layer!` is shown. The function `encode_network!` is the same as in algorithm 6.1.

*Mixed integer linear constraints for given  $\ell_i$  and  $\mathbf{u}_i$*  When we have the bounds, the constraints can be more tightly encoded:

$$z_{i,j} \geq \hat{z}_{i,j}, \quad (49a)$$

$$z_{i,j} \geq 0, \quad (49b)$$

$$z_{i,j} \leq \hat{u}_{i,j}\delta_{i,j}, \quad (49c)$$

$$z_{i,j} \leq \hat{z}_{i,j} - \hat{\ell}_{i,j}(1 - \delta_{i,j}). \quad (49d)$$

The function is shown in algorithm 6.6.

## 6.2 Objective Functions

In primal optimization, there are multiple ways to design the objective. Some of them are listed in algorithm 6.7.

*Violation of linear constraints* In many cases, the objective function is chosen to measure the violation of the constraints. For example, when  $\mathcal{Y}$  is represented by a half space (9), we maximize the following objective function

$$o := \mathbf{c}^\top \mathbf{y} - d. \quad (50)$$

Such design directly tells how much the output constraints can be violated in the problem (4). This objective is used in dual optimization methods, *e.g.*, Duality, ConvDual, and Certify.

```

struct BoundedMixedIntegerLP <: AbstractLinearProgram end

function encode_layer! (::BoundedMixedIntegerLP, model, layer,
                        zi, zi+1, δi+1, bounds)
     $\hat{z}$  = affine_map(layer, zi)
     $\hat{z}$ _bound = approximate_affine_map(layer, bounds)
     $\hat{l}$ ,  $\hat{u}$  = low( $\hat{z}$ _bound), high( $\hat{z}$ _bound)
    for j in 1:length(layer.bias)
        if  $\hat{l}[j] \geq 0.0$ 
            @constraint(model, zi+1[j] ==  $\hat{z}[j]$ )
        elseif  $\hat{u}[j] \leq 0.0$ 
            @constraint(model, zi+1[j] == 0.0)
        else
            @constraints(model, begin
                zi+1[j] >=  $\hat{z}[j]$ 
                zi+1[j] >= 0.0
                zi+1[j] <=  $\hat{u}[j] * \delta_{i+1}[j]$ 
                zi+1[j] <=  $\hat{z}[j] - \hat{l}[j] * (1 - \delta_{i+1}[j])$ 
            end)
        end
    end
end

```

Algorithm 6.6. Encoding as mixed integer linear constraints using the bounds on node values. Only the function `encode_layer!` is shown. The function `encode_network!` is the same as in algorithm 6.1.

*Maximum disturbance* The objective can also measure the maximum allowable disturbance. The disturbance with respect to a given input  $\mathbf{x}_0$  is computed as

$$o := \|\mathbf{x} - \mathbf{x}_0\|_{\infty}. \quad (51)$$

The maximum allowable disturbance is computed by minimizing  $o$  with respect to the constraint that  $\mathbf{f}(\mathbf{x}) \notin \mathcal{Y}$ . This objective is used in MIPVerify and ILP.

*Summation of variables* The objective can also be

$$o := \sum_{i,j} z_{i,j} \text{ or } \sum_{i,j} s_{i,j}. \quad (52)$$

The summation of neurons is used to compute tighter bounds in Planet. The summation of slack variables is used in Planet.

### 6.3 NSVerify

NSVerify[23] takes any linear constraints  $\mathcal{X}$  and  $\mathcal{Y}$ , and considers networks with only ReLU activations. NSVerify encodes the ReLU activation functions as a set of mixed integer linear constraints. It does not need an objective. The method is sound and complete.

Our implementation is shown in algorithm 6.8. The solver needs to specify  $m$  in (48). The solver first initializes variables  $\mathbf{z}_i$  and  $\delta_i$  for all  $i$ . Then the solver adds the input constraint  $\mathbf{z}_0 \in \mathcal{X}$  as well as the complement of the output constraint  $\mathbf{z}_n \notin \mathcal{Y}$ . Then it encodes the network as a set of mixed integer linear constraints. If there is a solution for the optimization, then we get a counter example. If not, the property is satisfied. In the implementation, for simplicity, we require that  $\mathcal{X}$  is an `HPolytope` and  $\mathcal{Y}$  a `PolytopeComplement`.

```

function linear_objective!(model::Model, map::HPolytope, var)
    c, d = tosimplehrep(map)
    o = c * var - d
    @objective(model, Min, o)
    return o
end

function max_disturbance!(model::Model, var)
    o = symbolic_infty_norm(var)
    @objective(model, Min, o)
    return o
end

function min_sum!(model::Model, var)
    o = sum(sum.(var))
    @objective(model, Min, o)
    return o
end

function max_sum!(model::Model, var)
    o = sum(sum.(var))
    @objective(model, Max, o)
    return o
end

```

Algorithm 6.7. Objective functions: zero objective, linear objective, minimax disturbance, minimal summation, and maximal summation.

```

struct NSVerify
    optimizer
    m::Float64
end

function solve(solver::NSVerify, problem::Problem)
    network = problem.network
    model = Model(solver)
    neurons = init_neurons(model, network)
    deltas = init_deltas(model, network)
    add_set_constraint!(model, problem.input, first(neurons))
    add_complementary_set_constraint!(model, problem.output, last(neurons))
    encode_network!(model, network, neurons, deltas, MixedIntegerLP(solver.m))
    feasibility_problem!(model)
    optimize!(model)
    if termination_status(model) == OPTIMAL
        return CounterExampleResult(:violated, value.(first(neurons)))
    end
    if termination_status(model) == INFEASIBLE
        return CounterExampleResult(:holds)
    end
    return CounterExampleResult(:unknown)
end

```

Algorithm 6.8. NSVerify. The verification problem is encoded as a mixed integer linear program. The naive encoding discussed in algorithm 6.5 is used in NSVerify. The parameter  $m$  is specified by the solver.

## 6.4 MIPVerify

MIPVerify [38] can be viewed as a direct extension of NSVerify. It also encodes the network as mixed integer constraints. But the encoding is more efficient since MIPVerify pre-computes the bounds of the problem. In this way, the solver does not need to specify  $m$  as in NSVerify.<sup>19</sup>

Our implementation is shown in algorithm 6.9. Similar to NSVerify, the solver first initializes variables  $\mathbf{z}_i$  and  $\delta_i$  for all  $i$ . Then the solver adds the complement of the output constraint  $\mathbf{z}_n \notin \mathcal{Y}$ . Then it computes the bounds for the neurons, and encodes the network as a set of mixed integer linear constraints. The objective is to compute the maximum allowable disturbance. The satisfiability is determined by comparing the allowable range of disturbance with the input set. In the implementation, for simplicity, we require that  $\mathcal{X}$  is a *Hyperrectangle* and  $\mathcal{Y}$  is a *PolytopeComplement*. MIPVerify supports the max activation function and the  $\ell_p$  norm in the objective, which have not been supported yet in our implementation.

<sup>19</sup> The original code can be found at <https://github.com/vtjeng/MIPVerify.jl>.

Algorithm 6.9. MIPVerify. The verification problem is encoded as a mixed integer linear program. The bounds of node values are considered in the encoding. MIPVerify computes the maximum allowable disturbance.

```

struct MIPVerify
    optimizer
end

function solve(solver::MIPVerify, problem::Problem)
    model = Model(solver)
    neurons = init_neurons(model, problem.network)
    deltas = init_deltas(model, problem.network)
    add_complementary_set_constraint!(model, problem.output, last(neurons))
    bounds = get_bounds(problem)
    encode_network!(model, problem.network, neurons,
                    deltas, bounds, BoundedMixedIntegerLP())
    o = max_disturbance!(model, first(neurons) - problem.input.center)
    optimize!(model)
    if termination_status(model) == INFEASIBLE
        return AdversarialResult(:holds)
    end
    if value(o) >= maximum(problem.input.radius)
        return AdversarialResult(:holds)
    else
        return AdversarialResult(:violated, value(o))
    end
end

```

## 6.5 ILP

ILP [4] encodes ReLU networks as linear constraints. It only considers a linear portion of the network that has the same activation pattern as the reference input. In our implementation, the reference input is chosen as the center of the input constraint set. The resulting problem can be solved by simply encoding a linear program using (43), where  $\delta_{i,j}$ 's denote the activation status of the reference input. To speed up the computation, ILP introduces iterative constraint solving. It first drops all inequality constraints with respect to  $\hat{z}_{i,j}$  in (43), *i.e.*, for all  $i$  and  $j$ , the following constraints are dropped

$$(2\delta_{i,j} - 1)\hat{z}_{i,j} \geq 0. \quad (53)$$

The above expression is a compact version of  $\hat{z}_{i,j} \geq 0$  for  $\delta_{i,j} = 1$  and  $\hat{z}_{i,j} \leq 0$  for  $\delta_{i,j} = 0$ . Without (53), the linear encoding reduces to the relaxed linear encoding in (44). The inequality constraint with respect to  $\hat{z}_{i,j}$  is iteratively added, if the solution at the current iteration violates (53) for any  $i$  and  $j$ .

Our implementation is shown in algorithm 6.10. For simplicity, we require that  $\mathcal{X}$  is a hyperrectangle and  $\mathcal{Y}$  the complement of a polytope, `PolytopeComplement`. The solver first computes the activation  $\delta_i$ 's according to the reference input, *i.e.*, the center of  $\mathcal{X}$ . Then it initializes neuron variables  $\mathbf{z}_i$ 's, adds the complement of the output constraint  $\mathbf{z}_n \notin \mathcal{Y}$ , and adds an objective function for maximum allowable disturbance. We provide both the iterative implementation and the non-iterative implementation to solve the LP problem, where the iterative version corresponds to ILP.<sup>20</sup> In the non-iterative approach, the solver simply encodes the network using the linear constraints in (43). In the iterative approach, the solver first encodes the network using the relaxed linear constraints in (44) and solves the relaxed problem. If the resulting solution violates any inequality constraint (53), we add the constraint to the problem and solve the problem again. The process is repeated until all constraints (53) are satisfied. The process is guaranteed to converge in a finite number of steps since there are only finitely many constraints. The number of constraints equals the number of neurons.

<sup>20</sup> It is claimed in [4] that the iterative approach computes faster than the non-iterative approach.

## 7 Dual Optimization

Primal optimization needs to deal with complicated constraints. Another approach is to consider the dual problem of (41), which can be relaxed to many independent optimization problems [12], [43]. The objective considered in these methods is the violation of output constraints (50). The dual problem provides a valid bound on the violation. In particular, Duality [12] uses Lagrangian relaxation, which handles general activation functions such as ReLU, tanh, sigmoid, maxpool, and etc. ConvDual [43] solves the dual problem of convexified (41), which handles ReLU only. Certify [31] uses semidefinite relaxation, which handles networks with one hidden layer, whose activation functions are differentiable almost everywhere.

### 7.1 Dual Network

This section introduces the concept of *dual network*, which is deeply related to the dual problem of the optimization with respect to a neural network.<sup>21</sup> The term “dual problem” refers to the Lagrangian dual problem, which is obtained by forming the Lagrangian of the optimization, using Lagrange multipliers to add the constraints to the objective function, and then solving for some primal variable values that optimize the Lagrangian. This process will be discussed in detail in section 7.2. The Lagrange multipliers are called dual variables. It will be shown that those dual variables form a dual network, whose structure is similar to the original network but which propagates in the opposite direction. Moreover, the dual variables encode the weights of corresponding nodes in a value function in the context of dynamic programming, if we regard the layer by layer propagation in a neural network as a dynamic system. In the following discussion, we first introduce the dual network in the context of dynamic programming, then point out its relationship with the Lagrangian dual problem.

<sup>21</sup> The dual network discussed here is not the dual neural network (DNN) [52], which is a recurrent neural network (RNN) to solve quadratic programming.

*Dynamic programming: General formulation* Many algorithms optimize an objective function that depends on non-input variables in the neural network, but constrained on the input  $\mathbf{x}$ .

```

struct ILP
    optimizer
    iterative::Bool
end

function solve(solver::ILP, problem::Problem)
    nnet = problem.network
    x = problem.input.center
    model = Model(solver)
     $\delta$  = get_activation(nnet, x)
    neurons = init_neurons(model, nnet)
    add_complementary_set_constraint!(model, problem.output, last(neurons))
    o = max_disturbance!(model, first(neurons) - problem.input.center)
    if !solver.iterative
        encode_network!(model, nnet, neurons,  $\delta$ , StandardLP())
        optimize!(model)
        if (termination_status(model) != OPTIMAL)
            return AdversarialResult(:unknown)
        end
        return interpret_result(solver, value(o), problem.input)
    end
    encode_network!(model, nnet, neurons,  $\delta$ , LinearRelaxedLP())
    while true
        optimize!(model)
        if (termination_status(model) != OPTIMAL)
            return AdversarialResult(:unknown)
        end
        x = value.(first(neurons))
        matched, index = match_activation(nnet, x,  $\delta$ )
        if matched
            return interpret_result(solver, value(o), problem.input)
        end
        add_constraint!(model, nnet, neurons,  $\delta$ , index)
    end
end

function interpret_result(solver::ILP, o, input)
    if o >= maximum(input.radius)
        return AdversarialResult(:holds, o)
    else
        return AdversarialResult(:violated, o)
    end
end

function add_constraint!(model, network
    z::Vector{Vector{VariableRef}},
     $\delta$ ::Vector{Vector{Bool}},
    (i, j)::Tuple{Int64, Int64})
    layer = network.layers[i]
    val = layer.weights[j, :] * z[i] + layer.bias[j]
    if  $\delta$ [i][j]
        @constraint(model, val >= 0.0)
    else
        @constraint(model, val <= 0.0)
    end
end

```

Algorithm 6.10. ILP. ILP computes the maximum allowable disturbance and returns adversarial results. Both iterative and non-iterative approaches are provided. The iterative approach first relaxes inequality constraints in `encode_network!` using linear relaxation, and then iteratively adds those inequality constraints in `add_constraint!`. The function `match_activation` finds the node that is violating the inequality constraint equation (53). Its implementation is not shown.



There is a nonlinear relationship between the objective function and the input  $\mathbf{x}$ . As feedforward neural networks are considered, we can use dynamic programming to simplify the nonlinear optimization problem and obtain the dual problem. Suppose the problem under consideration is

$$\max_{\mathbf{x} \in \mathcal{X}} o := \sum_{i=0}^n o_i(\hat{\mathbf{z}}_i), \quad (54)$$

where  $o_i$  is the objective for different layers.<sup>22</sup> Define the value function at layer  $i$  as

$$o_{i \rightarrow n}(\hat{\mathbf{z}}_i) := \max_{\hat{\mathbf{z}}_{i+1}, \dots, \hat{\mathbf{z}}_n} \sum_{k \geq i} o_k(\hat{\mathbf{z}}_k). \quad (55)$$

The value function represents the optimal value that can be achieved given certain initial state  $\hat{\mathbf{z}}_i$ . Hence, the value function  $o_i$  only has one variable  $\hat{\mathbf{z}}_i$ . The Bellman equation for dynamic programming can be written<sup>23</sup>

$$o_{i \rightarrow n}(\hat{\mathbf{z}}_i) = \max_{\hat{\mathbf{z}}_{i+1} = \mathbf{W}_{i+1} \sigma_i(\hat{\mathbf{z}}_i) + \mathbf{b}_{i+1}} o_i(\hat{\mathbf{z}}_i) + o_{(i+1) \rightarrow n}(\hat{\mathbf{z}}_{i+1}). \quad (56)$$

The Bellman equation can be solved by backward dynamic programming. Then the original optimization (54) is reduced to the following problem,

$$\max_{\mathbf{x} \in \mathcal{X}} o_{0 \rightarrow n}(\mathbf{x}), \quad (57)$$

whose objective only depends on  $\mathbf{x}$ .

This approach is widely used in discrete-time optimal control for dynamic systems where  $\hat{\mathbf{z}}_i$  are states at step  $i$ . In the following discussion, we derive the explicit solution for linear objectives.

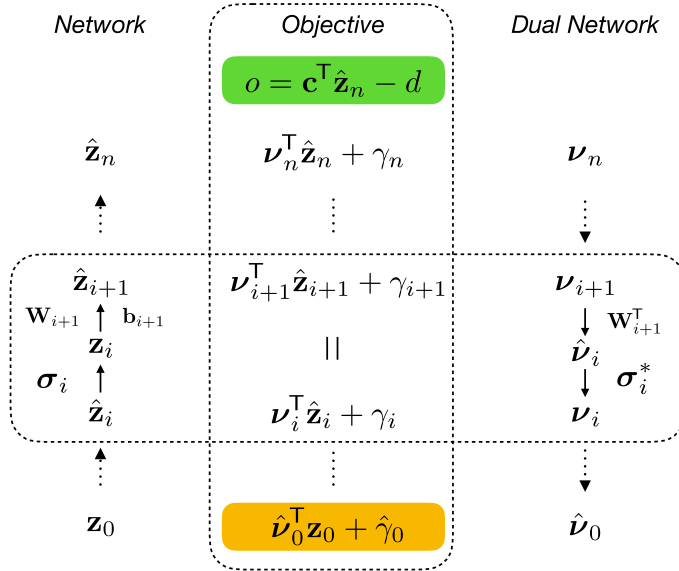


Figure 15. Illustration of dynamic programming and the dual network. Given an objective function  $o$  that depends on the network output, dynamic programming can transform  $o$  to a function that only depends on the network input. In the process, a dual network is constructed. The dual network propagates in the reverse order to the original network. All corresponding mappings are dual functions of the mappings in the original network. Moreover, the nodes  $\mathbf{v}_i$  in the dual network encode the weights of  $\mathbf{z}_i$  to the objective  $o$ . According to the stationary condition in dynamic programming, all expressions in the “objective” block are equivalent to one another. Then the objective function  $o$  that depends on the output (shaded in green) is transformed to a function that only depends on the input (shaded in yellow).

*Dynamic programming: Linear objective* Assume that  $o_i(\hat{\mathbf{z}}_i) = \mathbf{c}_i^T \hat{\mathbf{z}}_i - d_i$  where  $\mathbf{c}_i \in \mathbb{R}^{k_i}$  and  $d_i \in \mathbb{R}$ . Define  $\mathbf{v}_i \in \mathbb{R}^{k_i}$  to be the dual variable for  $\hat{\mathbf{z}}_i$ , and  $\hat{\mathbf{v}}_i \in \mathbb{R}^{k_i}$  to be the dual variable for  $\mathbf{z}_i$ . The dual variables encode the weights of the corresponding nodes in a value function, i.e.,

$$o_{i \rightarrow n}(\hat{\mathbf{z}}_i) = \mathbf{v}_i^T \hat{\mathbf{z}}_i + \gamma_i, \quad (58)$$

where  $\gamma_i \in \mathbb{R}$  is a bias term. Consider  $o_n$ , we have the boundary constraint

$$\mathbf{v}_n = \mathbf{c}_n, \gamma_n = -d_i. \quad (59)$$

Plugging (58) into the Bellman equation (56), we have

$$\mathbf{v}_i^\top \hat{\mathbf{z}}_i + \gamma_i = \mathbf{c}_i^\top \hat{\mathbf{z}}_i - d_i + \mathbf{v}_{i+1}^\top \mathbf{W}_{i+1} \sigma_i(\hat{\mathbf{z}}_i) + \mathbf{v}_{i+1}^\top \mathbf{b}_{i+1} + \gamma_{i+1}. \quad (60)$$

Matching the coefficient of  $\hat{\mathbf{z}}_i$  and the constant term on both sides, we obtain the backward relationship between  $\mathbf{v}_i, \gamma_i$  and  $\mathbf{v}_{i+1}, \gamma_{i+1}$  as

$$\mathbf{v}_i = \mathbf{c}_i + \sigma_i^*(\hat{\mathbf{v}}_i), \quad (61a)$$

$$\hat{\mathbf{v}}_i = \mathbf{W}_{i+1}^\top \mathbf{v}_{i+1}, \quad (61b)$$

$$\gamma_i = \gamma_{i+1} + \mathbf{v}_{i+1}^\top \mathbf{b}_{i+1} - d_i, \quad (61c)$$

where  $\sigma_i^*$  is the dual function of  $\sigma_i$ , defined to satisfy<sup>24</sup>

$$\sigma_i^*(\hat{\mathbf{v}}_i)^\top \hat{\mathbf{z}}_i \equiv \hat{\mathbf{v}}_i^\top \sigma_i(\hat{\mathbf{z}}_i). \quad (62)$$

In this way, the dual variables  $\mathbf{v}_i$  and  $\hat{\mathbf{v}}_i$  indeed form a backward dual network. Figure 15 illustrates the original network, the value function, and the dual network in the case that  $o_i = 0$  for  $i < n$ . Hence, the original optimization problem (54) can be solved by 1) backward computing the dual network from the boundary constraint at layer  $n$  to layer 0, 2) solving the reduced problem (57) that depends on the dual variables.

When  $\sigma_i$  is nonlinear, the dual function  $\sigma_i^*$  is difficult to handle, as it indeed depends on the value of  $\hat{\mathbf{z}}_i$ . In the case of ReLU activation, different approximations of  $\sigma_i$  are introduced to simplify the dual network. For example, ConvDual uses triangle relaxation and FastLin uses parallel relaxation. These approaches are to be introduced in section 7.3 and section 8.2.

*Dual network and Lagrange multipliers* The dual variables for dynamic programming are deeply related to Lagrange multipliers in Lagrangian dual problem. Let  $\mu_i$  be the multiplier for the constraint  $\hat{\mathbf{z}}_i = \mathbf{W}_i \mathbf{z}_{i-1} + \mathbf{b}_i$  and  $\lambda_i$  the multiplier for the constraint  $\mathbf{z}_i = \sigma_i(\hat{\mathbf{z}}_i)$ . Then the Bellman equation (56) can be rewritten as<sup>25</sup>

$$o_{i \rightarrow n}(\hat{\mathbf{z}}_i) = \min_{\mu_{i+1}, \lambda_i} \max_{\hat{\mathbf{z}}_{i+1}, \mathbf{z}_i} o_i(\hat{\mathbf{z}}_i) + o_{(i+1) \rightarrow n}(\hat{\mathbf{z}}_{i+1}) + \mu_{i+1}^\top (\hat{\mathbf{z}}_{i+1} - \mathbf{W}_{i+1} \mathbf{z}_i - \mathbf{b}_{i+1}) + \lambda_i^\top (\mathbf{z}_i - \sigma_i(\hat{\mathbf{z}}_i)). \quad (63)$$

Using the linearity assumption  $o_{(i+1) \rightarrow n}(\hat{\mathbf{z}}_{i+1}) = \mathbf{v}_{i+1}^\top \hat{\mathbf{z}}_{i+1} + \gamma_{i+1}$  in the above equation, we have

$$o_{i \rightarrow n}(\hat{\mathbf{z}}_i) = o_i^* + \min_{\mu_{i+1}, \lambda_i} \max_{\hat{\mathbf{z}}_{i+1}, \mathbf{z}_i} \mathbf{v}_{i+1}^\top \hat{\mathbf{z}}_{i+1} + \mu_{i+1}^\top (\hat{\mathbf{z}}_{i+1} - \mathbf{W}_{i+1} \mathbf{z}_i - \mathbf{b}_{i+1}) + \lambda_i^\top (\mathbf{z}_i - \sigma_i(\hat{\mathbf{z}}_i)), \quad (64)$$

where  $o_i^* = \mathbf{c}_i^\top \hat{\mathbf{z}}_i - d_i + \gamma_{i+1}$ . Rearrange the minimax problem,

$$\min_{\mu_{i+1}, \lambda_i} \max_{\hat{\mathbf{z}}_{i+1}, \mathbf{z}_i} (\mathbf{v}_{i+1} + \mu_{i+1})^\top \hat{\mathbf{z}}_{i+1} + (\lambda_i - \mathbf{W}_{i+1}^\top \mu_{i+1})^\top \mathbf{z}_i - \mu_{i+1}^\top \mathbf{b}_{i+1} - \lambda_i^\top \sigma_i(\hat{\mathbf{z}}_i). \quad (65)$$

By applying the result in footnote 25 to the first two terms in (65), we conclude that  $\mathbf{v}_{i+1} + \mu_{i+1} = \mathbf{0}$  and  $\lambda_i - \mathbf{W}_{i+1}^\top \mu_{i+1} = \mathbf{0}$ . Hence, we have the following relationship

$$\mu_{i+1} = -\mathbf{v}_{i+1}, \lambda_i = -\hat{\mathbf{v}}_i. \quad (66)$$

The nodes in the dual network are indeed the Lagrange multipliers if we do not consider the bounds on  $\mathbf{z}_i$ 's. Duality in section 7.2 provides a formulation that considers the bounds. In this case, the conclusion from footnote 25 does not hold. As a result, the relationship between the dual network and the Lagrange multipliers in (66) breaks.

<sup>24</sup> To obtain an upper bound of the primal problem (54), the dual function only needs to satisfy  $\sigma_i^*(\hat{\mathbf{v}}_i)^\top \hat{\mathbf{z}}_i \geq \hat{\mathbf{v}}_i^\top \sigma_i(\hat{\mathbf{z}}_i)$ .

<sup>25</sup> The solution of the unconstrained problem  $\min_{\mathbf{a}} \max_{\mathbf{b}} \mathbf{a}^\top \mathbf{b}$  is always  $\mathbf{0}$  with  $\mathbf{a} = \mathbf{b} = \mathbf{0}$ . Hence, the optimal solution of the Bellman equation (63) is always  $\mu_{i+1} = \hat{\mathbf{z}}_{i+1} - \mathbf{W}_{i+1} \mathbf{z}_i - \mathbf{b}_{i+1} = \mathbf{0}$  and  $\lambda_i = \mathbf{z}_i - \sigma_i(\hat{\mathbf{z}}_i) = \mathbf{0}$ .

*Dual network and backpropagation* When the objectives for hidden layers are all zero, *i.e.*,  $o_i = 0$  for all  $i < n$ , the dual variable  $\mathbf{v}_i$  is indeed the gradient from the objective  $o$  to the hidden variable  $\hat{\mathbf{z}}_i$ . The gradients are usually computed in backpropagation to train deep neural networks [15].

## 7.2 Duality

Duality [12] takes a hyperrectangle as its input set and has a half space as its output set. The input hyperrectangle is denoted  $|\mathbf{x} - \mathbf{x}_0| \leq \mathbf{r}$ , where  $\mathbf{x}_0$  and  $\mathbf{r}$  are the center and radius of the hyperrectangle. The output half space is denoted  $\mathbf{c}^\top \mathbf{y} \leq d$ . Then the optimization problem becomes

$$\max_{\mathbf{z}_0, \dots, \mathbf{z}_n, \hat{\mathbf{z}}_1, \dots, \hat{\mathbf{z}}_n} \mathbf{c}^\top \mathbf{z}_n - d, \quad (67a)$$

$$\text{s.t. } \mathbf{z}_i = \sigma_i(\hat{\mathbf{z}}_i), \forall i \in \{1, \dots, n\}, \quad (67b)$$

$$\hat{\mathbf{z}}_i = \mathbf{W}_i \mathbf{z}_{i-1} + \mathbf{b}_i, \forall i \in \{1, \dots, n\}, \quad (67c)$$

$$|\mathbf{z}_0 - \mathbf{x}_0| \leq \mathbf{r}. \quad (67d)$$

Given the bounds on  $\mathbf{z}_i$  and  $\hat{\mathbf{z}}_i$ , the optimal value of (67) is bounded by Lagrangian relaxation of the constraints

$$\max_{\mathbf{z}_0, \dots, \mathbf{z}_n, \hat{\mathbf{z}}_1, \dots, \hat{\mathbf{z}}_n} \mathbf{c}^\top \mathbf{z}_n - d + \sum_{i=1}^n \mu_i^\top (\hat{\mathbf{z}}_i - \mathbf{W}_i \mathbf{z}_{i-1} - \mathbf{b}_i) + \sum_{i=1}^n \lambda_i^\top (\mathbf{z}_i - \sigma_i(\hat{\mathbf{z}}_i)), \quad (68a)$$

$$\text{s.t. } \ell_i \leq \mathbf{z}_i \leq \mathbf{u}_i, \forall i \in \{1, \dots, n\}, \quad (68b)$$

$$\hat{\ell}_i \leq \hat{\mathbf{z}}_i \leq \hat{\mathbf{u}}_i, \forall i \in \{1, \dots, n\}, \quad (68c)$$

$$|\mathbf{z}_0 - \mathbf{x}_0| \leq \mathbf{r}, \quad (68d)$$

where  $\lambda_i \in \mathbb{R}^{k_i}$  and  $\mu \in \mathbb{R}^{k_i}$  are Lagrange multipliers. For any choice of  $\lambda$  and  $\mu$ , (68) provides a valid upper bound on the optimal value of (67). This property is known as weak duality.

Since the objective and constraints are separable in the layers, the variables in each layer can be optimized independently. The boundary condition is  $\lambda_n = -\mathbf{c}$ . The objective function in (68) can be decomposed into the following three parts:

- Input layer value with respect to  $\mathbf{z}_0$ .

$$f_0(\mu_1) = \max_{|\mathbf{z}_0 - \mathbf{x}_0| \leq \mathbf{r}} -\mu_1^\top (\mathbf{W}_1 \mathbf{z}_0 + \mathbf{b}_1), \quad (69a)$$

$$= -\mu_1^\top \mathbf{W}_1 \mathbf{x}_0 - \mu_1^\top \mathbf{b}_1 + \left| \mu_1^\top \mathbf{W}_1 \right| \mathbf{r}. \quad (69b)$$

- Layer value with respect to  $\mathbf{z}_i$ . For  $i \in \{1, \dots, n-1\}$ ,

$$f_i(\lambda_i, \mu_{i+1}) = \max_{\ell_i \leq \mathbf{z}_i \leq \mathbf{u}_i} -\mu_{i+1}^\top (\mathbf{W}_{i+1} \mathbf{z}_i + \mathbf{b}_{i+1}) + \lambda_i^\top \mathbf{z}_i, \quad (70a)$$

$$= \left( \lambda_i - \mathbf{W}_{i+1}^\top \mu_{i+1} \right)^\top \frac{\ell_i + \mathbf{u}_i}{2} - \mu_{i+1}^\top \mathbf{b}_{i+1} + \left| \lambda_i - \mathbf{W}_{i+1}^\top \mu_{i+1} \right|^\top \frac{\mathbf{u}_i - \ell_i}{2}. \quad (70b)$$

- Activation value with respect to  $\hat{\mathbf{z}}_i$ . For  $i \in \{1, \dots, n\}$ ,

$$\tilde{f}_i(\lambda_i, \mu_i) = \max_{\hat{\ell}_i \leq \hat{\mathbf{z}}_i \leq \hat{\mathbf{u}}_i} \mu_i^\top \hat{\mathbf{z}}_i - \lambda_i^\top \sigma_i(\hat{\mathbf{z}}_i), \quad (71a)$$

$$\leq \sum_j \max \{ \mu_{i,j} \hat{\ell}_{i,j}, \mu_{i,j} \hat{u}_{i,j} \} + \max \{ \lambda_{i,j} \sigma_{i,j}(\hat{\ell}_{i,j}), \lambda_{i,j} \sigma_{i,j}(\hat{u}_{i,j}) \}, \quad (71b)$$

where the inequality is taken by considering element-wise maximum.

Any choice of the dual variables  $\lambda_i, \mu_i$  in (68) provides an upper bound of the primal problem (67). To obtain a tight bound, we need to minimize (68) with respect to the dual variables. Hence, the dual problem can be constructed as

$$\min_{\lambda_1, \dots, \lambda_n, \mu_1, \dots, \mu_n} f_0(\mu_1) + \sum_{i=1}^{n-1} f_i(\lambda_i, \mu_{i+1}) + \sum_{i=1}^n \tilde{f}_i(\lambda_i, \mu_i) - d, \quad (72a)$$

$$\text{s.t. } \lambda_n = -\mathbf{c}. \quad (72b)$$

The problem is satisfied if the optimal solution of (72) is negative; otherwise, it is not satisfied. Algorithm 7.1 provides an implementation.

### 7.3 ConvDual

ConvDual [43] takes a hypercube input set and a half space output set. The input hypercube is denoted  $\|\mathbf{x} - \mathbf{x}_0\|_\infty \leq \epsilon$ , where  $\mathbf{x}_0$  and  $\epsilon$  are the center and radius of the hypercube. The output half space is denoted  $\mathbf{c}^\top \mathbf{y} \leq d$ . ConvDual only considers ReLU networks. It first relaxes the constraint using  $\Delta$ -relaxation. The primal optimization becomes a convex problem:<sup>26</sup>

$$\max_{\mathbf{z}_0, \dots, \mathbf{z}_n, \hat{\mathbf{z}}_1, \dots, \hat{\mathbf{z}}_n} \mathbf{c}^\top \hat{\mathbf{z}}_n - d, \quad (73a)$$

$$\text{s.t. } z_{i,j} = \hat{z}_{i,j}, \forall i \in \{1, \dots, n-1\}, j \in \Gamma_i^+, \quad (73b)$$

$$z_{i,j} = 0, \forall i \in \{1, \dots, n-1\}, j \in \Gamma_i^-, \quad (73c)$$

$$z_{i,j} \geq \hat{z}_{i,j}, z_{i,j} \geq 0, z_{i,j} \leq \frac{\hat{u}_{i,j}(\hat{z}_{i,j} - \hat{\ell}_{i,j})}{\hat{u}_{i,j} - \hat{\ell}_{i,j}}, \forall i \in \{1, \dots, n-1\}, j \in \Gamma_i, \quad (73d)$$

$$\hat{\mathbf{z}}_i = \mathbf{W}_i \mathbf{z}_{i-1} + \mathbf{b}_i, \forall i \in \{1, \dots, n\}, \quad (73e)$$

$$\|\mathbf{z}_0 - \mathbf{x}_0\|_\infty \leq \epsilon. \quad (73f)$$

<sup>26</sup> ConvDual considers the pre-activation bounds of the last layer, which is equivalent to the case that the activation in the last layer is identity.

The optimization (73) can be regarded as an  $n$ -step dynamic program. We derive the dual problem and the dual network following the procedure of dynamic programming discussed in section 7.1. The original paper [43] directly applies the dual problem formulation.

Recall that  $\mathbf{v}_i$  and  $\hat{\mathbf{v}}_i$  are the dual variables for  $\hat{\mathbf{z}}_i$  and  $\mathbf{z}_i$ . The boundary condition is  $\mathbf{v}_n = \mathbf{c}$  and  $\gamma_n = -d$ . The Bellman equation (60) is reduced to

$$\mathbf{v}_i^\top \hat{\mathbf{z}}_i + \gamma_i = \hat{\mathbf{v}}_i^\top \mathbf{z}_i + \mathbf{v}_{i+1}^\top \mathbf{b}_{i+1} + \gamma_{i+1}. \quad (74)$$

To obtain the backward relationship between  $\mathbf{v}_i, \gamma_i$  and  $\mathbf{v}_{i+1}, \gamma_{i+1}$  similar to (61), we consider the following three cases. The sets  $\Gamma_i^+, \Gamma_i^-$ , and  $\Gamma_i$  encode activation conditions as defined in (24).

- For active node  $j \in \Gamma_i^+$ .  
 $z_{i,j} = \hat{z}_{i,j}$ . Hence,  $v_{i,j} = \hat{v}_{i,j}$ .
- For inactive node  $j \in \Gamma_i^-$ .  
 $z_{i,j} = 0$ . Hence,  $v_{i,j} = 0$ .
- For undetermined node  $j \in \Gamma_i$ . Consider the  $\Delta$ -relaxation in (73d), the following conditions are satisfied,

$$\hat{v}_{i,j} z_{i,j} \leq \frac{\hat{v}_{i,j} \hat{u}_{i,j}}{\hat{u}_{i,j} - \hat{\ell}_{i,j}} \hat{z}_{i,j} - \frac{\hat{v}_{i,j} \hat{u}_{i,j} \hat{\ell}_{i,j}}{\hat{u}_{i,j} - \hat{\ell}_{i,j}} \quad \text{for } \hat{v}_{i,j} \geq 0, \quad (75a)$$

$$\hat{v}_{i,j} z_{i,j} \leq \alpha_{i,j} \hat{v}_{i,j} \hat{z}_{i,j} \quad \text{for } \hat{v}_{i,j} < 0, \quad (75b)$$

```

struct Duality
    optimizer
end

function solve(solver::Duality, problem::Problem)
    model = Model(solver)
    c, d = tosimplehrep(problem.output)
     $\lambda$  = init_multipliers(model, problem.network)
     $\mu$  = init_multipliers(model, problem.network)
    o = dual_value(solver, problem, model,  $\lambda$ ,  $\mu$ )
    @constraint(model, last( $\lambda$ ) .== -c)
    optimize!(model)
    return interpret_result(solver, termination_status(model), o - d[1])
end

function dual_value(solver::Duality, problem, model,  $\lambda$ ,  $\mu$ )
    bounds, layers = get_bounds(problem), problem.network.layers
    o = input_layer_value(layers[1],  $\mu$ [1], bounds[1])
    o += activation_value(layers[1],  $\mu$ [1],  $\lambda$ [1], bounds[1])
    for i in 2:length(layers)
        o += layer_value(layers[i],  $\mu$ [i],  $\lambda$ [i-1], bounds[i])
        o += activation_value(layers[i],  $\mu$ [i],  $\lambda$ [i], bounds[i])
    end
    @objective(model, Min, o)
    return o
end

function activation_value(layer::Layer,  $\mu_i$ ,  $\lambda_i$ , bound)
    o = zero(eltype( $\mu_i$ ))
    b_hat = approximate_affine_map(layer, bound)
    l_hat, u_hat = low(b_hat), high(b_hat)
    l, u = layer.activation(l_hat), layer.activation(u_hat)
    o += sum(symbolic_max( $\mu_i$  .* l_hat,  $\mu_i$  .* u_hat))
    o += sum(symbolic_max( $\lambda_i$  .* l,  $\lambda_i$  .* u))
    return o
end

function layer_value(layer::Layer,  $\mu_i$ ,  $\lambda_i$ , bound)
    c, r = bound.center, bound.radius
    o =  $\lambda_i$  * c -  $\mu_i$  * affine_map(layer, c)
    o += sum(symbolic_abs( $\lambda_i$  .- layer.weights *  $\mu_i$ ) .* r)
    return o
end

function input_layer_value(layer::Layer,  $\mu_i$ , input)
    W = layer.weights
    c, r = input.center, input.radius
    o = - $\mu_i$  * affine_map(layer, c)
    o += sum(symbolic_abs( $\mu_i$  * W) .* r)
    return o
end

```

Algorithm 7.1. Duality. Lagrangian relaxation is considered in Duality. The variables to be optimized are the Lagrange multipliers. The dual value is computed layer-wise. The function `input_layer_value` corresponds to equation (69). The function `layer_value` corresponds to equation (70). The function `activation_value` corresponds to equation (71). Duality returns basic satisfiability results.

where  $\alpha_{i,j} \in [0, 1]$ . (75) implies the following condition:

$$\hat{v}_{i,j} z_{i,j} \leq \frac{\hat{u}_{i,j}}{\hat{u}_{i,j} - \hat{\ell}_{i,j}} [\hat{v}_{i,j}]_+ \hat{z}_{i,j} + \alpha_{i,j} [\hat{v}_{i,j}]_- \hat{z}_{i,j} - \frac{\hat{u}_{i,j} \hat{\ell}_{i,j}}{\hat{u}_{i,j} - \hat{\ell}_{i,j}} [\hat{v}_{i,j}]_+. \quad (76)$$

Hence, the relationship among the dual variables that maximizes the value is

$$\gamma_i = \mathbf{v}_{i+1}^\top \mathbf{b}_{i+1} + \gamma_{i+1} - \sum_{j \in \Gamma_i} \frac{\hat{u}_{i,j} \hat{\ell}_{i,j}}{\hat{u}_{i,j} - \hat{\ell}_{i,j}} [\hat{v}_{i,j}]_+, \quad (77a)$$

$$v_{i,j} = \begin{cases} \hat{v}_{i,j} & j \in \Gamma_i^+ \\ 0 & j \in \Gamma_i^- \\ \frac{\hat{u}_{i,j}}{\hat{u}_{i,j} - \hat{\ell}_{i,j}} [\hat{v}_{i,j}]_+ + \alpha_{i,j} [\hat{v}_{i,j}]_- & j \in \Gamma_i \end{cases}, \quad (77b)$$

for  $i \in \{1, \dots, n-1\}$ . Note that when  $j \in \Gamma_i$  and  $v_{i,j} > 0$ ,  $v_{i,j} = \frac{\hat{u}_{i,j}}{\hat{u}_{i,j} - \hat{\ell}_{i,j}} [\hat{v}_{i,j}]_+$ . Then the term  $\frac{\hat{u}_{i,j} \hat{\ell}_{i,j}}{\hat{u}_{i,j} - \hat{\ell}_{i,j}} [\hat{v}_{i,j}]_+$  can be simplified as  $\hat{\ell}_{i,j} [v_{i,j}]_+$ .

Then optimizing  $\mathbf{v}_n^\top \hat{\mathbf{z}}_n + \gamma_n$  is equivalent to optimizing  $\mathbf{v}_1^\top \hat{\mathbf{z}}_1 + \gamma_1 = \hat{\mathbf{v}}_0^\top \mathbf{z}_0 + \mathbf{v}_1^\top \mathbf{b}_1 + \gamma_1$ . Hence,

$$\max_{\|\mathbf{z}_0 - \mathbf{x}_0\|_\infty \leq \epsilon} \hat{\mathbf{v}}_0^\top \mathbf{z}_0 + \mathbf{v}_1^\top \mathbf{b}_1 + \gamma_1, \quad (78a)$$

$$= \hat{\mathbf{v}}_0^\top \mathbf{x}_0 + \epsilon \|\hat{\mathbf{v}}_0\|_1 + \underbrace{\sum_{i=1}^n \mathbf{v}_i^\top \mathbf{b}_i - \sum_{i=1}^{n-1} \sum_{j \in \Gamma_i} \hat{\ell}_{i,j} [v_{i,j}]_+}_{\mathbf{v}_1^\top \mathbf{b}_1 + \gamma_1} - d, \quad (78b)$$

where  $\gamma_1$  is computed according to (77a).

Then the dual problem of (73) is

$$\min_{\alpha_{i,j} \in [0,1]} \hat{\mathbf{v}}_0^\top \mathbf{x}_0 + \epsilon \|\hat{\mathbf{v}}_0\|_1 + \sum_{i=1}^n \mathbf{v}_i^\top \mathbf{b}_i - \sum_{i=1}^{n-1} \sum_{j \in \Gamma_i} \hat{\ell}_{i,j} [v_{i,j}]_+ - d, \quad (79a)$$

$$\text{s.t. } \mathbf{v}_n = \mathbf{c}, \quad (79b)$$

$$\hat{\mathbf{v}}_i = \mathbf{W}_{i+1}^\top \mathbf{v}_{i+1}, \forall i \in \{0, \dots, n-1\}, \quad (79c)$$

$$v_{i,j} = \begin{cases} 0 & j \in \Gamma_i^- \\ \hat{v}_{i,j} & j \in \Gamma_i^+ \\ \frac{\hat{u}_{i,j}}{\hat{u}_{i,j} - \hat{\ell}_{i,j}} [\hat{v}_{i,j}]_+ + \alpha_{i,j} [\hat{v}_{i,j}]_- & j \in \Gamma_i \end{cases} \quad \forall i \in \{1, \dots, n-1\}. \quad (79d)$$

The optimal solution of (79) provides an upper bound of the primal problem (73). If the dual value is smaller than 0, then the problem is satisfiable. The dual network consists of  $\mathbf{v}_i$ 's and is almost identical to the back-propagation network. The difference is that for nodes  $j \in \mathcal{I}_i$ , there is the additional free variable  $\alpha_{i,j}$  that we can optimize over. One fixed and dual feasible solution of (79) is

$$\alpha_{i,j} = \frac{\hat{u}_{i,j}}{\hat{u}_{i,j} - \hat{\ell}_{i,j}}. \quad (80)$$

Under the condition, the dual network is linear. We will show in section 8.2 that the resulting dual network is identical to the case where we use parallel relaxation instead of  $\Delta$  relaxation.

Our implementation is in algorithm 7.2. The value is computed in `dual_value`. The dual network is computed in `backprop!` layer by layer. For simplicity, we directly use (80) in the implementation. The bounds  $\hat{\ell}_{i,j}$  and  $\hat{u}_{i,j}$  and the sets  $\Gamma_i$ ,  $\Gamma_i^-$ , and  $\Gamma_i^+$  can be computed using different methods, *e.g.*, interval arithmetic introduced in section 4.1 and section 5.4. The original paper computes the bounds by formulating the bounding problem as several optimization problems similar to (73). The objectives are  $\min z_{i,j}$  and  $\max z_{i,j}$  for all  $i$  and  $j$ . Those problems can also be solved by dynamic programming. Moreover, the bounds can be computed inductively layer by layer. The details of the computation will be discussed in FastLin in section 8.2.

#### 7.4 Certify

Certify [31] computes over-approximated certificates for a neural network with only one hidden layer. Similar to ConvDual, Certify also takes a hypercube input set  $\|\mathbf{x} - \mathbf{x}_0\|_\infty \leq \epsilon$  and a half space output set  $\mathbf{c}^\top \mathbf{y} \leq d$ . It works for any activation function as long as the function is differentiable almost everywhere and its gradient is bounded. For simplicity, we assume that the derivative of the activation function is bounded by 0 and 1, *i.e.*,  $\sigma'(\hat{z}) \in [0, 1]$  for any  $\hat{z}$ . If not, we can always scale it by a factor  $\max_{\hat{z}} \sigma'(\hat{z})$  and add the factor in the following derivation. The primal optimization problem considered in Certify is

$$\max_{\|\mathbf{x} - \mathbf{x}_0\|_\infty \leq \epsilon} o(\mathbf{x}) = \mathbf{c}^\top \mathbf{f}(\mathbf{x}) - d. \quad (81)$$

Since  $o$  is differentiable almost-everywhere, then

$$o(\mathbf{x}) \leq o(\mathbf{x}_0) + \epsilon \max_{\|\mathbf{x} - \mathbf{x}_0\|_\infty \leq \epsilon} \|\nabla o(\mathbf{x})\|_1. \quad (82)$$

For a neural network with only one hidden layer,  $o(\mathbf{x}) = \sum_j c_j \sigma_{1,j}(\mathbf{w}_{1,j} \mathbf{x} + b_{1,j})$ . For simplicity, we omit the first index (index for layer 1) in  $\sigma$ ,  $\mathbf{w}$  and  $b$ . Then

$$\nabla o(\mathbf{x}) = \sum_j c_j \sigma'_j(\mathbf{w}_j \mathbf{x} + b_j) \mathbf{w}_j^\top = \mathbf{W}^\top \text{diag}(\mathbf{c}) \nabla \sigma(\mathbf{W} \mathbf{x} + \mathbf{b}), \quad (83)$$

where  $\nabla \sigma \in [0, 1]^{k_1}$  is the vertical stack of  $\sigma'_j$ . Let  $\mathbf{s} = \sigma'(\mathbf{W} \mathbf{x} + \mathbf{b})$ . Then

$$\|\nabla o(\mathbf{x})\|_1 \leq \max_{\mathbf{s} \in [0, 1]^{k_1}, \mathbf{t} \in [-1, 1]^{k_0}} \mathbf{t}^\top \mathbf{W}^\top \text{diag}(\mathbf{c}) \mathbf{s}, \quad (84a)$$

$$= \max_{\mathbf{s} \in [-1, 1]^{k_1}, \mathbf{t} \in [-1, 1]^{k_0}} \frac{1}{2} \mathbf{t}^\top \mathbf{W}^\top \text{diag}(\mathbf{c}) (\mathbf{1} + \mathbf{s}), \quad (84b)$$

$$= \max_{\mathbf{p} \in [-1, 1]^{1+k_1+k_0}} \frac{1}{4} \mathbf{p}^\top \underbrace{\begin{bmatrix} 0 & 0 & \mathbf{1}^\top \mathbf{W}^\top \text{diag}(\mathbf{c}) \\ 0 & 0 & \mathbf{W}^\top \text{diag}(\mathbf{c}) \\ \text{diag}(\mathbf{c}) \mathbf{W} \mathbf{1} & \text{diag}(\mathbf{c}) \mathbf{W} & 0 \end{bmatrix}}_{\mathbf{M}(\mathbf{c}, \mathbf{W})} \mathbf{p}, \quad (84c)$$

$$= \max_{\mathbf{p} \in [-1, 1]^{1+k_1+k_0}} \frac{1}{4} \langle \mathbf{M}(\mathbf{c}, \mathbf{W}), \mathbf{p} \mathbf{p}^\top \rangle, \quad (84d)$$

$$\leq \max_{\mathbf{p} \geq 0, p_{jj} \leq 1} \frac{1}{4} \langle \mathbf{M}(\mathbf{c}, \mathbf{W}), \mathbf{P} \rangle. \quad (84e)$$

```

struct ConvDual end

function solve(solver::ConvDual, problem::Problem)
    o = dual_value(solver, problem.network, problem.input, problem.output)
    if o >= 0.0
        return BasicResult(:holds)
    end
    return BasicResult(:unknown)
end

function dual_value(solver::ConvDual, network, input, output)
    layers = network.layers
    L, U = get_bounds(network, input.center, input.radius[1])
    v0, d = tosimplehrep(output)
    v = vec(v0)
    o = d[1]
    for i in reverse(1:length(layers))
        o -= v'*layers[i].bias
        v = layers[i].weights'*v
        if i>1
            o += backprop!(v, U[i-1], L[i-1])
        end
    end
    o -= input.center' * v + input.radius[1] * sum(abs.(v))
    return o
end

function backprop!(v::Vector{Float64}, u::Vector{Float64}, l::Vector{Float64})
    o = 0.0
    for j in 1:length(v)
        val = relaxed_ReLU(l[j], u[j])
        if val < 1.0
            v[j] = v[j] * val
            o += v[j] * l[j]
        end
    end
    return o
end

function relaxed_ReLU(l::Float64, u::Float64)
    u <= 0.0 && return 0.0
    l >= 0.0 && return 1.0
    return u / (u - l)
end

```

Algorithm 7.2. ConvDual. The dual problem of the verification problem is considered.  $\Delta$  relaxation is applied. The dual problem is formulated by backward dynamic programming. The function `dual_value` explicitly computes the dual value equation (79) under the relaxation equation (80).



The inequality in (84a) is due to  $\|\mathbf{q}\|_1 \leq \max_{\|\mathbf{p}\|_\infty \leq 1} \mathbf{p}^\top \mathbf{q}$ . (84b) changes the range of  $\mathbf{s}$ . (84c) packs all variables into one vector  $\mathbf{p} := [1, \mathbf{t}, \mathbf{s}]$  and packs all parameters into the matrix  $\mathbf{M}(\mathbf{c}, \mathbf{W})$ . (84d) exploits the equivalence between matrix trace and quadratic form. The inner product between two matrices is defined as  $\langle \mathbf{M}, \mathbf{P} \rangle := \text{tr}(\mathbf{M}^\top \mathbf{P})$ . (84e) is obtained by taking  $\mathbf{P} := \mathbf{p}\mathbf{p}$ . The matrix  $\mathbf{P}$  is positive semidefinite, *i.e.*,  $\mathbf{P}^\top \succeq 0$ . Moreover, the diagonal term  $P_{jj} = p_j^2 \leq 1$  for  $j \in \{1, \dots, 1 + k_0 + k_1\}$ . Hence, the convex semidefinite relaxation of the value  $o(\mathbf{x})$  is

$$\max_{\|\mathbf{x} - \mathbf{x}_0\|_\infty \leq \epsilon} o(\mathbf{x}) \leq o(\mathbf{x}_0) + \frac{\epsilon}{4} \max_{\mathbf{P} \succeq 0, P_{jj} \leq 1} \langle \mathbf{M}(\mathbf{c}, \mathbf{W}), \mathbf{P} \rangle. \quad (85)$$

The right-hand side of (85) provides an upper bound of (81). If the bound is smaller than zero, then the problem is satisfiable. It is worth noting that the semidefinite program (the max function) only depends on  $\mathbf{c}$  and  $\mathbf{W}$ . As it does not depend on the input  $\mathbf{x}$ , it only needs to be computed once for a problem.

Our implementation is shown in algorithm 7.3, which directly constructs and solves the semidefinite program in (85).

```

struct Certify
    optimizer
end

function solve(solver::Certify, problem::Problem)
    model = Model(solver)
    c, d = tosimplehrep(problem.output)
    v = c * problem.network.layers[2].weights
    W = problem.network.layers[1].weights
    M = getM(v[1, :], W)
    n = size(M, 1)
    P = @variable(model, [1:n, 1:n], PSD)
    output = c * compute_output(problem.network, problem.input.center) .- d[1]
    epsilon = problem.input.radius[1]
    o = output .+ epsilon/4 * tr(M*P)
    @constraint(model, diag(P) .<= ones(n))
    @objective(model, Max, first(o))
    optimize!(model)
    return interpret_result(solver, termination_status(model), first(o))
end

```

Algorithm 7.3. Certify. It solves the semidefinite program equation (85).

## 8 Search and Reachability

This section discusses methods that combine layer-by-layer reachability analysis with search. ReluVal [40] uses symbolic intervals for reachability analysis and then searches the input domain for potential violations using iterative interval refinement. FastLin [41] uses network approximation for reachability analysis and then uses binary search to estimate a certified lower bound of maximum allowable disturbance. FastLip [41] is built upon FastLin, which further estimates the local Lipschitz constant. DLV [18] performs layer-by-layer search in hidden layers for potential counter examples. Note that the return types of these methods are not necessarily reachability results.

### 8.1 ReluVal

ReluVal [40] takes a hyperrectangle input set  $|\mathbf{x} - \mathbf{x}_0| \leq \mathbf{r}$  and any output set that implements the abstract polytope type. The reachability analysis is done symbolically, while the search is done through iterative interval refinement. Our implementation is shown in algorithms 8.1 to 8.3.

```

struct SymbolicInterval
  Low::Matrix{Float64}
  Up::Matrix{Float64}
  interval::Hyperrectangle
end

struct SymbolicIntervalMask
  sym::SymbolicInterval
  LA::Vector{Vector{Int64}}
  UA::Vector{Vector{Int64}}
end

```

Algorithm 8.1. The data structure in ReluVal. `SymbolicInterval` represents the symbolic interval defined in equation (86). `SymbolicIntervalMask` further includes the lower and upper bounds of  $\nabla \sigma_i$ , i.e., diagonal entries of the gradient bounds  $\underline{\mathbf{A}}_i$  and  $\overline{\mathbf{A}}_i$  introduced in section 4.3.

*Symbolic interval propagation* Reachability methods that use interval arithmetic usually provide very loose bounds, as they do not keep track of dependencies among the hidden nodes. Symbolic interval propagation can provide tighter bounds by keeping track of those dependencies layer by layer. Define the extended input as  $\mathbf{x}^e := [\mathbf{x}, 1]$ . Then a symbolic interval at layer  $i$  is defined as

$$\mathbf{z}_i \in [\mathbf{L}_i \mathbf{x}^e, \mathbf{U}_i \mathbf{x}^e], \text{ for } \mathbf{x} \in [\mathbf{x}_0 - \mathbf{r}, \mathbf{x}_0 + \mathbf{r}], \quad (86)$$

where  $\mathbf{L}_i, \mathbf{U}_i \in \mathbb{R}^{k_i \times (k_0+1)}$  are coefficients in the symbolic interval. For example, the  $j$ th node at layer  $i$  is bounded by

$$\ell_{i,j} \mathbf{x}^e \leq z_{i,j} \leq \mathbf{u}_{i,j} \mathbf{x}^e, \quad (87)$$

where  $\ell_{i,j}$  and  $\mathbf{u}_{i,j}$  are the  $j$ th row in  $\mathbf{L}_i$  and  $\mathbf{U}_i$  respectively.

The data structure `SymbolicInterval` in algorithm 8.1 is introduced to keep track of symbolic intervals, where the field `Low` corresponds to  $\mathbf{L}_i^e$ , `Up` to  $\mathbf{U}_i^e$ , and `interval` to the hyperrectangle  $[\mathbf{x}_0 - \mathbf{r}, \mathbf{x}_0 + \mathbf{r}]$ .

Given the symbolic interval,  $\mathbf{L}_i \mathbf{x}^e$  belongs to a hyperrectangle that centers at  $\mathbf{L}_i [\mathbf{x}_0, 1]$  with radius  $|\mathbf{L}_i|[\mathbf{r}, 0]$ , where  $|\mathbf{L}_i|$  is a matrix containing the element-wise absolute values of  $\mathbf{L}_i$ . Similarly,  $\mathbf{U}_i \mathbf{x}^e$  belongs to a hyperrectangle that centers at  $\mathbf{U}_i [\mathbf{x}_0, 1]$  with radius  $|\mathbf{U}_i|[\mathbf{r}, 0]$ . Let  $\mathbf{a} \mathbf{x}^e$  be a symbolic representation for a scalar variable, where  $\mathbf{a} \in \mathbb{R}^{k_0+1}$ . Let  $\underline{h}, \bar{h} : \mathbb{R}^{k_0+1} \rightarrow \mathbb{R}$  be functions that map the symbolic representation  $\mathbf{a} \mathbf{x}^e$  to its concrete lower bound and upper bound respectively, such that

$$\underline{h}(\mathbf{a}) := \mathbf{a}[\mathbf{x}_0, 1] - |\mathbf{a}|[\mathbf{r}, 0], \quad (88a)$$

$$\bar{h}(\mathbf{a}) := \mathbf{a}[\mathbf{x}_0, 1] + |\mathbf{a}|[\mathbf{r}, 0]. \quad (88b)$$

The symbolic intervals are computed layer by layer by calling `forward_network` in algorithm 5.1. The function `forward_layer` is shown in algorithm 8.2, which consists of the following two steps:

- Symbolic interval propagation through the linear map  $\hat{\mathbf{z}}_i = \mathbf{W}_i \mathbf{z}_{i-1} + \mathbf{b}_i$  for  $i \in \{2, \dots, n\}$ .

$$\hat{\mathbf{L}}_i = [\mathbf{W}_i]_+ \mathbf{L}_{i-1} + [\mathbf{W}_i]_- \mathbf{U}_{i-1} + [\mathbf{0} \ \mathbf{b}_i], \quad (89a)$$

$$\hat{\mathbf{U}}_i = [\mathbf{W}_i]_+ \mathbf{U}_{i-1} + [\mathbf{W}_i]_- \mathbf{L}_{i-1} + [\mathbf{0} \ \mathbf{b}_i]. \quad (89b)$$

For the first layer, the symbolic bounds are defined as horizontal concatenation of  $\mathbf{W}_1$  and  $\mathbf{b}_1$ ,

$$\hat{\mathbf{L}}_1 = \hat{\mathbf{U}}_1 = [\mathbf{W}_1 \ \mathbf{b}_1]. \quad (90)$$

This corresponds to `forward_linear` in algorithm 8.2.

- Symbolic interval propagation through ReLU activation function  $\mathbf{z}_i = [\hat{\mathbf{z}}_i]_+$ . For each node  $j$ , there are three possibilities: always active ( $j \in \Gamma_i^+$ ), never active ( $j \in \Gamma_i^-$ ), undetermined ( $j \in \Gamma_i$ ). Similar to (24), we have

$$\Gamma_i^+ = \{j : \underline{h}(\hat{\ell}_{i,j}) \geq 0\}, \quad (91a)$$

$$\Gamma_i^- = \{j : \bar{h}(\hat{\mathbf{u}}_{i,j}) \leq 0\}, \quad (91b)$$

$$\Gamma_i = \{j : j \notin \Gamma_i^+ \cup \Gamma_i^-\}. \quad (91c)$$

Then the symbolic interval for node  $j$  is computed as

$$j \in \Gamma_i^+ \Rightarrow \ell_{i,j} = \hat{\ell}_{i,j}, \mathbf{u}_{i,j} = \hat{\mathbf{u}}_{i,j}, \quad (92a)$$

$$j \in \Gamma_i^- \Rightarrow \ell_{i,j} = \mathbf{u}_{i,j} = \mathbf{0}, \quad (92b)$$

$$j \in \Gamma_i \Rightarrow \ell_{i,j} = \mathbf{0}, \mathbf{u}_{i,j} = \begin{cases} \hat{\mathbf{u}}_{i,j} & \underline{h}(\hat{\mathbf{u}}_{i,j}) \geq 0 \\ [\mathbf{0} \ \bar{h}(\hat{\mathbf{u}}_{i,j})] & \underline{h}(\hat{\mathbf{u}}_{i,j}) < 0 \end{cases}. \quad (92c)$$

When the node is always active in (92a), we keep the symbolic dependency on the input variables. When the node is never active in (92b), all outputs should be 0. When the activation is undetermined in (92c), the lower bound is set to 0. If it is possible for the symbolic upper bound to be smaller than 0, the input dependencies will be thrown away. The upper bound is set to be its concrete maximum, *i.e.*, the first  $k_0$  entries in  $\mathbf{u}_{i,j}$  are set to zero. Otherwise, we keep the symbolic dependency of the upper bound on input variables.

This corresponds to `forward_act` in algorithm 8.2.

Given the symbolic interval propagation, the output reachable set is a hyper-rectangle such that

$$\tilde{\mathcal{R}} = \{\mathbf{y} : y_j \in [\underline{h}(\ell_{n,j}), \bar{h}(\mathbf{u}_{n,j})], \forall j = 1, \dots, k_n\}. \quad (93)$$

The output reachable set computed by symbolic interval propagation is tighter than simple interval arithmetic as illustrated in figure 7.

Given the reachable set  $\tilde{\mathcal{R}}$ , its relationship with respect to the output set  $\mathcal{Y}$  is examined in the function `check_inclusion` in algorithm 8.3, which may return the following four different results:

- The return status is `:holds` if  $\tilde{\mathcal{R}} \subset \mathcal{Y}$ .
- The return status is `:violated` if  $\tilde{\mathcal{R}} \cap \mathcal{Y} = \emptyset$ .
- In addition, we sample the input interval to check for counter examples.<sup>27</sup> If the output with respect to any sample point does not belong to  $\mathcal{Y}$ , the sample point is returned as an unsatisfied `CounterExampleResult`.

<sup>27</sup> Heuristically, only the middle point of the input interval is checked in both the original implementation and our implementation.

- Otherwise, the return status is :`Undetermined`.

If the status is undetermined, ReluVal performs iterative interval refinement to minimize over-approximation in  $\tilde{\mathcal{R}}$ .

*Iterative interval refinement* Although symbolic interval propagation can provide us with tighter bounds than simple interval arithmetic, it may still have significant over-approximation, especially when the input intervals are comparably large. Recall that MaxSens partitions the input intervals into smaller sets to minimize over approximations. ReluVal performs iterative interval refinement instead, which splits the intervals of input nodes according to their influence on the output.<sup>28</sup>

We evaluate the influence by considering the bounds on gradients  $\underline{\mathbf{G}}_n$  and  $\overline{\mathbf{G}}_n$  defined in section 4, which essentially measure the sensitivity of the output with respect to each input feature. The bounds  $\underline{\mathbf{G}}_n$  and  $\overline{\mathbf{G}}_n$  can be obtained by calling the third `get_gradient` in algorithm 4.3. The bounds  $\underline{\Lambda}_i$  and  $\overline{\Lambda}_i$  on  $\nabla \sigma_i$  are computed in the forward propagation, and recorded in the data structure `SymbolicIntervalMask` in algorithm 8.1.

For each refinement step, ReluVal bisects the interval for input node  $j$  that has the highest smear value

$$S_j := \sum_k \max\{|UG_{n,j,k}|, |LG_{n,j,k}|\} r_k, \quad (94)$$

where  $UG_{n,j,k}$  and  $LG_{n,j,k}$  are the  $j$ th row and  $k$ th column entries in  $\overline{\mathbf{G}}_n$  and  $\underline{\mathbf{G}}_n$ , respectively. Let  $j^* = \arg \max_j S_j$ . The smear values are computed in `get_smear_index` in algorithm 8.3, which returns the index to split. The split is performed by `interval_split` in algorithm 4.2.

The main procedure is shown in algorithm 8.3 and illustrated in figure 16. We first compute the reachable set without splitting the interval. If the status of the reachability analysis is undetermined, iterative interval refinement will be performed. A list of reachable sets is maintained. The reachable sets correspond to different input intervals. At each iteration, a set in the list is picked out according to the tree search strategy specified by the solver (default depth first search). For the picked set, the bounds of its gradient are computed. Then it is split into two intervals according to the smear values. The reachable sets for the two smaller intervals are then computed. If a reachable set belongs to the output constraint, we drop the corresponding interval. If a counter example is found in an interval, we conclude that the problem is violated. Otherwise, we need to further split the interval whose reachable set is then pushed back to the list. If the list becomes empty, the property is verified to be hold.

## 8.2 FastLin

FastLin [41] computes the certified lower bound of the maximum allowable disturbance.<sup>29</sup> The method combines reachability analysis with binary search to estimate the bound, and returns an adversarial result. FastLin only works for ReLU activation functions. A general method that works for any activation function, called CROWN, is discussed in [51], but is not reviewed here.

The binary search procedure shown in the main function in algorithm 8.5 can be combined with any reachability method. In particular, FastLin computes the bounds based on linear approximation of the neurons. The method takes a hyperrectangle as input set and a polytope or the complement of a polytope as output set.

<sup>28</sup> The ReluVal paper discusses two approaches: baseline iterative refinement and optimizing iterative refinement. We consider optimizing iterative refinement.

<sup>29</sup> An earlier work [42] called CLEVER also estimates the lower bound. However, CLEVER is not sound, hence, is not reviewed here.

```

function forward_layer(solver::ReluVal, layer::Layer, input)
    return forward_act(forward_linear(input, layer))
end

function forward_linear(input::Hyperrectangle, layer::Layer)
    (W, b) = (layer.weights, layer.bias)
    sym = SymbolicInterval(hcat(W, b), hcat(W, b), input)
    LA = Vector{Vector{Int64}}(undef, 0)
    UA = Vector{Vector{Int64}}(undef, 0)
    return SymbolicIntervalMask(sym, LA, UA)
end

function forward_linear(input::SymbolicIntervalMask, layer::Layer)
    (W, b) = (layer.weights, layer.bias)
    output_Up = max.(W, 0) * input.sym.Up + min.(W, 0) * input.sym.Low
    output_Low = max.(W, 0) * input.sym.Low + min.(W, 0) * input.sym.Up
    output_Up[:, end] += b
    output_Low[:, end] += b
    sym = SymbolicInterval(output_Low, output_Up, input.sym.interval)
    return SymbolicIntervalMask(sym, input.LA, input.UA)
end

function forward_act(input::SymbolicIntervalMask, layer::Layer{ReLU})
    n_node, n_input = size(input.sym.Up)
    output_Low, output_Up = input.sym.Low[:, :], input.sym.Up[:, :]
    mask_lower, mask_upper = zeros(Int, n_node), ones(Int, n_node)
    interval = input.sym.interval
    for i in 1:n_node
        if upper_bound(input.sym.Up[i, :], interval) <= 0.0
            mask_lower[i], mask_upper[i] = 0, 0
            output_Up[i, :] = zeros(n_input)
            output_Low[i, :] = zeros(n_input)
        elseif lower_bound(input.sym.Low[i, :], interval) >= 0
            mask_lower[i], mask_upper[i] = 1, 1
        else
            mask_lower[i], mask_upper[i] = 0, 1
            output_Low[i, :] = zeros(n_input)
            if lower_bound(input.sym.Up[i, :], interval) < 0
                output_Up[i, :] = zeros(n_input)
                output_Up[i, end] = upper_bound(input.sym.Up[i, :], interval)
            end
        end
    end
    sym = SymbolicInterval(output_Low, output_Up, interval)
    LA = push!(input.LA, mask_lower)
    UA = push!(input.UA, mask_upper)
    return SymbolicIntervalMask(sym, LA, UA)
end

```

Algorithm 8.2. Symbolic interval propagation in ReluVal. The function `forward_layer` is called by `forward_network` in the layer-by-layer propagation. The function `forward_layer` consists of `forward_linear` for the linear mapping and `forward_act` for the nonlinear activation. The first `forward_linear` function is for the input layer, while the second `forward_linear` function is for the following layers. Except for the first layer, the data structure being passed through layers is `SymbolicIntervalMask`.

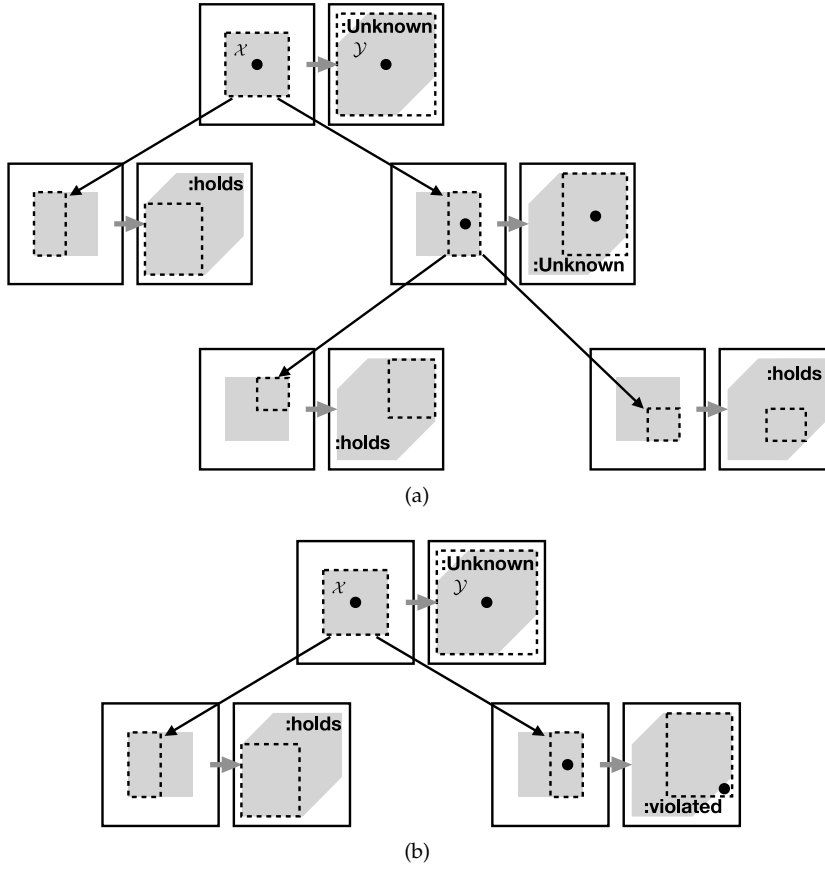


Figure 16. Illustration of iterative interval refinement. The two sub-figures illustrate two search trees in two different scenarios. Each node in the search tree consists of two boxes connected by an arrow. The left box corresponds to the input space and the right box corresponds to the output space. The input constraint and the output constraint are shaded in corresponding spaces. The dashed box in the input space represents the refined interval, while the dashed box in the output space represents the reachable set for that interval. If the reachable set belongs to  $\mathcal{Y}$ , then the status is holds. If the reachable set overlaps with  $\mathcal{Y}$  but is not a subset of  $\mathcal{Y}$ , the status is unknown. In this case, a point is sampled in the input space (black dot). If the output of that sample point does not belong to  $\mathcal{Y}$ , the status is violated. Otherwise, the interval is split into two. (a) Scenario 1: property holds as the reachable sets all belong to  $\mathcal{Y}$ . (b) Scenario 2: property violated as a counter example is found.

```

struct ReluVal
  tree_search::Symbol
end

function solve(solver::ReluVal, problem::Problem)
  reach = forward_network(solver, problem.network, problem.input)
  result = check_inclusion(reach.sym, problem.output, problem.network)
  result.status == :Unknown || return result
  reach_list = SymbolicIntervalMask[reach]
  while true
    length(reach_list) > 0 || return BasicResult(:holds)
    reach = pick_out!(reach_list, solver.tree_search)
    LG, UG = get_gradient(problem.network, reach.LA, reach.UA)
    feature = get_smear_index(problem.network, reach.sym.interval, LG, UG)
    intervals = split_interval(reach.sym.interval, feature)
    for interval in intervals
      reach = forward_network(solver, problem.network, interval)
      result = check_inclusion(reach.sym, problem.output, problem.network)
      result.status == :violated && return result
      result.status == :holds || (push!(reach_list, reach))
    end
  end
end

function check_inclusion(reach::SymbolicInterval, output, network)
  reachable = symbol_to_concrete(reach)
  issubset(reachable, output) && return BasicResult(:holds)
  middle_point = (high(reach.interval) + low(reach.interval))./2
  y = compute_output(network, middle_point)
  ∈(y, output) || return CounterExampleResult(:violated, middle_point)
  return BasicResult(:Unknown)
end

```

Algorithm 8.3. The main function in ReluVal. The reachable set for a given input interval is computed by calling `forward_network`, which then calls `forward_layer`. In order to perform iterative interval refinement, the `solve` function keeps track of a list of reachable sets that correspond to different input intervals. At each iteration, a set in the list is picked out according to the tree search strategy specified by the solver by `pick_out!`. For the picked set, the bounds of its gradient are computed by `get_gradient`. Then the index to be split is determined by `get_smear_index`. The interval is split into two by `split_interval`. The reachable sets for the two smaller intervals are then computed. The `check_inclusion` checks the status of each reachable set. If a counter example is found, the result is directly returned. If the status is unknown, we need to further split the input interval. The corresponding reachable set is pushed back to the list. If the list becomes empty, the problem is verified.

*Reachability via network relaxation* Given the bounds from interval arithmetic, a ReLU node that has an undetermined activation can be approximated using parallel relaxation.<sup>30</sup> Given the bounds  $\ell_i$  and  $\mathbf{u}_i$  for  $i \leq k$ , the bounds  $\hat{\ell}_{k+1}$  and  $\hat{\mathbf{u}}_{k+1}$  can be computed by directly optimizing the node values. Similar to the derivation in ConvDual, we derive the lower and upper bounds using backward dynamic programming.<sup>31</sup> The optimization and dynamic programming approach will provide tighter bounds than interval arithmetic discussed in section 4.1, since the correlation among nodes in the hidden layers is considered in the optimization problem, but not in interval arithmetic algorithms.

<sup>30</sup> Parallel relaxation is looser than  $\Delta$ -relaxation.

<sup>31</sup> The FastLin paper provides a different derivation.

The bounds are initiated as  $\ell_0 = \mathbf{x}_0 - \epsilon \mathbf{1}$  and  $\mathbf{u}_0 = \mathbf{x}_0 + \epsilon \mathbf{1}$ . Given the bounds  $\ell_i$  and  $\mathbf{u}_i$  for  $i \leq k$ , we show the derivation for  $\ell_{k+1}$  and  $\mathbf{u}_{k+1}$ . For simplicity and without loss of generality, we assume that the  $k+1$ th layer only contains one node. The derivation below can be easily extended to the case with multiple nodes. The lower and upper bounds of node  $\hat{z}_{k+1}$  are given by

$$\hat{\ell}_{k+1} = \min_{\|\mathbf{x}-\mathbf{x}_0\| \leq \epsilon} \hat{z}_{k+1}, \hat{\mathbf{u}}_{k+1} = \max_{\|\mathbf{x}-\mathbf{x}_0\| \leq \epsilon} \hat{z}_{k+1}. \quad (95)$$

For simplicity, we only show the detailed derivation of  $\hat{\mathbf{u}}_{k+1}$ . Recall that  $\mathbf{v}_i$  and  $\hat{\mathbf{v}}_i = \mathbf{W}_{i+1} \mathbf{v}_{i+1}$  are the dual variables for  $\hat{\mathbf{z}}_i$  and  $\mathbf{z}_i$ , and  $\gamma_i$  is the bias in the value function. The boundary conditions are  $v_{k+1} = 1$  and  $\gamma_{k+1} = 0$ . Then the equation in (74) follows. The sets  $\Gamma_i^+$ ,  $\Gamma_i^-$ , and  $\Gamma_i$  for  $i \leq k$  can be constructed using the known bounds according to (24). For  $j \in \Gamma_i^+$ ,  $v_{i,j} = \hat{v}_{i,j}$ . For  $\Gamma_i^-$ ,  $v_{i,j} = 0$ . These two cases are identical to the cases in ConvDual in section 7.3. For  $j \in \Gamma_i$ , consider the parallel relaxation (47) instead of  $\Delta$ -relaxation, we have

$$\hat{v}_{i,j} z_{i,j} \leq \frac{\hat{v}_{i,j} \hat{\mathbf{u}}_{i,j}}{\hat{\mathbf{u}}_{i,j} - \hat{\ell}_{i,j}} \hat{z}_{i,j} - \frac{\hat{v}_{i,j} \hat{\mathbf{u}}_{i,j} \hat{\ell}_{i,j}}{\hat{\mathbf{u}}_{i,j} - \hat{\ell}_{i,j}} \quad \text{for } \hat{v}_{i,j} \geq 0, \quad (96a)$$

$$\hat{v}_{i,j} z_{i,j} \leq \frac{\hat{v}_{i,j} \hat{\mathbf{u}}_{i,j}}{\hat{\mathbf{u}}_{i,j} - \hat{\ell}_{i,j}} \hat{z}_{i,j} \quad \text{for } \hat{v}_{i,j} < 0, \quad (96b)$$

which is different from (75) in that there is no degree of freedom for a slack variable  $\alpha$ . In this way, the relationship among the dual variables that maximizes the value is that for  $i \in \{1, \dots, k\}$ ,

$$\gamma_i = \mathbf{v}_{i+1}^\top \mathbf{b}_{i+1} + \gamma_{i+1} - \sum_{j \in \Gamma_i} \hat{\ell}_{i,j} [v_{i,j}]_+, \quad (97a)$$

$$v_{i,j} = \begin{cases} \hat{v}_{i,j} & j \in \Gamma_i^+ \\ 0 & j \in \Gamma_i^- \\ \frac{\hat{\mathbf{u}}_{i,j}}{\hat{\mathbf{u}}_{i,j} - \hat{\ell}_{i,j}} \hat{v}_{i,j} & j \in \Gamma_i \end{cases}. \quad (97b)$$

The resulting dual network defined by  $v_{i,j}$ 's is identical to the dual network in ConvDual when  $\alpha_{i,j}$  is chosen according to (80). For simplification, define a diagonal matrix  $\mathbf{D}_i \in \mathbb{R}^{k_i \times k_i}$  whose diagonal entries  $d_{i,j,j}$  for all  $j$  satisfy

$$d_{i,j,j} = \begin{cases} 1 & j \in \Gamma_i^+ \\ 0 & j \in \Gamma_i^- \\ \frac{\hat{\mathbf{u}}_{i,j}}{\hat{\mathbf{u}}_{i,j} - \hat{\ell}_{i,j}} & j \in \Gamma_i \end{cases}. \quad (98)$$

Then the dual network satisfies

$$\mathbf{v}_i = \mathbf{D}_i \mathbf{W}_{i+1}^\top \mathbf{v}_{i+1}, \forall i \leq k. \quad (99)$$



The dual variables provide an upper bound of  $\hat{z}_{k+1}$  such that  $\hat{z}_{k+1} \leq \mathbf{v}_1^T \hat{\mathbf{z}}_1 + \gamma_1 = \hat{\mathbf{v}}_0^T \mathbf{x} + \mathbf{v}_1^T \mathbf{b}_1 + \gamma_1$  where  $\mathbf{x}$  satisfies the  $\ell_p$  bounds  $\|\mathbf{x} - \mathbf{x}_0\|_p \leq \epsilon$ . According to (97a) and the boundary condition  $\gamma_{k+1} = 0$ ,

$$\mathbf{v}_1^T \mathbf{b}_1 + \gamma_1 = \sum_{i=1}^n \mathbf{v}_i^T \mathbf{b}_i - \sum_{i=1}^{n-1} \sum_{j \in \Gamma_i} \hat{\ell}_{i,j}[v_{i,j}]_+ =: \mu^+. \quad (100)$$

Hence, the upper bound of  $\hat{z}_{k+1}$  is

$$\hat{u}_{k+1} := \max_{\|\mathbf{x} - \mathbf{x}_0\|_p \leq \epsilon} \hat{\mathbf{v}}_0^T \mathbf{x} + \mu^+ = \hat{\mathbf{v}}_0^T \mathbf{x}_0 + \epsilon \|\hat{\mathbf{v}}_0\|_q + \mu^+, \quad (101)$$

where  $q$  is the dual variable for  $p$ , i.e.,  $p^{-1} + q^{-1} = 1$ . Our implementation only considers the case where  $p = \infty$  and  $q = 1$ .

Similarly, the lower bound of  $\hat{z}_{k+1}$  can be computed as

$$\hat{\ell}_{k+1} := \min_{\|\mathbf{x} - \mathbf{x}_0\|_p \leq \epsilon} \hat{\mathbf{v}}_0^T \mathbf{x} + \mu^- = \hat{\mathbf{v}}_0^T \mathbf{x}_0 - \epsilon \|\hat{\mathbf{v}}_0\|_q + \mu^-, \quad (102)$$

where

$$\mu^- = \sum_{i=1}^n \mathbf{v}_i^T \mathbf{b}_i - \sum_{i=1}^{n-1} \sum_{j \in \Gamma_i} \hat{\ell}_{i,j}[v_{i,j}]_-. \quad (103)$$

We are using the same dual network for both the upper bound and the lower bound.

The above process should be performed iteratively from  $k = 0$  to  $k = n - 1$ . If there are multiple nodes in layer  $k + 1$ , we need to combine all dual variables together. Denote the variables in the dual network and the value function for the  $j$ th node in layer  $k + 1$  as  $\mathbf{v}_i^{k+1,j}$  and  $\gamma_i^{k+1,j}$  for  $i \leq k + 1$ . The derivation of  $\mathbf{v}_i^{k+1,j}$  and  $\gamma_i^{k+1,j}$  follows the discussion above, except for the boundary conditions. The boundary condition for the dual network is now  $\mathbf{v}_{k+1}^{k+1,j} = [0, \dots, 0, 1, 0, \dots, 0]$  where the value is 1 for the  $j$ th entry and 0 otherwise. When we optimize the value for the  $j$ th node, the objective is  $(\mathbf{v}_{k+1}^{k+1,j})^T \mathbf{z}_{k+1}$ . The boundary condition for the bias is kept unchanged  $\gamma_{k+1}^{k+1,j} = 0$ . Define a vector  $\gamma_i^{k+1} := [\gamma_i^{k+1,1}, \dots, \gamma_i^{k+1,k_{k+1}}] \in \mathbb{R}^{k_{k+1}}$ . Define a matrix  $\mathbf{V}_i^{k+1} \in \mathbb{R}^{k_i \times k_{k+1}}$  to be the horizontal concatenation of  $\mathbf{v}_i^{k+1,j}$  for all  $j \in \{1, \dots, k_{k+1}\}$ , i.e.,

$$\mathbf{V}_i^{k+1} = [\mathbf{v}_i^{k+1,1} \quad \mathbf{v}_i^{k+1,2} \quad \dots \quad \mathbf{v}_i^{k+1,k_{k+1}}]. \quad (104)$$

The matrices  $\mathbf{V}_i^{k+1}$  for all  $i \leq k + 1$  also have a network structure. According to (99), the network structure can be described by

$$\mathbf{V}_i^{k+1} = \mathbf{D}_i \mathbf{W}_{i+1}^T \mathbf{V}_{i+1}^{k+1}, \forall i \leq k, \quad (105)$$

with boundary condition

$$\mathbf{V}_{k+1}^{k+1} = \mathbf{I}. \quad (106)$$

Following (101) and (102), the upper and lower bounds for  $\hat{z}_{k+1}$  can be computed as

$$\hat{u}_{k+1} = (\mathbf{V}_1^{k+1})^T \mathbf{W}_1 \mathbf{x}_0 + \epsilon \|(\mathbf{V}_1^{k+1})^T \mathbf{W}_1\|_q + \sum_{i=1}^n (\mathbf{V}_i^{k+1})^T \mathbf{b}_i - \sum_{i=1}^{n-1} [(\mathbf{V}_i^{k+1})]_+^T \mathbf{D}_i^* \hat{\ell}_i, \quad (107a)$$

$$\hat{\ell}_{k+1} = (\mathbf{V}_1^{k+1})^T \mathbf{W}_1 \mathbf{x}_0 - \epsilon \|(\mathbf{V}_1^{k+1})^T \mathbf{W}_1\|_q + \sum_{i=1}^n (\mathbf{V}_i^{k+1})^T \mathbf{b}_i - \sum_{i=1}^{n-1} [(\mathbf{V}_i^{k+1})]_-^T \mathbf{D}_i^* \hat{\ell}_i, \quad (107b)$$

where  $\mathbf{D}_i^*$  is a diagonal matrix whose  $j$ th diagonal entry is 1 for  $j \in \Gamma_i$  and 0 otherwise. The  $q$  norm is taken row-wise, i.e.,  $\|(\mathbf{V}_1^{k+1})^T \mathbf{W}_1\|_q := [\|(\mathbf{v}_1^{k+1,1})^T \mathbf{W}_1\|_q, \dots, \|(\mathbf{v}_1^{k+1,k_{k+1}})^T \mathbf{W}_1\|_q]$ .

*Relationships among dual networks* For every  $k$ , we indeed have a dual network of  $k$  hidden layers. The dual variables in the dual network are related. The correlations are derived below. Note that  $\mathbf{V}_i^{k+1} = \mathbf{D}_i \mathbf{W}_{i+1}^\top \mathbf{V}_{i+1}^{k+1} = \mathbf{D}_i \mathbf{W}_{i+1}^\top \mathbf{D}_{i+1} \mathbf{W}_{i+2}^\top \cdots \mathbf{D}_k \mathbf{W}_{k+1}^\top$ . And  $\mathbf{V}_i^k = \mathbf{D}_i \mathbf{W}_{i+1}^\top \mathbf{V}_{i+1}^k = \mathbf{D}_i \mathbf{W}_{i+1}^\top \mathbf{D}_{i+1} \mathbf{W}_{i+2}^\top \cdots \mathbf{D}_{k-1} \mathbf{W}_k^\top$ . Hence,

$$\mathbf{V}_i^{k+1} = \mathbf{V}_i^k \mathbf{D}_k \mathbf{W}_{k+1}^\top. \quad (108)$$

The relationships among all  $\mathbf{V}_i^{k'}$ s are shown below.

$$\begin{array}{ccccccc} & & & & & & \mathbf{V}_n^n \\ & & & & & & \downarrow \mathbf{D}_{n-1} \mathbf{W}_n^\top \\ & & & & \mathbf{V}_{n-1}^{n-1} & \xrightarrow{\cdot \mathbf{D}_{n-1} \mathbf{W}_n^\top} & \mathbf{V}_{n-1}^n \\ & & & & \vdots & & \vdots \\ & & & & \vdots & & \vdots \\ & & & & \mathbf{V}_3^{n-1} & \xrightarrow{\cdot \mathbf{D}_{n-1} \mathbf{W}_n^\top} & \mathbf{V}_3^n \\ & & & & \downarrow \mathbf{D}_2 \mathbf{W}_3^\top & & \downarrow \mathbf{D}_2 \mathbf{W}_3^\top \\ & & & & \mathbf{V}_2^{n-1} & \xrightarrow{\cdot \mathbf{D}_{n-1} \mathbf{W}_n^\top} & \mathbf{V}_2^n \\ & & & & \downarrow \mathbf{D}_1 \mathbf{W}_2^\top & & \downarrow \mathbf{D}_1 \mathbf{W}_2^\top \\ & & & & \mathbf{V}_1^{n-1} & \xrightarrow{\cdot \mathbf{D}_{n-1} \mathbf{W}_n^\top} & \mathbf{V}_1^n \\ & & & & \vdots & & \vdots \\ & & & & \mathbf{V}_1^1 & \xrightarrow{\cdot \mathbf{D}_1 \mathbf{W}_2^\top} & \mathbf{V}_1^2 \\ & & & & \downarrow \mathbf{D}_1 \mathbf{W}_2^\top & & \downarrow \mathbf{D}_1 \mathbf{W}_2^\top \\ & & & & \mathbf{V}_1^2 & \xrightarrow{\cdot \mathbf{D}_2 \mathbf{W}_3^\top} & \mathbf{V}_1^3 \\ & & & & \downarrow \mathbf{D}_2 \mathbf{W}_3^\top & & \downarrow \mathbf{D}_2 \mathbf{W}_3^\top \\ & & & & \mathbf{V}_1^3 & \xrightarrow{\cdot \mathbf{D}_3 \mathbf{W}_4^\top} & \mathbf{V}_1^4 \\ & & & & \vdots & & \vdots \end{array} \quad (109)$$

There are two approaches to compute  $\mathbf{V}_i^k$  for different  $i$  and  $k$ .

1. To compute the dual network from scratch for every  $k$ .
2. To compute the dual network for  $k+1$  by reusing the dual network from the last  $k$  following (108).

The first approach is used in the original implementation in FastLin. The second approach is used in the original implementation in ConvDual. The second approach is more computationally efficient, and is shown in algorithm 8.4.

*Binary search* The binary search process of the allowable disturbance  $\epsilon$  is illustrated in figure 17. We keep track of lower and upper bounds of  $\epsilon$ , denoted  $\underline{\epsilon}$  and  $\bar{\epsilon}$ . The output reachable set for input set with radius  $\epsilon$  is denoted  $\mathcal{R}(\epsilon)$ , which satisfies the bounds computed in (107). The variables  $\underline{\epsilon}$  and  $\bar{\epsilon}$  need to satisfy that

$$\mathcal{R}(\underline{\epsilon}) \subseteq \mathcal{Y}, \mathcal{R}(\bar{\epsilon}) \not\subseteq \mathcal{Y}. \quad (110)$$

At every search step, the reachable set  $\mathcal{R}(\epsilon)$  for  $\epsilon := \frac{\underline{\epsilon} + \bar{\epsilon}}{2}$  is computed. There are two possibilities.

- If  $\mathcal{R}(\epsilon) \subseteq \mathcal{Y}$ , then  $\underline{\epsilon}$  is updated to be  $\epsilon$ .
- If  $\mathcal{R}(\epsilon) \not\subseteq \mathcal{Y}$ , then  $\bar{\epsilon}$  is updated to be  $\epsilon$ .

The search process can be terminated if either the maximum iteration is reached or the bounds are close enough to each other, i.e.,  $\bar{\epsilon} - \underline{\epsilon}$  is small. The binary search is implemented in the main loop in algorithm 8.5.

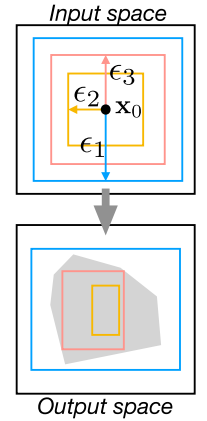


Figure 17. Illustration of the binary search in FastLin. The upper plot shows three different input sets with radius  $\epsilon_1$ ,  $\epsilon_2$ , and  $\epsilon_3$ . The lower plot shows the output constraint  $\mathcal{Y}$  (gray area), and the output reachable sets that correspond to the three input sets. The first reachable set  $\mathcal{R}(\epsilon_1)$  does not belong to  $\mathcal{Y}$ . Then a smaller radius  $\epsilon_2$  is chosen. The resulting reachable set  $\mathcal{R}(\epsilon_2)$  is a subset of  $\mathcal{Y}$ . Then a larger radius  $\epsilon_3 \in (\epsilon_2, \epsilon_1)$  is chosen. This process is repeated.

```

function get_bounds(nnet::Network, input::Vector{Float64}, ε::Float64)
    layers = nnet.layers
    n_layer = length(layers)
    l = Vector{Vector{Float64}}{ }
    u = Vector{Vector{Float64}}{ }
    γ = Vector{Vector{Float64}}{ }
    μ = Vector{Vector{Vector{Float64}}}{ }
    v1 = layers[1].weights'
    push!(γ, layers[1].bias)
    l1, u1 = input_layer_bounds(layers[1], input, ε)
    push!(l, l1)
    push!(u, u1)
    for i in 2:n_layer
        n_input = length(layers[i-1].bias)
        n_output = length(layers[i].bias)
        input_ReLU = relaxed_ReLU.(last(l), last(u))
        D = Diagonal(input_ReLU)
        WD = layers[i].weights*D
        v1 = v1 * WD'
        map!(g -> WD*g, γ, γ)
        for M in μ
            map!(m -> WD*m, M, M)
        end
        push!(γ, layers[i].bias)
        push!(μ, new_μ(n_input, n_output, input_ReLU, WD))
        ψ = v1' * input + sum(γ)
        eps_v1_sum = ε * vec(sum(abs, v1, dims = 1))
        neg, pos = all_neg_pos_sums(input_ReLU, l, μ, n_output)
        push!(l, ψ - eps_v1_sum + neg )
        push!(u, ψ + eps_v1_sum - pos )
    end
    return l, u
end

```

Algorithm 8.4. Reachability analysis in FastLin via network relaxation. The variables in the  $(k + 1)$ th dual network is computed by reusing the variables in the  $k$ th dual network.

```

struct FastLin
    maxIter::Int64
     $\epsilon_0$ ::Float64
    accuracy::Float64
end

function solve(solver::FastLin, problem::Problem)
     $\epsilon_{\text{upper}}$  = 2 * max(solver. $\epsilon_0$ , maximum(problem.input.radius))
     $\epsilon$  = fill(maximum(problem.input.radius), solver.maxIter+1)
     $\epsilon_{\text{lower}}$  = 0.0
    n_input = dim(problem.input)
    for i = 1:solver.maxIter
        input_bounds = Hyperrectangle(problem.input.center, fill( $\epsilon[i]$ , n_input))
        output_bounds = forward_network(solver, problem.network, input_bounds)
        if issubset(output_bounds, problem.output)
             $\epsilon_{\text{lower}}$  =  $\epsilon[i]$ 
             $\epsilon[i+1]$  = ( $\epsilon[i]$  +  $\epsilon_{\text{upper}}$ ) / 2
            abs( $\epsilon[i]$  -  $\epsilon[i+1]$ ) > solver.accuracy || break
        else
             $\epsilon_{\text{upper}}$  =  $\epsilon[i]$ 
             $\epsilon[i+1]$  = ( $\epsilon[i]$  +  $\epsilon_{\text{lower}}$ ) / 2
        end
    end
    if  $\epsilon_{\text{lower}}$  > maximum(problem.input.radius)
        return AdversarialResult(:holds,  $\epsilon_{\text{lower}}$ )
    else
        return AdversarialResult(:violated,  $\epsilon_{\text{lower}}$ )
    end
end

```

Algorithm 8.5. FastLin. This main function shows the binary search process to determine the maximum allowable disturbance in the input space.

### 8.3 FastLip

FastLip [41] estimates the local Lipschitz constant, which is equivalent to what Certify computes in (84). Certify considers only networks with one hidden layer, while FastLip deals with networks with multiple layers. Same as in FastLin, FastLip takes in a hyperrectangle input set and a polytope output set. For simplicity, we assume the output set is only a half space defined by  $\mathbf{c}^\top \mathbf{y} \leq d$ . FastLip optimizes the following problem,

$$\min_{\mathbf{x}} \epsilon, \quad (111a)$$

$$\text{s.t. } \|\mathbf{x} - \mathbf{x}_0\|_q \geq \epsilon, \mathbf{y} = \mathbf{f}(\mathbf{x}), \mathbf{c}^\top \mathbf{y} \geq d. \quad (111b)$$

Let  $o(\mathbf{x}) = \mathbf{c}^\top \mathbf{y} - d = \mathbf{c}^\top \mathbf{f}(\mathbf{x}) - d$ . Assume  $\epsilon$  satisfies the above optimization, then  $o(\mathbf{x}) \geq 0$  for some  $\mathbf{x}$  with  $\|\mathbf{x} - \mathbf{x}_0\|_q = \epsilon$ . Similar to (81), we have the following relationship

$$0 \leq o(\mathbf{x}) \leq o(\mathbf{x}_0) + \epsilon \max_{\|\mathbf{x} - \mathbf{x}_0\|_q \leq \epsilon} \|\nabla o\|_p. \quad (112)$$

Hence, the solution of the problem (111) is bounded by

$$\min_{\|\mathbf{x} - \mathbf{x}_0\|_q \geq \epsilon, o(\mathbf{x}) \geq 0} \epsilon \geq -\frac{o(\mathbf{x}_0)}{\max_{\|\mathbf{x} - \mathbf{x}_0\|_q \leq \epsilon} \|\nabla o\|_p}. \quad (113)$$

The max over  $\|\nabla o\|_p$  is hard to compute without knowing the optimal solution  $\epsilon$  on the left hand side. In practice, the max over  $\|\nabla o\|_p$  is taken over  $\mathcal{X} = \{\mathbf{x} = \|\mathbf{x} - \mathbf{x}_0\|_q \leq \epsilon^{FastLin}\}$  where  $\epsilon^{FastLin}$  is the solution obtained from FastLin. The adversarial bound is chosen as the minimum of  $\epsilon^{FastLin}$  and the right hand side of (113). For simplicity, we only consider the norm with  $p = 1$  and  $q = \infty$ .

As shown in (113), to compute the adversarial bound  $\epsilon$ , we need to compute the gradient  $\nabla o$ . Recall that Certify uses semidefinite relaxation to compute  $\|\nabla o\|_1$ . FastLip uses a different approach to estimate element-wise derivative  $\nabla_{x_j} o = \frac{\partial o}{\partial x_j}$  for  $j \in \{1, \dots, k_0\}$ , then

$$\max_{\mathbf{x} \in \mathcal{X}} \|\nabla_{\mathbf{x}} o\|_1 \leq \sum_j \max_{\mathbf{x} \in \mathcal{X}} |\nabla_{x_j} o|. \quad (114)$$

The computation of the gradient follows from the method discussed in section 4.3, which uses interval arithmetic and considers the chain rule. Given the bounds computed in FastLin, the activation pattern can be determined. Hence, the lower and upper bounds of the gradients of the activation functions can be determined. Then the bounds on gradients  $\underline{\mathbf{G}}_i$  and  $\overline{\mathbf{G}}_i$  can be computed following (22) and (23).<sup>32</sup> Given the bounds on the gradients, the gradient of  $o$  satisfies that

$$\underbrace{[\mathbf{c}]_+^\top \underline{\mathbf{G}}_n + [\mathbf{c}]_-^\top \overline{\mathbf{G}}_n}_{\mathbf{a}} \leq \nabla o \leq \underbrace{[\mathbf{c}]_+^\top \overline{\mathbf{G}}_n + [\mathbf{c}]_-^\top \underline{\mathbf{G}}_n}_{\mathbf{b}}. \quad (115)$$

<sup>32</sup> The original paper uses a different set of equations. The two derivations are indeed equivalent.

Then the maximum 1-norm gradient is

$$\max_{\mathbf{x} \in \mathcal{X}} \|\nabla_{\mathbf{x}} o\|_1 \leq \sum_j \max_{\mathbf{x} \in \mathcal{X}} |\nabla_{x_j} o| = \sum_j \max\{|a_j|, |b_j|\}. \quad (116)$$

Then

$$\epsilon^{FastLip} := \min \left\{ \frac{-o(\mathbf{x}_0)}{\sum_j \max\{|a_j|, |b_j|\}}, \epsilon^{FastLin} \right\}. \quad (117)$$

Our implementation is shown in algorithm 8.6.

```

struct FastLip
    maxIter::Int64
     $\epsilon_0$ ::Float64
    accuracy::Float64
end

function solve(solver::FastLip, problem::Problem)
    c, d = tosimplehrep(convert(HPolytope, problem.output))
    y = compute_output(problem.network, problem.input.center)
    o = (c * y - d)[1]
    if o > 0
        return AdversarialResult(:violated, -o)
    end
    result = solve(FastLin(solver), problem)
    result.status == :violated && return result
     $\epsilon_{\text{fastLin}}$  = result.max_disturbance
    LG, UG = get_gradient(problem.network, problem.input)
    a, b = interval_map(c, LG, UG)
    v = max.(abs.(a), abs.(b))
     $\epsilon$  = min(-o/sum(v),  $\epsilon_{\text{fastLin}}$ )
    if  $\epsilon$  > maximum(problem.input.radius)
        return AdversarialResult(:holds,  $\epsilon$ )
    else
        return AdversarialResult(:violated,  $\epsilon$ )
    end
end

```

Algorithm 8.6. FastLip. FastLip depends on FastLin and further estimates the allowable input range by computing a local Lipschitz constant of the network.

#### 8.4 DLV

DLV [18] searches not only the input space but also the hidden layers for a possible counter example. It uses a layer-by-layer approach. At layer  $i$ , it tries to find an assignment of the hidden nodes  $\mathbf{z}_i$  that satisfies the following two constraints:

1. Under the forward mapping  $\mathbf{f}_{i+1 \rightarrow n} := \mathbf{f}_n \circ \dots \circ \mathbf{f}_{i+1}$ , the output  $\mathbf{f}_{i+1 \rightarrow n}(\mathbf{z}_i)$  does not belong to the desired constraint  $\mathcal{Y}$ .
2. Under the backward mapping  $\mathbf{f}_{1 \rightarrow i} := \mathbf{f}_i \circ \dots \circ \mathbf{f}_1$ , there is a valid input  $\mathbf{x} \in \mathcal{X}$  such that  $\mathbf{f}_{1 \rightarrow i}(\mathbf{x}) = \mathbf{z}_i$ .

The conditions can be written concisely as

$$(\mathbf{f}_{1 \rightarrow i})^{-1}(\mathbf{z}_i) \cap \mathcal{X} \neq \emptyset, \quad (118a)$$

$$\mathbf{f}_{i+1 \rightarrow n}(\mathbf{z}_i) \notin \mathcal{Y}. \quad (118b)$$

*Overall procedure* To search for  $\mathbf{z}_i$  that satisfies (118), DLV does the following steps layer by layer.

1. Find a reachable set  $\mathcal{R}_i$  at layer  $i$  given the reachable set  $\mathcal{R}_{i-1}$  at layer  $i-1$ . The reachable set at the input layer is  $\mathcal{R}_0 := \mathcal{X}$ .
2. Build a search tree inside  $\mathcal{R}_i$ , denoted  $\mathcal{T}_i$ . The search tree  $\mathcal{T}_i$  should be a refinement of the search tree  $\mathcal{T}_{i-1}$  in the previous layer.<sup>33</sup>

<sup>33</sup> The branches in the search tree are called ladders in the original paper [18].

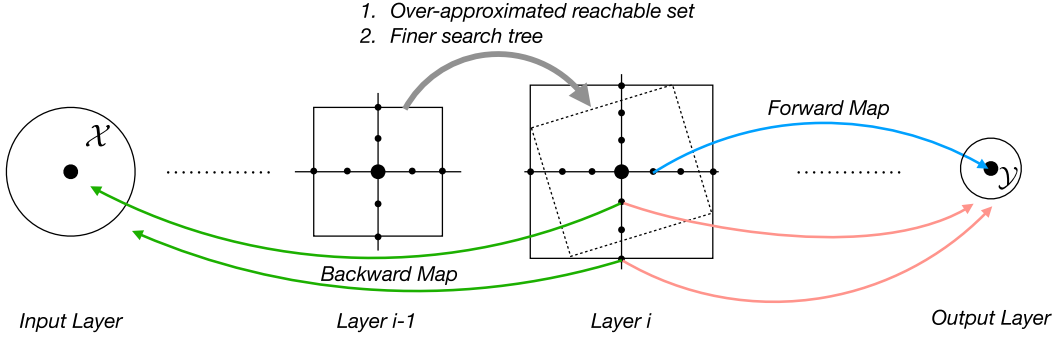


Figure 18. Illustration of the DLV approach. DLV uses layer-by-layer analysis. At layer  $i$ , it computes an over-approximated reachable set and a finer search tree based on the reachable set and the search tree in layer  $i - 1$ . For each point on the search tree, DLV first checks whether it violates the output constraint  $\mathcal{Y}$  using the forward map. For a point that violates the output constraint, DLV then checks whether it is reachable given the input constraint  $\mathcal{X}$  using the backward map. If such a point is reachable, then the corresponding input value is taken as a counter example.

3. Search for a counter example in the search tree  $\mathcal{T}_i$  that satisfies (118). If such an example is found, the property is violated. If not, we continue to the next layer.

The procedure is illustrated in figure 18. The key insight is that we can refine the search tree layer by layer. When it is closer to the output layer, it is easier to sample for counter examples. The original DLV implementation<sup>34</sup> uses SMT solvers to construct a reachable set in step 1 and build a search tree in step 2. For simplicity, our implementation in algorithm 8.7 does not involve SMT solvers. For step 1, we may compute the reachable set using any reachability method discussed in section 5. In algorithm 8.7, we use interval arithmetic by directly calling MaxSens using the function `get_bounds` in algorithm 5.5. The considerations in building the search tree in step 2 and the satisfiability check in step 3 are discussed below.

<sup>34</sup> <https://github.com/VeriDeep/DLV>

**Search tree** To build a search tree, we may explore all possible directions as shown in figure 19a. However, the complexity grows exponentially with  $k_i$ , the number of nodes in layer  $i$ . The authors of DLV suggest that we decompose the high-dimensional search space into several perpendicular low-dimensional search spaces, and only build search trees in those low-dimensional search spaces. Here, we simplify the search to element-wise exploration, *i.e.*, we sample every hidden node separately as shown in figure 19b. The construction of the search tree for a new layer needs to ensure that the tree is finer in the new layer. A set of conditions that precisely define “finer” are discussed in the DLV paper [18]. The problem of finding a search tree is then solved by calling an SMT solver.

For simplicity, our implementation takes equidistant samples for every hidden node. The sampling interval for node  $z_{i,j}$  is denoted  $\epsilon_{i,j}$ . The sample intervals are refined layer by layer. For the input node, the sampling intervals  $\epsilon_{0,j}$  for all  $j \in \{1, \dots, k_0\}$  are set to a solver-specified value. The sample interval at layer  $i - 1$  indeed defines a hyperrectangle spanned by the following vectors

$$\mathbf{v}_{i-1,1} := [\epsilon_{i-1,1}, 0, \dots, 0], \mathbf{v}_{i-1,2} := [0, \epsilon_{i-1,2}, 0, \dots, 0], \dots, \mathbf{v}_{i-1,k_{i-1}} := [0, \dots, 0, \epsilon_{i-1,k_{i-1}}]. \quad (119)$$

No point, with the exception of the vertices of the hyperrectangle, are evaluated at layer  $i - 1$ . In layer  $i$ , we need to ensure that we evaluate points inside the hyperrectangle, as shown in figure 18. The vertices of the hyperrectangle are mapped to  $\mathbf{W}_i \mathbf{v}_{i-1,j'}$  at layer  $i$  for all  $j' \in \{1, \dots, k_{i-1}\}$ . For the  $j$ th node in layer  $i$ , the projected sampling interval given the hyperrectangle at layer  $i - 1$  is  $\max_{j'} \sigma_{i,j}(|\mathbf{w}_{i,j} \mathbf{v}_{i-1,j'}|) = \max_{j'} \sigma_{i,j}(|w_{i,j,j'}| \epsilon_{i-1,j'})$ .<sup>35</sup> We then set the sample interval to

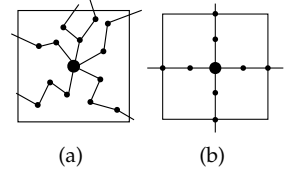


Figure 19. Illustration of a search tree in a 2D hidden space. (a) Complex search tree that explores all directions. (b) Simplified search tree for element-wise exploration.

<sup>35</sup>  $\sigma_{i,j}$  is the activation function for node  $j$  at layer  $i$ .  $\mathbf{w}_{i,j}$  is the  $j$ th row in the weight matrix  $\mathbf{W}_i$ .  $w_{i,j,k}$  is the  $k$ th entry in  $\mathbf{w}_{i,j}$ .

```

struct DLV
  optimizer
   $\epsilon$ ::Float64
end

function solve(solver::DLV, problem::Problem)
   $\eta$  = get_bounds(problem)
   $\delta$  = Vector{Vector{Float64}}(undef, length( $\eta$ ))
   $\delta[1]$  = fill(solver. $\epsilon$ , dim( $\eta[1]$ ))
  if issubset(last( $\eta$ ), problem.output)
    return CounterExampleResult(:holds)
  end
  output = compute_output(problem.network, problem.input.center)
  for (i, layer) in enumerate(problem.network.layers)
     $\delta[i+1]$  = get_manipulation(layer,  $\delta[i]$ ,  $\eta[i+1]$ )
    if i == length(problem.network.layers)
      mapping = x -> (x  $\in$  problem.output)
    else
      forward_nnet = Network(problem.network.layers[i+1:end])
      mapping = x -> (compute_output(forward_nnet, x)  $\in$  problem.output)
    end
    var, y = bounded_variation( $\eta[i+1]$ , mapping,  $\delta[i+1]$ )
    if var
      backward_nnet = Network(problem.network.layers[1:i])
      status, x = backward_map(y, backward_nnet,  $\eta[1:i+1]$ )
      if status
        return CounterExampleResult(:violated, x)
      end
    end
  end
  return ReachabilityResult(:violated, [last( $\eta$ )])
end

```

Algorithm 8.7. Main code for DLV. First, the reachable set  $\eta$  for each layer is computed by calling `get_bounds`. Then the sampling intervals for the input layer are initialized. The solver first checks for counter examples in the input layer by calling `bounded_variation`. If no counter example is found, the solver then proceeds to the hidden layers. For every hidden layer, the desired sampling intervals are computed by calling `get_manipulation`. Then the solver checks for hidden values that do not map to  $\mathcal{Y}$  by calling `bounded_variation`. For those hidden values, the solver then maps back to the input layer by calling `backward_map`. If a corresponding input is found, it returns the input as a counter example for the problem. Otherwise, it goes to the next layer. The details of the functions `get_manipulation` and `bounded_variation` are not included.



be a value smaller than the projected sampling interval:

$$\epsilon_{i,j} := \gamma \max_{j'} \sigma_{i,j}(|w_{i,j,j'}| \epsilon_{i-1,j'}), \quad (120)$$

where  $\gamma \in (0, 1)$  is set by the solver. This function is implemented in `get_manipulation`.

Given the sampling interval  $\epsilon_i$ , the search tree  $\mathcal{T}_i$  consists of  $2k_i$  chain branches. All branches are centered at  $\mathbf{f}_{1 \rightarrow i}(\mathbf{x}_0)$ . Then chains move along  $\mathbf{v}_{i,j}$  in either the positive direction or the negative direction,

$$\mathbf{f}_{1 \rightarrow i}(\mathbf{x}_0) \rightarrow \mathbf{f}_{1 \rightarrow i}(\mathbf{x}_0) + 1 \cdot \mathbf{v}_{i,j} \rightarrow \mathbf{f}_{1 \rightarrow i}(\mathbf{x}_0) + 2 \cdot \mathbf{v}_{i,j} \rightarrow \dots \quad (121a)$$

$$\mathbf{f}_{1 \rightarrow i}(\mathbf{x}_0) \rightarrow \mathbf{f}_{1 \rightarrow i}(\mathbf{x}_0) - 1 \cdot \mathbf{v}_{i,j} \rightarrow \mathbf{f}_{1 \rightarrow i}(\mathbf{x}_0) - 2 \cdot \mathbf{v}_{i,j} \rightarrow \dots \quad (121b)$$

Since  $j \in \{1, \dots, k_i\}$ , we then have  $2k_i$  chain branches. For example, in figure 19b, the search tree in two-dimensional space contains four branches. We don't need to explicitly construct the search tree. The search tree will be implicitly traversed when we check for satisfiability in step 3.

The original implementation of DLV is always complete and is sound when the search tree obtained by SMT solvers is minimal [18]. The tree nodes on a minimal search tree effectively cover all branches of a network. However, the search tree in our implementation is built heuristically and may not be minimal. To still make it sound, we only return holds when the output reachable set belongs to the output constraint. Otherwise, even if a counter example is not found, we return a violated reachability result. Hence, our implementation is sound but not complete.

*Satisfiability check* For every sampled value  $\mathbf{z}_i$  in the search tree, we first check if the corresponding output belongs to  $\mathcal{Y}$ . The function is implemented in `bounded_variation`, which outputs the points on the search tree that violate the output constraint. The original paper [18] indeed computes the maximum bounded variation of the search tree, *i.e.*, the maximum number of links that connect one satisfying node and one unsatisfying node along any tree path. As only one counter example is needed to falsify our problem, our implementation only checks for the existence of such a link.

If such a  $\mathbf{z}_i$  is found, we solve the following optimization problem to see if  $\mathbf{z}_i$  is reachable from  $\mathcal{X}$ :<sup>36</sup>

$$\min_{\mathbf{x}} \|\mathbf{x} - \mathbf{x}_0\|_{\infty}, \quad (122a)$$

$$\text{s.t. } \mathbf{x} \in \mathcal{X}, \mathbf{f}_{1 \rightarrow i}(\mathbf{x}) = \mathbf{z}_i. \quad (122b)$$

<sup>36</sup> The point  $\mathbf{z}_i$  may lie in the over-approximated area that is not reachable from the input set  $\mathcal{X}$ .

The optimization problem is solved using a MILP encoding in `backward_map`. The method is similar to NSVerify. If we find such an  $\mathbf{x}$ , then it is a counter example.

## 9 Search and Optimization

This section discusses methods that combine search with optimization approaches. These methods leverage the piecewise linearity in the activation functions to develop efficient algorithms. Sherlock [10] uses local and global search to compute bounds on the output, solving different optimization problems during the local search and the global search. BaB [7] uses branch and bound to compute bounds on the output, where the branch step corresponds to search and the bound step corresponds to optimization. Planet [13] formulates the verification problem as a

satisfiability problem, searching for an activation pattern such that a feasible input is mapped to an infeasible output. Reluplex [20] uses a simplex algorithm that searches for a feasible activation pattern that leads to an infeasible output.

### 9.1 Sherlock

Sherlock [10] estimates the output range for a network with a single output, *i.e.*,  $k_n = 1$ . If  $k_n > 1$ , the method works for each individual variable. Given the input domain  $\mathcal{X}$ , we compute the tightest output bounds

$$\ell_n = \min_{\mathbf{x} \in \mathcal{X}} f(\mathbf{x}), \quad u_n = \max_{\mathbf{x} \in \mathcal{X}} f(\mathbf{x}). \quad (123)$$

Sherlock solves (123) by combining local search and global search. In local search, it solves a linear program to find the optimal value in a given line segment of the function  $f$ . In global search, it solves a feasibility program to check whether the current local optimal bound can be improved. To find the global optimal bound, the solver iteratively does local search and global search. This approach is illustrated in figure 20. The main process is implemented in algorithm 9.1.

```

struct Sherlock
    optimizer
     $\epsilon :: \text{Float64}$ 
end

function solve(solver::Sherlock, problem::Problem)
    ( $x_u$ ,  $u$ ) = output_bound(solver, problem, :max)
    ( $x_l$ ,  $l$ ) = output_bound(solver, problem, :min)
    bound = Hyperrectangle(low = [ $l$ ], high = [ $u$ ])
    reach = Hyperrectangle(low = [ $l$  - solver. $\epsilon$ ], high = [ $u$  + solver. $\epsilon$ ])
    return interpret_result(reach, bound, problem.output,  $x_l$ ,  $x_u$ )
end

function output_bound(solver::Sherlock, problem::Problem, type::Symbol)
    opt = solver.optimizer
     $x$  = sample(problem.input)
    while true
        ( $x$ , bound) = local_search(problem,  $x$ , opt, type)
        bound_ $\epsilon$  = bound + ifelse(type == :max, solver. $\epsilon$ , -solver. $\epsilon$ )
        ( $x_{\text{new}}$ , bound_new, feasible) = global_search(problem, bound_ $\epsilon$ ,
                                                    opt, type)

        feasible || return ( $x$ , bound)
        ( $x$ , bound) = ( $x_{\text{new}}$ , bound_new)
    end
end

```

Algorithm 9.1. The main function in Sherlock. Sherlock computes both the upper and the lower bound of the output node through the function `output_bound` and then interprets the result with respect to the given output constraint. The function `output_bound` performs iterative local and global search in the while loop. The global search step needs to improve the bound by at least  $\epsilon$ . The while loop is broken if the global search fails. The value  $\epsilon$  is specified by the solver, which affects the tightness of the bound.

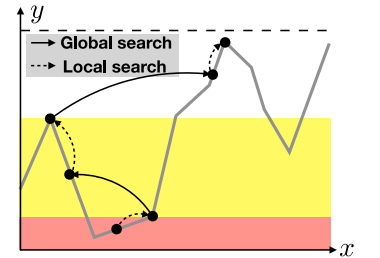


Figure 20. Illustration of the Sherlock approach. Sherlock combines local and global search. For a network with ReLU activations, the function represented by the network is piecewise linear. Local search improves the bound in a linear segment. Global search finds a point that improves the local bound by  $\epsilon$ , which lies in another linear segment.

*Local search* Starting from a sampled point  $\mathbf{x}^*$  in the input domain. Local search tries to find a better solution that has the same activation pattern as  $\mathbf{x}^*$ , *i.e.*, in the same line segment of  $f$ . Denote the activation pattern with respect to  $\mathbf{x}^*$  as  $\delta_{i,j}^*$  for all layers  $i$  and nodes  $j$ . The network

is encoded using (43). The local optimization problem for the upper bound is formulated as

$$\max_{\mathbf{z}_0, \dots, \mathbf{z}_n, \hat{\mathbf{z}}_1, \dots, \hat{\mathbf{z}}_n} \nabla f(\mathbf{x}^*)^\top \mathbf{z}_0, \quad (124a)$$

$$\text{s.t. } \mathbf{z}_0 \in \mathcal{X}, \quad (124b)$$

$$z_{i,j} = \hat{z}_{i,j} \geq 0, \text{ if } \delta_{i,j}^* = 1, \forall i \in \{1, \dots, n\}, j \in \{1, \dots, k_i\}, \quad (124c)$$

$$z_{i,j} = 0, \hat{z}_{i,j} \leq 0, \text{ if } \delta_{i,j}^* = 0, \forall i \in \{1, \dots, n\}, j \in \{1, \dots, k_i\}, \quad (124d)$$

$$\hat{\mathbf{z}}_i = \mathbf{W}_i \mathbf{z}_{i-1} + \mathbf{b}_i, \forall i \in \{1, \dots, n\}, \quad (124e)$$

where  $\nabla f(\mathbf{x}^*)$  is the gradient at  $\mathbf{x}^*$  computed by (18). The optimal solution is denoted as  $\mathbf{x}^l$ . The corresponding bound is  $u_n^l := f(\mathbf{x}^l)$ . Similarly we can compute the lower bound by changing the max to min. The local search is implemented in algorithm 9.2.

```
function local_search(problem::Problem, x, optimizer, type::Symbol)
    nnet = problem.network
    act_pattern = get_activation(nnet, x)
    gradient = get_gradient(nnet, x)
    model = Model(with_optimizer(optimizer))
    neurons = init_neurons(model, nnet)
    add_set_constraint!(model, problem.input, first(neurons))
    encode_network!(model, nnet, neurons, act_pattern, StandardLP())
    o = gradient * neurons[1]
    index = ifelse(type == :max, 1, -1)
    @objective(model, Max, index * o[1])
    optimize!(model)
    x_new = value(neurons[1])
    bound_new = compute_output(nnet, x_new)
    return (x_new, bound_new[1])
end
```

Algorithm 9.2. Local search in Sherlock, which solves equation (124). The network is encoded as a set of linear constraints according to the given activation pattern. Those constraints confine the function  $f$  to the line segment that  $\mathbf{x}^*$  lies on. The slope of the line segment equals the gradient of  $f$  at  $\mathbf{x}^*$ . Hence, the solution of the local search is a local optimum of the function  $f$ .

*Global search* Given the bound from local search, a global search aims to improve the bound by  $\epsilon$ , a solver-specified value. The global search problem is a feasibility problem. For example, for the upper bound, it tries to find a solution that satisfies

$$\mathbf{x} \in \mathcal{X}, f(\mathbf{x}) \geq u_n^l + \epsilon. \quad (125)$$

This problem is solved by calling NSVerify. If a solution  $\mathbf{x}^g$  is found, then the upper bound is updated to  $u_n^g = f(\mathbf{x}^g)$ . The global search is implemented in algorithm 9.3.

Since global search only solves a feasibility problem, the solution is not guaranteed to be a local optimum. Once a new global bound is found, local search will be called again to obtain the local optimum with respect to the new result. The reference point in the local search is set to  $\mathbf{x}^* := \mathbf{x}^g$ . After the local search, the global search will be called again. If there is no solution for the global search, we conclude that the upper bound is the bound obtained by the last local search, i.e.,  $u_n := u_n^l$ . Similarly, we can compute the global lower bound.

```

function global_search(problem::Problem, bound, optimizer, type::Symbol)
    index = ifelse(type == :max, 1.0, -1.0)
    h = HalfSpace([index], index * bound)
    output_set = HPolytope([h])
    problem_new = Problem(problem.network, problem.input, output_set)
    solver = NSVerify(optimizer)
    result = solve(solver, problem_new)
    if result.status == :violated
        x = result.counter_example
        bound = compute_output(problem.network, x)
        return (x, bound[1], true)
    else
        return ([], 0.0, false)
    end
end

```

Algorithm 9.3. Global search in Sherlock. It calls NSVerify to solve equation (125) to determine if the bound can be improved by  $\epsilon$  or not. If there is such a point that improves the bound by at least  $\epsilon$ , the point and the new bound are returned. If there does not exist such a point, the problem is determined infeasible by returning a false flag.

*Result interpretation* The original purpose of Sherlock is only to obtain tight bounds. To fit into our problem formulation (4), the obtained bounds are compared against the output constraint  $\mathcal{Y}$ . The bounding set  $\mathcal{B} := [\ell_n, u_n]$  is tight with at most  $\epsilon$  uncertainty. The reachable set is defined as  $\tilde{\mathcal{R}} := [\ell_n - \epsilon, u_n + \epsilon]$ . The solution of the problem can be determined by

$$\tilde{\mathcal{R}} \in \mathcal{Y} \Rightarrow \text{holds}, \quad (126a)$$

$$\mathcal{B} \notin \mathcal{Y} \Rightarrow \text{violated}, \quad (126b)$$

$$\text{otherwise} \Rightarrow \text{Undetermined}. \quad (126c)$$

The three conditions are illustrated in figure 21 and implemented in algorithm 9.4. When the reachable set belongs to the output constraint, the property holds. When either bound  $u_n$  or  $\ell_n$  exceeds the output constraint, the property is violated. In this case, we can output the counter example, which is the point that achieves either  $u_n$  or  $\ell_n$  that violates the constraint. Otherwise, the problem is undetermined. The smaller the  $\epsilon$ , the more accurate the result, as the problem is less likely to be undetermined. It is worth noting that the key advantage of Sherlock is to output tight bounds. It calls optimization solvers multiple times during the search for bounds. If the purpose is only to verify a given output constraint, it would be more computationally efficient to solve a feasibility problem using NSVerify, which corresponds to one global search step in Sherlock.

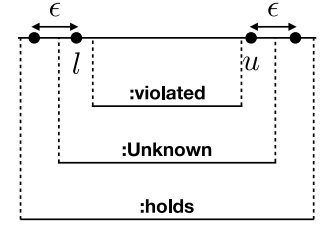


Figure 21. Interpreting results. The upper line illustrates the computed bound  $[l, u]$  and the potential reachable set  $[l - \epsilon, u + \epsilon]$ . The bottom three lines illustrate three different output constraints  $\mathcal{Y}$ . If  $\mathcal{Y}$  does not cover either  $\ell$  or  $u$ , the solver returns violated. If  $\mathcal{Y}$  covers the reachable set, the solver returns holds. Otherwise, the solver returns unknown.

```

function interpret_result(reach, bound, output, x_l, x_u)
    if high(reach) > high(output) && low(reach) < low(output)
        return ReachabilityResult(:holds, reach)
    end
    high(bound) > high(output) && return CounterExampleResult(:violated, x_u)
    low(bound) < low(output) && return CounterExampleResult(:violated, x_l)
    return ReachabilityResult(:Unknown, reach)
end

```

Algorithm 9.4. Interpretation of results. The code implements equation (126) and outputs corresponding counter examples when there is a violation. The input `reach` corresponds to  $[\ell_n - \epsilon, u_n + \epsilon]$ . The input `bound` corresponds to  $[\ell_n, u_n]$ . The input `output` is the output constraint  $\mathcal{Y}$ . The last two inputs are the points that achieve  $\ell_n$  and  $u_n$  respectively.

## 9.2 BaB

BaB [7] uses branch and bound to estimate the output bounds of a network.<sup>37</sup> The main loop is shown in algorithm 9.5. For simplicity, we assume that the network only has one output node. Similar to Sherlock, the solver tries to compute the lower and upper bounds of the output in (123). The bounds are computed using the function `output_bound`. The results are interpreted by `interpret_result` in algorithm 9.4. BaB is different from Sherlock mainly in the method to estimate bounds. In the following discussion, we discuss the approach to estimate the upper bound  $u_n$ . Estimation of  $\ell_n$  follows easily.

We maintain a global upper bound  $\bar{u}_n$  of  $u_n$ , and a global lower bound  $\underline{u}_n$  of  $u_n$ . The lower bound  $\underline{u}_n$  is computed by sampling for a concrete input, hence it is also called the concrete bound. The upper bound  $\bar{u}_n$  is computed with respect to a relaxation of the problem, hence it is called the approximated bound. The concrete bound always provides an under-estimation, while the upper bound always provides an over-estimation. For the lower bound  $\ell_n$ , its lower bound  $\underline{\ell}_n$  is the approximated bound, while its upper bound  $\bar{\ell}_n$  is the concrete bound. These two possibilities are encoded by the `:max` and `:min` symbols in algorithm 9.5.

If  $\bar{u}_n - \underline{u}_n < \epsilon$  for some small  $\epsilon$  specified by the solver, the over-estimation is small. Then we conclude  $u_n$  is found. To minimize the over-approximation, the bounds  $\bar{u}_n$  and  $\underline{u}_n$  are improved by iterative interval refinement (a search process) similar to the procedure discussed in ReluVal in section 8.1.

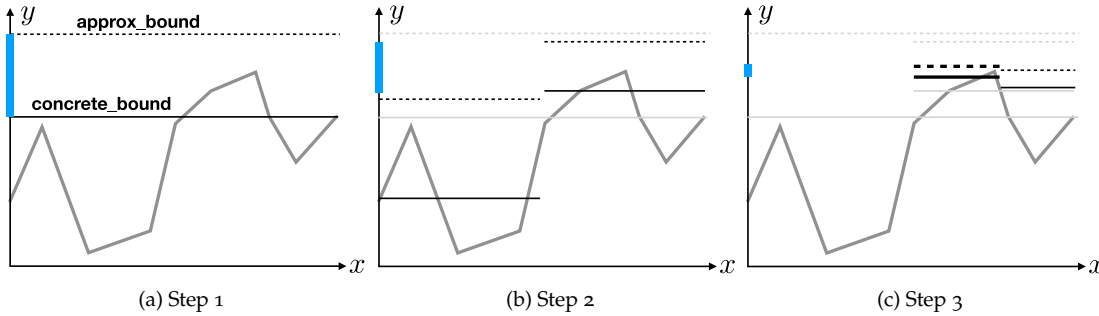


Figure 22. Illustration of branch and bound to estimate the upper bound of  $y$ . In step 1, an approximated upper bound (dashed line) and a concrete lower bound (solid line) are computed for the whole input interval. As there is a large gap between the two bounds (blue range), the input interval is split in two. In step 2, approximated upper bounds and concrete lower bounds are computed for the two intervals respectively. As the upper bound of the left interval is smaller than the lower bound of the right interval, the left interval is pruned out. The new global bounds are updated to be the bounds of the right interval. In step 3, the right interval is further split into two intervals. The previous process is repeated. The new bounds are close to each other. Hence, the global upper bound of  $y$  is found.

**Search strategy** During the search process, a list of input subsets  $\{\mathcal{X}_i\}_i$  are recorded. In addition, the approximated bound of that subset  $\mathcal{X}_i$  is recorded, which is used to prioritize the list. For the estimation of  $u_n$ , we record  $\bar{u}_n(\mathcal{X}_i)$ . The list satisfies that  $\bar{u}_n(\mathcal{X}_1) \geq \bar{u}_n(\mathcal{X}_2) \geq \dots$ . For the estimation of  $\ell_n$ , we record  $\underline{\ell}_n(\mathcal{X}_i)$ . The list satisfies that  $\underline{\ell}_n(\mathcal{X}_1) \leq \underline{\ell}_n(\mathcal{X}_2) \leq \dots$ .

At each iteration, the first domain in the priority queue, *i.e.*, the domain with the largest upper bound  $\bar{u}_n(\mathcal{X}_1)$  or the domain with the smallest lower bound  $\underline{\ell}_n(\mathcal{X}_1)$ , is picked out and split into two domains. In ReluVal, we split the index that corresponds to the greatest smear value. In BaB, we split the index that has the longest interval. The resulting two domains are denoted  $\mathcal{X}_1^j$  for  $j \in \{1, 2\}$ .

For the two domains, we compute their approximated and concrete bounds  $\bar{u}_n(\mathcal{X}_1^j)$  and  $\underline{u}_n(\mathcal{X}_1^j)$  for  $j \in \{1, 2\}$ . The global lower bound  $\underline{u}_n(\mathcal{X})$  is updated to  $\max_j \underline{u}_n(\mathcal{X}_1^j)$  if it is greater than the previous global lower bound. If the upper bound  $\bar{u}_n(\mathcal{X}_1^j)$  is greater than the global

lower bound, the corresponding sub domain  $\mathcal{X}_1^j$  needs further splits. Then  $\mathcal{X}_1^j$  is pushed back to the priority list. The current global upper bound  $\bar{u}_n(\mathcal{X})$  is  $\bar{u}_n(\mathcal{X}_1)$  in the new list.

The search terminates if the global concrete and approximated bounds are close to each other, i.e.,  $\bar{u}_n(\mathcal{X}) - \underline{u}_n(\mathcal{X}) < \epsilon$ . The search process is illustrated in figure 22.

```

struct BaB
    optimizer
     $\epsilon :: \text{Float64}$ 
end

function solve(solver::BaB, problem::Problem)
    (u_approx, u, x_u) = output_bound(solver, problem, :max)
    (l_approx, l, x_l) = output_bound(solver, problem, :min)
    bound = Hyperrectangle(low = [l], high = [u])
    reach = Hyperrectangle(low = [l_approx], high = [u_approx])
    return interpret_result(reach, bound, problem.output, x_l, x_u)
end

function output_bound(solver::BaB, problem::Problem, type::Symbol)
    nnet = problem.network
    global_concrete, x_star = concrete_bound(nnet, problem.input, type)
    global_approx = approx_bound(nnet, problem.input, solver.optimizer, type)
    doms = [(global_approx, problem.input)]
    index = ifelse(type == :max, 1, -1)
    while index * (global_approx - global_concrete) > solver. $\epsilon$ 
        dom = pick_out(doms)
        subdoms = split_dom(dom[2])
        for i in 1:length(subdoms)
            dom_concrete, x = concrete_bound(nnet, subdoms[i], type)
            dom_approx = approx_bound(nnet, subdoms[i], solver.optimizer, type)
            if index * (dom_concrete - global_concrete) > 0
                (global_concrete, x_star) = (dom_concrete, x)
            end
            if index * (dom_approx - global_concrete) > 0
                add_domain!(doms, (dom_approx, subdoms[i]), type)
            end
        end
        global_approx = doms[1][1]
    end
    return (global_approx, global_concrete, x_star)
end

```

Algorithm 9.5. The main function in BaB. BaB computes both the upper and the lower bound of the output node through the function `output_bound` and then interprets the result with respect to the given output constraint. The function `output_bound` performs branch and bound in the while loop. The solver keeps track of a concrete under-estimation and an approximated over-estimation in the search process. The search terminates once the two bounds are close to each other. The value  $\epsilon$  is specified by the solver, which affects the tightness of the bound.

*Concrete bounds* The concrete bounds  $\underline{u}_n(\mathcal{X}_i)$  and  $\bar{\ell}_n(\mathcal{X}_i)$  are computed by sampling in the domain  $\mathcal{X}_i$ .<sup>38</sup> Heuristically, we only check three points in a domain, the upper and lower bounds of the domain, and the center point. Hence, an upper bound is always accompanied with a concrete sample input that achieves that bound. Computation of concrete bounds is implemented in the function `concrete_bound` in algorithm 9.6.

*Approximated bounds* The approximated bounds  $\bar{u}_n(\mathcal{X}_i)$  and  $\underline{\ell}_n(\mathcal{X}_i)$  are computed by minimizing the output given the  $\Delta$ -relaxation of the network. For example,  $\bar{u}_n(\mathcal{X}_i)$  is computed by

<sup>38</sup> ReluVal uses a similar approach to search for violations in `check_inclusion`.

$$\max_{z_0, \dots, z_n, \hat{z}_1, \dots, \hat{z}_n} z_n, \quad (127a)$$

$$\text{s.t. } z_{i,j} = \hat{z}_{i,j}, \forall i \in \{1, \dots, n-1\}, j \in \Gamma_i^+, \quad (127b)$$

$$z_{i,j} = 0, \forall i \in \{1, \dots, n-1\}, j \in \Gamma_i^-, \quad (127c)$$

$$z_{i,j} \geq \hat{z}_{i,j}, z_{i,j} \geq 0, z_{i,j} \leq \frac{\hat{u}_{i,j}(\hat{z}_{i,j} - \hat{\ell}_{i,j})}{\hat{u}_{i,j} - \hat{\ell}_{i,j}}, \forall i \in \{1, \dots, n-1\}, j \in \Gamma_i, \quad (127d)$$

$$\hat{z}_i = \mathbf{W}_i \mathbf{z}_{i-1} + \mathbf{b}_i, \forall i \in \{1, \dots, n-1\}, \quad (127e)$$

$$\mathbf{z}_0 \in \mathcal{X}_i. \quad (127f)$$

The approximated bound  $\ell_n(\mathcal{X}_i)$  can be computed by changing the maximization to minimization in (127). ConvDual solves a similar optimization problem in (73), though it transforms it to the dual problem and then obtains a heuristic solution. Due to relaxation, the above optimization overestimates the true bounds, *i.e.*,  $\bar{u}_n(\mathcal{X}_i) \geq u_n(\mathcal{X}_i)$  and  $\ell_n(\mathcal{X}_i) \leq \ell_n(\mathcal{X}_i)$ . The finer  $\mathcal{X}_i$ , the closer the approximated bound is to the true value. Hence, the approximated bound is improved during iterative refinement. Computation of the approximated bounds is implemented in the function `approx_bound` in algorithm 9.6.

```

function concrete_bound(nnet::Network, subdom::Hyperrectangle, type::Symbol)
    points = [subdom.center, low(subdom), high(subdom)]
    values = Vector{Float64}(undef, 0)
    for p in points
        push!(values, sum(compute_output(nnet, p)))
    end
    value, index = ifelse(type == :min, findmin(values), findmax(values))
    return (value, points[index])
end

function approx_bound(nnet::Network, dom::Hyperrectangle, optimizer, type)
    bounds = get_bounds(nnet, dom)
    model = Model(with_optimizer(optimizer))
    neurons = init_neurons(model, nnet)
    add_set_constraint!(model, dom, first(neurons))
    encode_network!(model, nnet, neurons, bounds, TriangularRelaxedLP())
    index = ifelse(type == :max, 1, -1)
    o = sum(last(neurons))
    @objective(model, Max, index * o)
    optimize!(model)
    termination_status(model) == OPTIMAL && return value(o)
end

```

Algorithm 9.6. Compute bounds in BaB. The concrete bound is computed by sampling in the domain. Heuristically, we sample three points: lower bound of domain, upper bound of domain, and center of domain. The approximated bound is computed by minimizing the output constrained on  $\Delta$ -relaxation of the network.

*Result interpretation* Similar to Sherlock, the original purpose of BaB is only to obtain tight bounds. To fit into our problem formulation (4), the obtained bounds are compared against the output constraint  $\mathcal{Y}$ . Define the bounding set  $\mathcal{B} := [\bar{\ell}_n, \bar{u}_n]$ , which considers only the concrete bounds, and are tight with at most  $\epsilon$  uncertainty. The reachable set is defined as  $\tilde{\mathcal{R}} := [\ell_n, \bar{u}_n]$ , which considers only the approximated bounds. Then the solution to the problem can be determined following the conditions in (126).



### 9.3 Planet

Planet [13] formulates the problem (4) as a satisfiability problem, which consists of trying to find an assignment for  $\delta_i$  for all  $i$  such that a point in the input set can be mapped to the complement of the output set. The search in Planet can be understood as a binary tree search as the assignment is either 0 or 1 as illustrated in figure 23. The novelty of Planet is that it uses linear programming to 1) infer tighter bounds on the nodes, 2) filter out conflicting assignments, and 3) infer more assignments given a partial assignment of  $\delta_i$ . In this way, the search can be more efficient.

The original implementation<sup>39</sup> is eager, which modifies the main loop in a state-of-the-art SAT solver (*i.e.*, MiniSAT) to impose network constraints during the search phase with partial assignments for  $\delta_i$ . If conflicts are detected in the partial assignments, the solver performs back-tracking.

For simplicity, we didn't follow the original eager implementation. Instead, we only check for conflicts when all  $\delta$  have been assigned. We call PicoSAT.jl<sup>40</sup> to solve the SAT problem for assignment. In this way, we can leave the SAT solver untouched. The main loop in algorithm 9.7 is different from the Planet paper. In the non-eager implementation, the node-wise bounds are computed first. Planet introduces an optimization approach to compute tighter bounds, which will be discussed below. Then a clause  $\Psi$  is initialized with respect to the bounds. The clause can be understood as a pruned search tree. A full assignment of all  $\delta$ 's according to the clause is computed by calling PicoSAT. The feasibility of this assignment is checked by elastic filtering, which returns conflicts in the assignment if the assignment is not feasible. Then the conflicts are added to the clause, which helps further pruning the search tree. If the assignment is feasible, that means there is an activation pattern that maps a desired input to an undesired output. Then the property is violated. If no such assignment exists, the property holds.

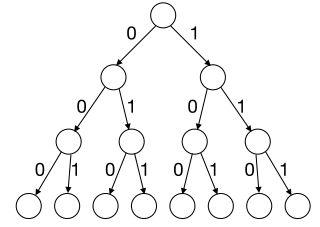


Figure 23. Illustration of binary tree search to assign values to  $\delta_{i,j}$  for all nodes  $j$  in layer  $i$ .

<sup>39</sup> <https://github.com/progirep/planet>

<sup>40</sup> <https://github.com/jakebolewski/PicoSAT.jl>

```

struct Planet
    optimizer
    eager::Bool
end

function solve(solver::Planet, problem::Problem)
    status, bounds = tighten_bounds(problem, solver.optimizer)
    status == OPTIMAL || return CounterExampleResult(:holds)
     $\psi$  = init_ $\psi$ (problem.network, bounds)
     $\delta$  = PicoSAT.solve( $\psi$ )
    opt = solver.optimizer
    while  $\delta$  != :unsatisfiable
        status, conflict = elastic_filtering(problem,  $\delta$ , bounds, opt)
        status == INFEASIBLE || return CounterExampleResult(:violated, conflict)
        push!( $\psi$ , conflict)
         $\delta$  = PicoSAT.solve( $\psi$ )
    end
    return CounterExampleResult(:holds)
end

```

Algorithm 9.7. Main loop in Planet. This is a non-eager implementation. First, tight bounds of node values are computed through `tighten_bounds`. Then the clause  $\Psi$  is initialized with respect to the bounds. A full assignment according to the clause is computed by calling `PicoSAT`. The feasibility of this assignment is checked by `elastic_filtering`, which returns conflicts in the assignment if the assignment is not feasible. Then the conflicts are added to the clause. The while loop repeats previous steps. If the assignment is feasible, that means there is an activation pattern that maps a desired input to an undesired output. Then the property is violated. If no such assignment exists, the property is violated.

*Computing tight bounds* We first compute the bounds through `tighten_bounds` in algorithm 9.8. This function first gets relatively loose bounds  $\ell_i$  and  $\mathbf{u}_i$  from `get_bounds` in algorithm 5.5. Then



the following optimization problem is solved

$$\min_{\mathbf{z}_0, \dots, \mathbf{z}_n, \hat{\mathbf{z}}_1, \dots, \hat{\mathbf{z}}_n} q \sum_{i=1}^n \sum_{j=1}^{k_i} z_{i,j}, \quad (128a)$$

$$\text{s.t. } \mathbf{z}_0 \in \mathcal{X}, \mathbf{z}_n \notin \mathcal{Y}, \quad (128b)$$

$$z_{i,j} = \hat{z}_{i,j}, \forall i \in \{1, \dots, n-1\}, j \in \Gamma_i^+, \quad (128c)$$

$$z_{i,j} = 0, \forall i \in \{1, \dots, n-1\}, j \in \Gamma_i^-, \quad (128d)$$

$$z_{i,j} \geq \hat{z}_{i,j}, z_{i,j} \geq 0, z_{i,j} \leq \frac{\hat{u}_{i,j}(\hat{z}_{i,j} - \hat{\ell}_{i,j})}{\hat{u}_{i,j} - \hat{\ell}_{i,j}}, \forall i \in \{1, \dots, n-1\}, j \in \Gamma_i, \quad (128e)$$

$$\hat{\mathbf{z}}_i = \mathbf{W}_i \mathbf{z}_{i-1} + \mathbf{b}_i, \forall i \in \{1, \dots, n\}, \quad (128f)$$

where  $q \in \{-1, 1\}$ . When  $q = -1$ , we compute the tighter upper bounds. When  $q = 1$ , we get the lower bounds. Here we use  $\Delta$ -relaxation in (46). If there is no solution in (128), that means the constraints are infeasible. Hence, for all  $\mathbf{x} \in \mathcal{X}$ ,  $\mathbf{y} = \mathbf{f}(\mathbf{x}) \in \mathcal{Y}$ . Then the property is satisfied.

```
function tighten_bounds(problem::Problem, optimizer)
    bounds = get_bounds(problem)
    network = problem.network
    model = Model(with_optimizer(optimizer))
    neurons = init_neurons(model, network)
    add_set_constraint!(model, problem.input, first(neurons))
    add_complementary_set_constraint!(model, problem.output, last(neurons))
    encode_network!(model, network, neurons, bounds, TriangularRelaxedLP())

    min_sum!(model, neurons)
    optimize!(model)
    termination_status(model) == OPTIMAL || return (INFEASIBLE, bounds)
    lower = value.(neurons)

    max_sum!(model, neurons)
    optimize!(model)
    termination_status(model) == OPTIMAL || return (INFEASIBLE, bounds)
    upper = value.(neurons)

    new_bounds = Vector{Hyperrectangle}(undef, length(neurons))
    for i in 1:length(neurons)
        new_bounds[i] = Hyperrectangle(low = lower[i], high = upper[i])
    end
    return (OPTIMAL, new_bounds)
end
```

Algorithm 9.8. Obtaining tighter bounds using optimization. The solution is tighter than `get_bounds` in algorithm 5.5. The implementation encodes equation (128) and solves the optimization problem by calling the JuMP solver.

*Solving the SAT problem* A clause  $\Psi$  corresponds to a search tree, which encodes a set of binary conditions. According to the bounds computed earlier, the clause can be initialized as

$$\Psi = \bigwedge_{\hat{\ell}_{i,j} \leq 0 \leq \hat{u}_{i,j}} [\{\delta_{i,j} = 1\} \vee \{\delta_{i,j} = 0\}] \bigwedge_{0 < \hat{\ell}_{i,j}} [\delta_{i,j} = 1] \bigwedge_{\hat{u}_{i,j} < 0} [\delta_{i,j} = 0]. \quad (129)$$

Solving the SAT problem means finding an assignment of  $\delta$ 's such that the clause  $\Psi$  is true. The case that  $\Psi = \bigwedge_{i,j} [\{\delta_{i,j} = 1\} \vee \{\delta_{i,j} = 0\}]$  corresponds to a full search tree shown in figure 23. As more conditions are added to the clause, we are essentially pruning the search tree. Elastic filtering, to be discussed below is used to help the pruning. There may be more than one solution that satisfies the clause  $\Psi$ . We only consider one solution at a time. PicoSAT is called to find a feasible assignment for  $\delta$ 's.

*Elastic filtering* Given an assignment of  $\delta_{i,j}$ 's, we check for conflicts using `elastic_filtering` in algorithm 9.9. The problem is defined as

$$\min_{\mathbf{z}_0, \dots, \mathbf{z}_n, \hat{\mathbf{z}}_1, \dots, \hat{\mathbf{z}}_n, \mathbf{s}_1, \dots, \mathbf{s}_n} q \sum_{i=1}^n \sum_{j=1}^{k_i} s_{i,j}, \quad (130a)$$

$$\text{s.t. } \mathbf{z}_0 \in \mathcal{X}, \mathbf{z}_n \notin \mathcal{Y}, \quad (130b)$$

$$z_{i,j} = \hat{z}_{i,j}, \forall i \in \{1, \dots, n-1\}, j \in \Gamma_i^+, \quad (130c)$$

$$z_{i,j} = 0, \forall i \in \{1, \dots, n-1\}, j \in \Gamma_i^-, \quad (130d)$$

$$z_{i,j} \geq \hat{z}_{i,j}, z_{i,j} \geq 0, z_{i,j} \leq \frac{\hat{u}_{i,j}(\hat{z}_{i,j} - \hat{\ell}_{i,j})}{\hat{u}_{i,j} - \hat{\ell}_{i,j}}, \forall i \in \{1, \dots, n-1\}, j \in \Gamma_i, \quad (130e)$$

$$\hat{\mathbf{z}}_i = \mathbf{W}_i \mathbf{z}_{i-1} + \mathbf{b}_i, \forall i \in \{1, \dots, n\}, \quad (130f)$$

$$z_{i,j} = \hat{z}_{i,j} + s_{i,j} \geq 0, \text{ for } \delta_{i,j} = 1, \forall i \in \{1, \dots, n\}, j \in \{1, \dots, k_i\}, \quad (130g)$$

$$z_{i,j} = 0, \hat{z}_{i,j} - s_{i,j} \leq 0, \text{ for } \delta_{i,j} = 0, \forall i \in \{1, \dots, n\}, j \in \{1, \dots, k_i\}, \quad (130h)$$

which uses both  $\Delta$ -relaxation (given bounds) and slack relaxation (given activation). If the assignment of  $\delta_{i,j}$ 's are feasible, then all the slack variables should be non-positive. If some slack variables are positive, we fix the node with the largest slack value by adding a condition  $s_{i,j} = 0$ . Then the optimization problem is solved again and again until there is no feasible solution. The list of node  $\{(i^{(1)}, j^{(1)}), (i^{(2)}, j^{(2)}), \dots, (i^{(k)}, j^{(k)})\}$  and their  $\delta$  assignments that we fixed during the process is a conflict. A conflict means that the  $\delta$  assignments of those nodes cannot be satisfied simultaneously. The intuition behind elastic filtering is that we can find conflicts faster by fixing the most violated constraints.

Once a conflict is determined with respect to an assignment, we can add the conflict back to the clauses and solve the SAT problem again. If there is no feasible solution of  $\Psi$ , the property holds. If no conflict is found for a specific assignment, the property is violated. Our implementation is still sound and complete, though it is less efficient than the original eager implementation. Our implementation requires that input set  $\mathcal{X}$  be a `Hyperrectangle` and the output set  $\mathcal{Y}$  be a `PolytopeComplement`.

#### 9.4 Reluplex

Reluplex [20] applies a simplex algorithm to ReLU networks. It searches for a counter example (10).

For each node  $(i, j)$ , Reluplex optimizes over two variables  $z_{i,j}$  and  $\hat{z}_{i,j}$ . There are three possible statuses for each node  $(i, j)$ , active (denoted  $(i, j) \in \mathcal{A}$ ), inactive (denoted  $(i, j) \in \mathcal{N}$ ), and undetermined (denoted  $(i, j) \in \mathcal{U}$ ). Reluplex tries to find a feasible assignment for undetermined nodes through depth-first search. The basic constraints in Reluplex without considering the

```

function elastic_filtering(problem::Problem,  $\delta$ , bounds, optimizer)
    network = problem.network
    model = Model(with_optimizer(optimizer))
    neurons = init_neurons(model, network)
    add_set_constraint!(model, problem.input, first(neurons))
    add_complementary_set_constraint!(model, problem.output, last(neurons))
    encode_network!(model, network, neurons, bounds, TriangularRelaxedLP())
    SLP = encode_network!(model, network, neurons,  $\delta$ , SlackLP())
    min_sum!(model, SLP.slack)
    conflict = Vector{Int64}()
    act = get_activation(network, bounds)
    while true
        optimize!(model)
        termination_status(model) == OPTIMAL || return (INFEASIBLE, conflict)
        (m, index) = max_slack(value.(SLP.slack), act)
        m > 0.0 || return (:Feasible, value.(neurons[1]))
        coeff =  $\delta$ [index[1]][index[2]] ? -1 : 1
        node = coeff * get_node_id(network, index)
        push!(conflict, node)
        @constraint(model, SLP.slack[index[1]][index[2]] == 0.0)
    end
end

```

Algorithm 9.9. Elastic filtering is used to determine conflicts in an assignment of all  $\delta$ 's. It iteratively solves equation (130). At each iteration, it fixes the largest slack variable to zero. The conflicting sequence is found once the optimization becomes infeasible.

undetermined nodes are

$$\mathcal{B} = \{\mathbf{z}_i, \hat{\mathbf{z}}_i : \mathbf{z}_0 \in \mathcal{X}, \mathbf{z}_n \notin \mathcal{Y}, \quad (131a)$$

$$z_{i,j} \geq 0, z_{i,j} \geq \hat{z}_{i,j}, \hat{\ell}_{i,j} \leq \hat{z}_{i,j} \leq \hat{u}_{i,j}, \quad (131b)$$

$$z_{i,j} = \hat{z}_{i,j}, \hat{z}_{i,j} \geq 0, \forall (i, j) \in \mathcal{A}, \quad (131c)$$

$$z_{i,j} = 0, \hat{z}_{i,j} < 0, \forall (i, j) \in \mathcal{N}, \quad (131d)$$

$$\hat{\mathbf{z}}_i = \mathbf{W}_i \mathbf{z}_{i-1} + \mathbf{b}_i, \forall i \in \{1, \dots, n\}, \quad (131e)$$

At each search step, Reluplex either finds a counter example, or determines that there is no such counter example (then does backtracking), or assigns one node  $(i^*, j^*) \in \mathcal{U}$  to be active or inactive. The set  $\mathcal{B}^0 := \mathcal{B}$ ,  $\mathcal{A}^0 := \mathcal{A}$ ,  $\mathcal{N}^0 := \mathcal{N}$ , and  $\mathcal{U}^0 := \mathcal{U}$  are initialized at the beginning of the search. At search depth  $k$ , we have a set of constraints  $\mathcal{B}^k$ , which is an intersection of the basic constraint  $\mathcal{B}$  and the constraint induced by the assignments of nodes in  $\mathcal{U}^0$  during the search.  $\mathcal{B}^k$  can be represented in the form of (131) by adding superscript  $k$  to  $\mathcal{B}$ ,  $\mathcal{A}$ , and  $\mathcal{N}$ . The relationships among the sets at different depths are

$$\mathcal{A}^0 \subseteq \mathcal{A}^1 \subseteq \dots \subseteq \mathcal{A}^k, \quad (132a)$$

$$\mathcal{N}^0 \subseteq \mathcal{N}^1 \subseteq \dots \subseteq \mathcal{N}^k, \quad (132b)$$

$$\mathcal{U}^0 \supset \mathcal{U}^1 \supset \dots \supset \mathcal{U}^k. \quad (132c)$$

At each search step  $k$ , we first solve a feasibility problem to check if there is a solution of  $\mathcal{B}^{k-1}$ . If there is no feasible solution that satisfies  $\mathcal{B}^{k-1}$ , we do back tracking. If there exists a solution that satisfies  $\mathcal{B}^{k-1}$ , we consider the following two possibilities.

- If all ReLU constraints are satisfied, *i.e.*,  $z_{i,j} = \max\{\hat{z}_{i,j}, 0\}$ , then we indeed find a counter example for the problem (4).

- If some ReLU constraints are not satisfied, *i.e.*,  $z_{i,j} \neq \max\{\hat{z}_{i,j}, 0\}$ , then we need to fix those broken nodes during the search. We pick such a broken node  $(i^k, j^k)$  from  $\mathcal{U}^{k-1}$ . Then  $\mathcal{U}^k = \mathcal{U}^{k-1} \setminus \{(i^k, j^k)\}$ . We may assign the node to be either active or inactive. In the active case,  $\mathcal{A}^k = \mathcal{A}^{k-1} \cup \{(i^k, j^k)\}$ ,  $\mathcal{N}^k = \mathcal{N}^{k-1}$ , and

$$\mathcal{B}^k := \mathcal{B}^{k-1} \cap \{\mathbf{z}_i, \hat{\mathbf{z}}_i : z_{i^k, j^k} = \hat{z}_{i^k, j^k} \geq 0\}. \quad (133)$$

In the inactive case,  $\mathcal{A}^k = \mathcal{A}^{k-1}$ ,  $\mathcal{N}^k = \mathcal{N}^{k-1} \cup \{(i^k, j^k)\}$ , and

$$\mathcal{B}^k := \mathcal{B}^{k-1} \cap \{\mathbf{z}_i, \hat{\mathbf{z}}_i : z_{i^k, j^k} = 0, \hat{z}_{i^k, j^k} \leq 0\}. \quad (134)$$

During the search, we either end up finding a counter example or concluding that there is no such counter example. The worst case scenario complexity is  $2^{|\mathcal{U}|}$ , *i.e.*, we traverse a depth  $|\mathcal{U}|$  binary tree, where  $|\mathcal{U}|$  computes the cardinality of the set  $\mathcal{U}$ . It is possible to use elastic filtering introduced in Planet to help detect conflicts faster.

Our implementation supports input sets  $\mathcal{X}$  of type `Hyperrectangle` and output sets  $\mathcal{Y}$  of type `PolytopeComplement` and is shown in algorithm 9.10.<sup>41</sup> The depth-first search is performed in the function `reluplexStep`. The construction of  $\mathcal{B}^k$  at each search step is done by `encode` in algorithm 9.11. The inputs to `reluplexStep` include,

- `model`, which encodes the constraint  $\mathcal{B}^k$  similar to (131).
- `b_vars` and `f_vars`, which are  $\hat{\mathbf{z}}$ 's and  $\mathbf{z}$ 's.
- `relu_status`, which can be either 0, 1 or 2. If it is 0, the corresponding node belongs  $\mathcal{U}^k$ . If it is 1, the corresponding node belongs  $\mathcal{A}^k$ . If it is 2, the corresponding node belongs  $\mathcal{N}^k$ .

<sup>41</sup> The original Reluplex uses a data structure called Tableau to encode new LP constraints. Conceptually, our Julia implementation is the same as the original implementation, but in a less efficient way.

## 10 Comparison and Results

This section presents experimental results for our implementation of the algorithms.<sup>42</sup> Different algorithms handle problems with different specifications as shown in table 1. As described previously, the objects used to represent the input set  $\mathcal{X}$  and the output set  $\mathcal{Y}$  vary along with the characteristics of the sets that each of the algorithms support. Additionally, different algorithms output different types of results. We have, consequently, split the algorithms in six groups such that the same verification problem can be solved by all the algorithms within a group, facilitating a comparison of the implementations provided with this work. The groups are as follows:

1. Ai2, ExactReach, and maxSens - Input: `HPolytope`, Output: `HPolytope` (bounded).
2. ILP, MIPVerify, and NSVerify - Input: `Hyperrectangle`, Output: `PolytopeComplement`.
3. Duality and convDual - Input: `Hyperrectangle` (uniform radius), Output: `Halfspace`.
4. FastLin, FastLip, ILP, and MIPVerify - Input: `Hyperrectangle`, Output: `Halfspace`.
5. BaB, DLV, ReluVal, and Sherlock - Input: `Hyperrectangle`, Output: `Hyperrectangle` (1-D).
6. Planet, Reluplex, and ReluVal - Input: `Hyperrectangle`, Output: `PolytopeComplement` and `Hyperrectangle`.

Despite having different problem specifications, the exact same property can be encoded across Groups 2, 3, 4, and 6 by using their corresponding input and output sets to represent the same constraints.

<sup>42</sup> In our implementations, readability was favored over speed. Some of our implementations are simplified versions of the original ones that still output the same results but can be slower.

```

struct Reluplex
  optimizer
end

function solve(solver::Reluplex, problem::Problem)
  initial_model = Model(solver)
  bs, fs = encode(solver, initial_model, problem)
  layers = problem.network.layers
  initial_status = [zeros(Int, n) for n in n_nodes.(layers)]
  insert!(initial_status, 1, zeros(Int, dim(problem.input)))

  return reluplex_step(solver, problem, initial_model, bs, fs, initial_status)
end

function reluplex_step(solver::Reluplex,
  problem::Problem,
  model::Model,
   $\hat{z}$ ::Vector{Vector{VariableRef}},
  z::Vector{Vector{VariableRef}},
  relu_status::Vector{Vector{Int}})
  optimize!(model)
  if termination_status(model) == OPTIMAL
    i, j = find_relu_to_fix( $\hat{z}$ , z)
    i == 0 && return CounterExampleResult(:violated, value.(first( $\hat{z}$ )))
    for repair_type in 1:2
      relu_status[i][j] = repair_type
      new_m = Model(solver)
      bs, fs = encode(solver, new_m, problem)
      enforce_repairs!(new_m, bs, fs, relu_status)
      result = reluplex_step(solver, problem, new_m, bs, fs, relu_status)
      relu_status[i][j] = 0
      result.status == :violated && return result
    end
  end
  return CounterExampleResult(:holds)
end

function enforce_repairs!(model::Model,  $\hat{z}$ , z, relu_status)
  for i in 1:length(relu_status), j in 1:length(relu_status[i])
     $\hat{z}_{ij}$  =  $\hat{z}$ [i][j]
     $z_{ij}$  = z[i][j]
    if relu_status[i][j] == 1
      type_one_repair!(model,  $\hat{z}_{ij}$ ,  $z_{ij}$ )
    elseif relu_status[i][j] == 2
      type_two_repair!(model,  $\hat{z}_{ij}$ ,  $z_{ij}$ )
    end
  end
end

function type_one_repair!(model,  $\hat{z}_{ij}$ ,  $z_{ij}$ )
  @constraint(model,  $\hat{z}_{ij}$  ==  $z_{ij}$ )
  @constraint(model,  $\hat{z}_{ij}$  >= 0.0)
end

function type_two_repair!(model,  $\hat{z}_{ij}$ ,  $z_{ij}$ )
  @constraint(model,  $\hat{z}_{ij}$  <= 0.0)
  @constraint(model,  $z_{ij}$  == 0.0)
end

```

Algorithm 9.10. Main loop in Reluplex. The function `reluplexStep` performs depth-first search. At every search step, it first solves a feasibility problem encoded in `model`. If the problem is infeasible, meaning that no counter example can be found, we output `:holds`. If there is a solution of the problem, we find the first node such that the ReLU activation is broken, *i.e.*,  $z_{i,j} \neq [\hat{z}_{i,j}]_+$ . If no such broken node is found, meaning that we have found a counter example, we output `:violated` with the counter example. For the broken node, we can either change its status to 1 or 2, which corresponds to the two branches in the search. For each branch, a new constrained problem is formulated. New constraints are added through `enforce_repairs!`, which encode constraints according to `relu_status`. With the new model, the search continues into the next depth.

```

function encode(solver::Reluplex, model::Model, problem::Problem)
    layers = problem.network.layers
     $\hat{z}$  = init_neurons(model, layers)
    z = init_neurons(model, layers)
    activation_constraint!(model,  $\hat{z}$ [1], z[1], Id())
    bounds = get_bounds(problem)
    for (i, L) in enumerate(layers)
        @constraint(model, affine_map(L, z[i]) .==  $\hat{z}$ [i+1])
        add_set_constraint!(model, bounds[i],  $\hat{z}$ [i])
        activation_constraint!(model,  $\hat{z}$ [i+1], z[i+1], L.activation)
    end
    add_complementary_set_constraint!(model, problem.output, last(z))
    feasibility_problem!(model)
    return  $\hat{z}$ , z
end

```

Algorithm 9.11. Encoding the optimization problem in Reluplex. The optimization problem has zero objective and is constrained on  $\mathcal{B}$  in equation (131). In the search process, more constraints are to be added by `enforce_repairs!`, which encode constraints according to `relu_status`.

### 10.1 Experiments

We used different neural networks to benchmark the performance of the algorithms in different scenarios. We focused on three contexts: networks of varying sizes trained to classify hand-written digits from the MNIST dataset [21], the Aircraft Collision Avoidance System (ACAS) network [20].<sup>43</sup> and we created a tiny toy network (small nnet) for which we analytically derived its transfer function.

*Small nnet* We manually specified a toy network and analytically derived the one dimensional function that it represents. We evaluated simple properties corresponding to upper and lower bounds of the image of this function for a small interval in the input set. The network has two hidden layers of two units each.

*MNIST* For the MNIST networks, we constructed properties that encode regions centered around a point corresponding to a hand-written digit in the original dataset. We verified a property associated to the correct classification of the image. The property requires that the logits (outputs of the final layer) do not vary significantly from those of the original image when the image is perturbed within a region in the input space. This corresponds to an  $\ell_1$  ball centered around the point that corresponds to a sample hand-written image of the number 1 digit. The output set  $\mathcal{V}$  is an  $\ell_1$  ball built around the original output of the network corresponding to that input. For algorithms that only support halfspace or polytopeComplement output sets, we encoded the region where the logit of the digit 0 is less than the logit of the digit 1. The networks have the following characteristics:

- mnist1. Input size: 748, 1 hidden layer of size: 25, output size: 10 (0.9009 test accuracy).
- mnist2. Input size: 748, 1 hidden layer of size: 100, output size: 10 (0.9463 test accuracy).
- mnist3. Input size: 748, 4 hidden layers of size: 25, output size: 10 (0.9555 test accuracy).
- mnist4. Input size: 748, 6 hidden layers of size: 50, output size: 10 (0.9664 test accuracy).

<sup>43</sup> This network is based on a neural network trained on a very early prototype of ACAS Xu, targeted for unmanned aircraft. Details can be found in the article by Julian, Kochenderfer, and Owen [19].

**ACAS** For the ACAS network, we verified property 10 introduced by Katz et al. [20]. Property 10 corresponds to the situation where the intruder aircraft is far away from the ownship and the desired output is that the advisory is clear-of-conflict. To make the property faster to verify, we reduced the volume of the input region by fixing the last three inputs of the network to specific values instead of the ranges originally defined for the property. This property has been verified in prior work [20], [40]. For the algorithms that can only support halfspace or PolytopeComplement output sets, we encoded the region where the first output is less than the last output. The network has five input units, six hidden layers of 50 units each, and five output units.

The experiments in Group 5 required networks with a single output node. We used pruned versions of the exact same networks. We encoded the corresponding properties in the dimension of the preserved output node. For the MNIST networks this is the output corresponding to digit 1 and for the ACAS network this is the output corresponding to the cost of advising clear-of-conflict.

## 10.2 Results

In the following table, we summarize the results of evaluating the aforementioned properties with the different algorithms. For each experiment, we recorded the time that it took for each algorithm to terminate. Timed-out threshold is set to be 24 hours. We also recorded the result of each algorithm to identify cases in which incomplete algorithms failed to correctly identify properties that hold.

Group 1 supports hyperrectangle input sets. Groups 2, 3, 4, and 6 support H-polytope input sets. Group 5 supports hyperrectangle input sets and networks with only one output node.

Algorithm	small_nnet	mnist1	mnist2	mnist3	mnist4	acas
ExactReach	0.004070968	-	-	-	-	-
Ai2	0.005170172	-	-	-	-	-
maxSens	0.000201878	16.06772067	16.20866383	16.1892307	15.93491465	0.006839497

Algorithm	small_nnet	mnist1	mnist2	mnist3	mnist4	acas
NSVerify	0.000437805	0.561558217	1.23185184	0.234906589	-	-
MIPVerify	0.422803839	0.236930468	1.055669556	46.36556919	-	-
ILP	0.423599637	0.374052375	0.65545287	0.951015935*	4.866014534*	0.108586473*
convDual	0.001725546	0.374052375	0.02184014	0.001638679	0.005836024	0.002674586
Duality	0.001336414*	21.60260384*	130.7149674*	37.3058392*	52.87126159*	0.020148564*
FastLin	2.088460281	0.477470278	0.002369427	0.007551539	0.01564011	0.166508046**
FastLip	9.179938173	0.130989072	0.059900531	0.051348471**	0.091895366**	0.001703324**
MIPVerify	9.940695891	0.256747563	1.23054911	46.77128226	-	-
ILP	0.347950783	0.367925285	0.665364297	0.890090777*	4.664959506*	0.00834114*
Planet	0.000247003	0.211006714	0.301637916	0.186482861	-	-
Reluplex	0.012245108	125.0347088	62582.48883	-	-	2952.784643
Reluval	0.000031296	0.646829741	0.008979679	0.008030365	0.824760803*	0.196084517*

Table 2. Experimental results for group 1. All results are in seconds, missing entries correspond to experiments that timed-out. An asterisk indicates the result was :unknown and two asterisks indicate that the algorithm incorrectly returned :violated.

Table 3. Experimental results for groups 2, 3, 4 and 6. All results are in seconds, missing entries correspond to experiments that timed-out. An asterisk indicates the result was :unknown and two asterisks indicate that the algorithm incorrectly returned :violated.



Algorithm	small_nnet	mnist1	mnist2	mnist3	mnist4	acas
Reluval	0.096123516	0.104757137	0.082225486	0.019977288	0.021589774*	0.000375971*
DLV	0.145483368	0.055010885	0.7606918**	0.203945542**	3.426374089**	0.450508522**
Sherlock	0.150546186	-	-	-	-	-
BaB	0.160190662	-	-	-	-	-

Table 4. Experimental results for group 5. All results are in seconds, missing entries correspond to experiments that timed-out. An asterisk indicates the result was :unknown and two asterisks indicate that the algorithm incorrectly returned :violated.

### 10.3 Analysis

The experimental results shown in the previous section demonstrate the capability of the pedagogical implementation to verify realistic networks. Many of the algorithms are able to verify properties for networks as large as the ACAS Xu networks, which has been used for prior benchmarks [20], [40]. Overall we observed that algorithms that are complete take a longer time to run. Consequently, complete algorithms are more amenable, at least at this point and this implementation, to verifying properties of smaller networks. Algorithms that are not complete usually rely on over-approximations or other schemes that significantly reduce their computational cost. Incomplete algorithms are faster and can more easily be used to verify properties on larger networks.

In particular, for Group 1, we can observe in table 2 that ExactReach and Ai2 timed out for most of the properties. Currently, our implementations of this algorithms require the conversion of sets from H-representation to V-representation which is computationally expensive in general. For example, converting a 700-dimensional polytope is an unreasonable task to perform. The original implementations of these algorithms are better equipped to handle larger networks. In particular, Ai2 was specifically designed to avoid this task by using zonotopes, in the near future we will update our implementation to incorporate this better approach.

For Groups 2, 3, 4 and 6, we can observe in table 3 that NSVerify, MIPVerify, Planet and Reluplex timed out for some of the properties associated to larger networks, this result is not surprising as completeness comes at a high computational cost. It is worth noting that 1) many of the algorithms that are not complete were able to terminate in significantly shorter amounts of time compared to their complete counterparts, but 2) many of them (the ones marked with two asterisks) exhibited their in-completeness by returning :violated for properties that, in fact, hold. Other algorithms, particularly in Group 3, were unable to reach a conclusion and returned :unknown as a result.

Algorithms in Group 5, as shown in table 4, either terminated very quickly but produced either incorrect or inconclusive results (Reluval and DLV respectively) or timed out for most networks such as Sherlock and BaB. Sherlock and BaB try to compute the exact bounds on the output even if the bounds already lie outside of the output constraint, which can explain why they timed-out for most networks as other approaches only check if a counter example can be found, hence should stop much earlier than Sherlock and BaB converge to precise-enough bounds.

Finally, algorithms that rely on optimization are susceptible to numerical stability issues and we observed this in the case of MIPVerify and Planet.

## 11 Conclusion

This article surveyed algorithms for verification of deep neural networks. A unified mathematical framework was introduced to verify satisfiability of a neural network given certain



input and output constraints. Three basic verification methods were identified: reachability, optimization, and search. We classified existing methods into five induced categories according to their core methodologies, and pointed out the connections among them. In particular, we reviewed the following methods: 1) reachability methods: ExactReach, Ai2, and MaxSens; 2) primal optimization methods: NSVerify, MIPVerify, and ILP; 3) dual optimization methods: Duality, ConvDual, and Certify; 4) search and reachability methods: ReluVal, FastLin, FastLip, and DLV; and 5) search and optimization methods: Sherlock, BaB, Planet, and Reluplex. In the numerical experiments, we compare methods that either use similar methodologies or can solve the same problems. In general, there is a trade-off between completeness of a verification algorithm and its scalability. Complete algorithms run slower on larger networks, while incomplete algorithms are more conservative. Pedagogical implementations of all these methods were provided in Julia. The connections and differences among different methods were pointed out. This article can serve as a tutorial for students and professionals interested in this emerging field as well as a benchmark to facilitate the design of new verification algorithms.

### Acknowledgments

This work is partially supported by the Center for Automotive Research at Stanford (CARS). The authors would like to thank many of the authors of the referenced papers for their help in clarifying their algorithms and reviewing early drafts of this survey: Weiming Xiang, Taylor Johnson, Hoang-Dung Tran, Martin Vechev, Gagandeep Singh, Alessio Lomuscio, Michael Akintunde, Osbert Bastani, Zico Kolter, Shiqi Wang, Huan Zhang, Xiaowei Huang, Rudy Bunel, Reudiger Ehlers, and Guy Katz. The authors would also like to thank Christian Schilling and Marcelo Forets, the authors of LazySets.jl, for their implementation support, as well as Amelia Hardy and Zongzhang Zhang for their comments.

### References

1. M. E. Akintunde, A. Lomuscio, L. Maganti, and E. Pirovano, "Reachability Analysis for Neural Agent-Environment Systems," in *International Conference on Principles of Knowledge Representation and Reasoning*, 2018.
2. M. E. Akintunde, A. Kevorchian, A. Lomuscio, and E. Pirovano, "Verification of RNN-Based Neural Agent-Environment Systems," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2019.
3. C. Barrett and C. Tinelli, "Satisfiability Modulo Theories," in *Handbook of Model Checking*, Springer, 2018, pp. 305–343.
4. O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi, "Measuring Neural Net Robustness with Constraints," in *Advances in Neural Information Processing Systems (NIPS)*, 2016.
5. J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017.
6. S. Bogomolov, M. Forets, G. Frehse, K. Potomkin, and C. Schilling, "JuliaReach: a Toolbox for Set-Based Reachability," *ArXiv*, no. 1901.10736, 2019.
7. R. Bunel, I. Turkaslan, P. H. Torr, P. Kohli, and M. P. Kumar, "A Unified View of Piecewise Linear Neural Network Verification," *ArXiv*, no. 1711.00455, 2017.
8. C.-H. Cheng, G. Nührenberg, and H. Ruess, "Verification of Binarized Neural Networks," *ArXiv*, no. 1710.03107, 2017.

9. C.-H. Cheng, G. Nührenberg, C.-H. Huang, and H. Ruess, "Verification of Binarized Neural Networks via Inter-Neuron Factoring," in *Verified Software. Theories, Tools, and Experiments*, 2018.
10. S. Dutta, S. Jha, S. Sanakaranarayanan, and A. Tiwari, "Output Range Analysis for Deep Neural Networks," *ArXiv*, no. 1709.09130, 2017.
11. S. Dutta, S. Jha, S. Sanakaranarayanan, and A. Tiwari, "Learning and Verification of Feedback Control Systems Using Feedforward Neural Networks," in *IFAC Conference on Analysis and Design of Hybrid Systems (ADHS)*, 2018.
12. K. Dvijotham, R. Stanforth, S. Gowal, T. Mann, and P. Kohli, "A Dual Approach to Scalable Verification of Deep Networks," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2018.
13. R. Ehlers, "Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks," in *International Symposium on Automated Technology for Verification and Analysis*, 2017.
14. T. Gehr, M. Mirman, D. Drashler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Aiz: Safety and Robustness Certification of Neural Networks with Abstract Interpretation," in *IEEE Symposium on Security and Privacy (SP)*, 2018.
15. I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep Learning*. MIT Press, 2016.
16. K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierison, "A Practical Tutorial on Modified Condition/decision Coverage," 2001.
17. K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
18. X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety Verification of Deep Neural Networks," in *International Conference on Computer Aided Verification*, 2017.
19. K. Julian, M. J. Kochenderfer, and M. P. Owen, "Deep Neural Network Compression for Aircraft Collision Avoidance Systems," *AIAA Journal of Guidance, Control, and Dynamics*, vol. 42, no. 3, pp. 598–608, 2019.
20. 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*, 2017.
21. Y. LeCun and C. Cortes, "MNIST Handwritten Digit Database," 2010.
22. F. Leofante, N. Narodytska, L. Pulina, and A. Tacchella, "Automated Verification of Neural Networks: Advances, Challenges and Perspectives," *ArXiv*, no. 1805.09938, 2018.
23. A. Lomuscio and L. Maganti, "An Approach to Reachability Analysis for Feed-Forward Relu Neural Networks," *ArXiv*, no. 1706.07351, 2017.
24. C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky, "The Stanford CoreNLP Natural Language Processing Toolkit," in *Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 2014.
25. M. Mirman, T. Gehr, and M. Vechev, "Differentiable Abstract Interpretation for Provably Robust Neural Networks," in *International Conference on Machine Learning (ICML)*, 2018.
26. V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., "Human-Level Control Through Deep Reinforcement Learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.
27. N. Narodytska, S. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh, "Verifying Properties of Binarized Deep Neural Networks," 2018.
28. J. D. Olden and D. A. Jackson, "Illuminating the "Black Box": a Randomization Approach for Understanding Variable Contributions in Artificial Neural Networks," *Ecological Modelling*, vol. 154, no. 1-2, pp. 135–150, 2002.
29. N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The Limitations of Deep Learning in Adversarial Settings," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016.

30. K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated Whitebox Testing of Deep Learning Systems," in *Symposium on Operating Systems Principles*, 2017.
31. A. Raghunathan, J. Steinhardt, and P. Liang, "Certified Defenses against Adversarial Examples," in *International Conference on Learning Representations*, 2018.
32. H. Salman, G. Yang, H. Zhang, C.-J. Hsieh, and P. Zhang, "A Convex Relaxation Barrier to Tight Robust Verification of Neural Networks," *ArXiv*, no. 1902.08722, 2019.
33. G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev, "Fast and Effective Robustness Certification," in *Advances in Neural Information Processing Systems (NIPS)*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds., Curran Associates, 2018, pp. 10 825–10 836.
34. G. Singh, T. Gehr, M. Puschel, and M. Vechev, "An Abstract Domain for Certifying Neural Networks," in *ACM Symposium on Principles of Programming Languages*, 2019.
35. G. Singh, T. Gehr, M. Puschel, and M. Vechev, "Boosting Robustness Certification of Neural Networks," in *International Conference on Learning Representations*, 2019.
36. Y. Sun, X. Huang, and D. Kroening, "Testing Deep Neural Networks," *ArXiv*, no. 1803.04792, 2018.
37. Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars," in *International Conference on Software Engineering*, 2018.
38. V. Tjeng, K. Xiao, and R. Tedrake, "Evaluating Robustness of Neural Networks with Mixed Integer Programming," *ArXiv*, no. 1711.07356, 2017.
39. S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Efficient Formal Safety Analysis of Neural Networks," *ArXiv*, no. 1809.08098, 2018.
40. 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)*, 2018.
41. L. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, L. Daniel, D. Boning, and I. Dhillon, "Towards Fast Computation of Certified Robustness for ReLU Networks," in *International Conference on Machine Learning (ICML)*, vol. 80, 2018.
42. T.-W. Weng, H. Zhang, P.-Y. Chen, J. Yi, D. Su, Y. Gao, C.-J. Hsieh, and L. Daniel, "Evaluating the Robustness of Neural Networks: An Extreme Value Theory Approach," in *International Conference on Learning Representations*, 2018.
43. E. Wong and Z. Kolter, "Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope," in *International Conference on Machine Learning (ICML)*, 2018.
44. W. Xiang, H. Tran, and T. T. Johnson, "Output Reachable Set Estimation and Verification for Multi-layer Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 11, pp. 5777–5783, 2018.
45. W. Xiang, H. Tran, J. A. Rosenfeld, and T. T. Johnson, "Reachable Set Estimation and Safety Verification for Piecewise Linear Systems with Neural Network Controllers," in *American Control Conference (ACC)*, 2018.
46. W. Xiang, H.-D. Tran, and T. T. Johnson, "Reachable Set Computation and Safety Verification for Neural Networks with ReLU Activations," *ArXiv*, no. 1712.08163, 2017.
47. W. Xiang, H.-D. Tran, and T. T. Johnson, "Specification-Guided Safety Verification for Feedforward Neural Networks," *ArXiv*, no. 1812.06161, 2018.
48. W. Xiang, P. Musau, A. A. Wild, D. M. Lopez, N. Hamilton, X. Yang, J. Rosenfeld, and T. T. Johnson, "Verification for Machine Learning, Autonomy, and Neural Networks Survey," *ArXiv*, no. 1810.01989, 2018.
49. P. Yang, J. Liu, J. Li, L. Chen, and X. Huang, "Analyzing Deep Neural Networks with Symbolic Propagation: Towards Higher Precision and Faster Verification," *ArXiv Preprint ArXiv:1902.09866*, 2019.

- 50. H. Zhang, P. Zhang, and C.-J. Hsieh, "RecurJac: An Efficient Recursive Algorithm for Bounding Jacobian Matrix of Neural Networks and Its Applications," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2019.
- 51. 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 (NIPS)*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds., Curran Associates, 2018, pp. 4944–4953.
- 52. Y. Zhang and Z. Zhang, "Dual Neural Network," in *Repetitive Motion Planning and Control of Redundant Robot Manipulators*. Springer, 2013, pp. 33–56.