LinSyn: Synthesizing Tight Linear Bounds for Arbitrary Neural Network Activation Functions*

Brandon Paulsen✉ and Chao Wang

University of Southern California, Los Angeles CA 90089, USA
{bpaulsen,wang626}@usc.edu

Abstract. The most scalable approaches to certifying neural network robustness depend on computing sound linear lower and upper bounds for the network's activation functions. Current approaches are limited in that the linear bounds must be handcrafted by an expert, and can be sub-optimal, especially when the network's architecture composes operations using, for example, multiplication such as in LSTMs and the recently popular *Swish* activation. The dependence on an expert prevents the application of robustness certification to developments in the state-of-the-art of activation functions, and furthermore the lack of tightness guarantees may give a false sense of insecurity about a particular model. To the best of our knowledge, we are the first to consider the problem of *automatically* synthesizing *tight* linear bounds for arbitrary n-dimensional activation functions. We propose the first fully automated method that achieves tight linear bounds while only leveraging the mathematical definition of the activation function itself. Our method leverages an efficient heuristic technique to synthesize bounds that are tight and *usually sound*, and then verifies the soundness (and adjusts the bounds if necessary) using the highly optimized branch-and-bound SMT solver, dReal. Even though our method depends on an SMT solver, we show that the runtime is reasonable in practice, and, compared with state of the art, our method often achieves 2-5X tighter final output bounds and more than quadruple certified robustness.

1 Introduction

Prior work has shown that neural networks are vulnerable to various types of (adversarial) perturbations, such as small l-norm bounded perturbations [39], geometric transformations [13, 22], and word substitutions [2]. Such perturbations can often cause a misclassification for any given input, which may have serious consequences, especially in safety critical systems. Certifying robustness to these perturbations has become an important problem as it can show the network does not exhibit these misclassifications, and furthermore previous work has shown that a given input feature's certified robustness can be a useful indicator to determine the feature's importance in the network's decision [34, 25].

* This work was partially funded by the U.S. National Science Foundation grants CNS-1813117 and CNS-1722710, and the U.S. Office of Naval Research (ONR) grant N00014-17-1-2896.

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13243, pp. 357–376, 2022.
https://doi.org/10.1007/978-3-030-99524-9_19

Indeed, many approaches have been proposed for certifying the robustness of inputs to these perturbations. Previous work typically leverages two types of techniques: (1) fast and scalable, but approximate techniques [36, 15, 45, 34, 25], and (2) expensive but exact techniques that leverage some type of constraint solver [23, 24, 40]. Several works have also combined the two [37, 35, 43, 42]. The most successful approaches, in terms of scalability in practice, are built on top of the approximate techniques, which all depend on computing *linear bounds* for the non-linear activation functions.

However, a key limitation is that the linear bounds must be handcrafted and proven sound by experts. Not only is this process difficult, but also ensuring the tightness of the crafted bounds presents an additional challenge. Unfortunately, prior work has only crafted bounds for the most common activation functions and architectures, namely ReLU [43], sigmoid, tanh [36, 48, 46], the exp function [34], and some 2-dimensional activations found in LSTM networks [25]. As a result, existing tools for neural network verification cannot handle a large number of activation functions that are frequently used in practice. Examples include the *GELU* function [18], which is currently the activation function used in OpenAI's GPT [31], and the *Swish* function which has been shown to outperform the standard ReLU function in some applications [32] and, in particular, can reduce over-fitting in adversarial training [38]. In addition, these recently introduced activation functions are often significantly more complex than previous activation functions, e.g., we have $gelu(x) = 0.5x(1 + \tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$.

In this work, we study the problem of *efficiently* and *automatically* synthesizing *sound* and *tight* linear bounds for any *arbitrary activation function*. By *arbitrary activation function*, we mean *any* (non-linear) computable function $z = \sigma(x_1, \ldots, x_d)$ used inside a neural network with d input variables. By *sound* we mean, given an interval bound on each variable $x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \ldots, x_d \in [l_d, u_d]$, the problem is to *efficiently* compute lower bound coefficients $c_1^l, c_2^l, \ldots, c_{d+1}^l$, and upper bound coefficients $c_1^u, c_2^u, \ldots, c_{d+1}^u$ such that the following holds:

$$\begin{gathered} \forall x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \ldots, x_d \in [l_d, u_d] \\ c_1^l x_1 + c_2^l x_2 + \cdots + c_{d+1}^l \leq \sigma(x_1, \ldots, x_d) \leq c_1^u x_1 + c_2^u x_2 + \cdots + c_{d+1}^u \end{gathered} \tag{1}$$

By *automatically*, we mean that the above is done using only the definition of the activation function itself. Finally, by *tight*, we mean that some formal measure, such as the volume above/below the linear bound, is minimized/maximized.

We have developed a new method, named LinSyn, that can *automatically* synthesize tight linear bounds for *any arbitrary* non-linear activation function $\sigma(\cdot)$. We illustrate the flow of our method on the left-hand side of Fig. 1. As shown, LinSyn takes two inputs: a definition of the activation function, and an interval for each of its inputs. LinSyn outputs linear coefficients such that Equation 1 holds. Internally, LinSyn uses sampling and an LP (linear programming) solver to synthesize candidate lower and upper bound coefficients. Next, it uses an efficient local minimizer to compute a good estimate of the offset needed to ensure soundness of the linear bounds. Since the candidate bounding functions constructed in this manner may still be unsound, finally, we use a highly optimized branch-and-bound nonlinear SMT solver, named dReal [14], to verify

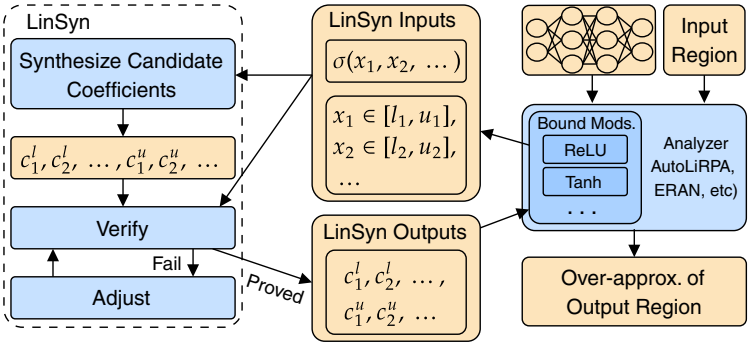

Fig. 1. The overall flow of LinSyn.

the soundness of the linear bounds. Even though our new method involves the use of solvers and optimizers, the entire process typically takes less than 1/100th of a second per pair of bounds.

Fig. 1 also illustrates how LinSyn fits in with existing neural network verification frameworks, such as ERAN [1], and AutoLiRPA [47]. These tools take as input a neural network, and a region of the neural networks input space, and compute an over-approximation of the neural network's outputs. Internally, these frameworks have modules that compute linear bounds for a specific activation functions. LinSyn is a one-size-fits-all drop-in replacement for these modules that are invoked at runtime whenever a linear bound of a non-linear activation function is needed.

Our method differs from these existing frameworks because a user (usually an expert in neural network verification) must provide hand-crafted, sound linear bounds for the activation functions of a neural network. However, to date, they only support the previously mentioned activation functions. We note however that the recent framework AutoLiRPA supports binary operations (namely addition, subtraction, multiplication, and division) as "activation functions". Thus, while it's not explicitly designed to handle complex activations, it has the ability to by decomposing, e.g., $gelu(x)$ into operations that it supports, and then combining them. In contrast, LinSyn bounds the activation function *as a whole*, which we will show produces much tighter linear bounds.

We have implemented our method in tool called LinSyn, and evaluated it on benchmarks in computer vision and natural language processing (NLP). Our evaluation shows that we can obtain final output bounds often 2-5X tighter than the most general tool [47], thus allowing us to drastically increase certified robustness. In addition, our tool achieves accuracy equal to or better than the handcrafted LSTM bounds of Popqorn [25], which is currently the most accurate tool for analyzing LSTM-based NLP models, at a comparable runtime.

To summarize, this paper makes the following contributions:

– We propose the first method for automatically synthesizing tight linear bounds for arbitrary activation functions.

– We implement our approach in a tool called LinSyn, and integrate it as a bounding module into the AutoLiRPA framework, thus producing a neural network verification tool that can theoretically compute tight linear bounds for any arbitrary activation function.
– We extensively evaluate our approach and show it outperforms state-of-the-art tools in terms of accuracy and certified robustness by a large margin.

The rest of this paper is organized as follows. First, we provide the technical background in Section 2. Then, we present our method for synthesizing the linear bounds in Section 3 and our method for verifying the linear bounds in Section 4. Next, we present the experimental results in Section 5. We review the related work in Section 6 and, finally, give our conclusions in Section 7.

2 Preliminaries

In this section, we define the neural network verification problem, and illustrate both how state-of-the-art verification techniques work, and their limitations.

2.1 Neural Networks

Following conventional notation, we refer to matrices with capital bold letters (e.g. $\mathbf{W} \in \mathbb{R}^{n \times m}$), vectors as lower case bold letters (e.g. $\mathbf{x} \in \mathbb{R}^n$), and scalars or variables with lower case letters (e.g. $x \in \mathbb{R}$). Slightly deviating from the convention, we refer to a set of elements with capital letters (e.g. $X \subseteq \mathbb{R}^n$).

We consider two types of networks in our work: feed-forward and recurrent. We consider a feed-forward neural network to be a (highly) non-linear function $f : \mathbb{X} \to \mathbb{Y}$, where $\mathbb{X} \subseteq \mathbb{R}^n$ and $\mathbb{Y} \subseteq \mathbb{R}^m$. We focus on neural network *classifiers*. For an input $\mathbf{x} \in \mathbb{X}$, each element in the output $f(\mathbf{x})$ represents a score for a particular class, and the class associated with the largest element is the chosen class. For example, in image classification, $\mathbb{X}$ would be the set of all images, each element of an input $\mathbf{x} \in \mathbb{X}$ represents a pixel's value, and each element in $\mathbb{Y}$ is associated with a particular object that the image might contain.

In feed-forward neural networks the output $f(\mathbf{x})$ is computed by performing a series of affine transformations, i.e., multiplying by a weight matrix, followed by application of an activation function $\sigma(\cdot)$. Formally, a neural network with l layers has l two-dimensional weight matrices and l one-dimensional bias vectors $\mathbf{W_i}, \mathbf{b_i}$, where $i \in 1..l$, and thus we have $f(\mathbf{x}) = \mathbf{W_l} \cdot \sigma(\mathbf{W_{l-1}} \cdots \cdot \sigma(\mathbf{W_1} \cdot \mathbf{x} + \mathbf{b_1}) \cdots + \mathbf{b_{l-1}}) + \mathbf{b_l}$, where $\sigma(\cdot)$ is the activation function applied element-wise to the input vector. The default choice of activation is typically the sigmoid $\sigma(x) = 1/(1 + e^{-x})$, tanh , or ReLU function $\sigma(x) = max(0, x)$, however recent work [18, 32, 31] has shown that functions such as $gelu(x)$ and $swish(x) = x \times sigmoid(x)$ can have better performance and desirable theoretical properties.

Unlike feed-forward neural networks, recurrent neural networks receive a sequence of inputs $[\mathbf{x^{(1)}}, \ldots, \mathbf{x^{(t)}}]$, and the final output of f on $\mathbf{x_t}$ is used to perform the classification of the whole sequence. Recurrent neural networks are *state-ful*, meaning they maintain a state vector that contains information about inputs previously given to f, which also gets updated on each call to f. In particular,

we focus on *long short-term memory* (LSTM) networks, which have seen wide adoption in natural language processing (NLP) tasks due to their sequential nature. For LSTMs trained for NLP tasks, the network receives a sequence of *word embeddings*. A word embedding is an n-dimensional vector that is associated with a particular word in a (natural) language. The distance between word embeddings carries semantic significance – two word embeddings that are close to each other in $\mathbb{R}^n$ typically have similar meanings or carry a semantic relatedness (e.g. *dog* and *cat* or *king* and *queen*), whereas unrelated words typically are farther apart.

LSTM networks further differ from feed-forward networks in that their internal activation functions are *two*-dimensional. Specifically, we have the following two activation patterns: $\sigma_1(x) \times \sigma_2(y)$ and $x \times \sigma_1(y)$. The default choices are $\sigma_1(x) = sigmoid(x)$, and $\sigma_2(x) = tanh(x)$. However, we can swap σ_1 with any function with output range bounded by $[0, 1]$, and swap σ_2 with any function with output range bounded by $[-1, 1]$. Indeed, prior work [16] has shown that $\sigma_1(x) = 1 - e^{e^{-x}}$ can achieve better results in some applications.

2.2 Neural Network Verification

A large number of problems in neural network verification can be phrased as the following: given an input region $X \subseteq \mathbb{X}$, compute an over-approximation Y, such that $\{f(\mathbf{x}) \mid \mathbf{x} \in X\} \subseteq Y \subseteq \mathbb{Y}$. Typically X and Y are hyper-boxes represented by an interval for each of their elements. A common problem is to prove that a point $\mathbf{x} \in \mathbb{X}$ is *robust*, meaning that small perturbations will not cause an incorrect classification. In this case, X is the set of all perturbed versions of $\mathbf{x}$, and to prove robustness, we check that the element of the correct class in Y has a lower bound that is greater than the upper bound of all other elements.

We illustrate a simple verification problem on the neural network shown in Fig. 2. The network has two inputs, x_1, x_2, and two outputs x_7, x_8 which represent scores for two different classes. We refer to the remaining hidden neurons as $x_i, i \in 3..6$. Following prior work [36], we break the affine transformation and application of the activation function into two separate neurons, and the neurons are assumed to be ordered such that, if x_i is in a layer before x_j, then $i < j$. For simplicity, in this motivating example, we let $\sigma(x) = max(0, x)$ (the ReLU function). We are interested in proving that the region $x_1 \in [-1, 1], x_2 \in [-1, 1]$ always maps to the first class, or in other words, we want to show that the lower bound of x_7 is greater than the upper bound x_8.

2.3 Existing Methods

The most scalable approaches (to date) for neural network verification are based on linear bounding and back-substitution [47], also referred to as abstract interpretation in the polyhedral abstract domain [36] or symbolic interval analysis [43] in prior work.

For each neuron x_j in the network, these approaches compute a concrete lower and upper bound l_j, u_j, and a linear lower and upper bound in terms of the previous layer's neurons. The linear bounds (regardless of the choice of $\sigma(\cdot)$)

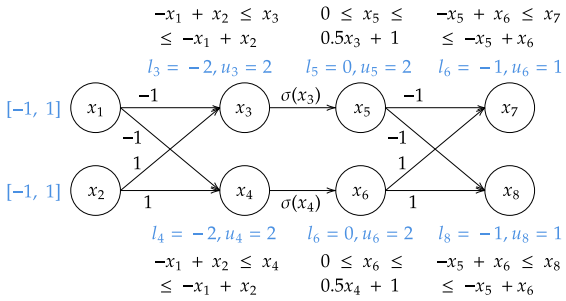

Fig. 2. Example of neural network verification.

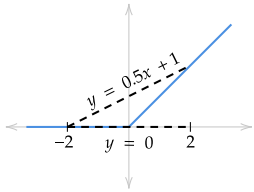

Fig. 3. Linear bounds for ReLU activation.

have the following form: $\sum_{i=0}^{j-1} x_i \cdot c_i^l + c_j^l \le x_j \le \sum_{i=0}^{j-1} x_i \cdot c_i^u + c_j^u$. The bounds are computed in a forward, layer-by-layer fashion which guarantees that any referenced neurons will already have a bound computed when back-substitution is performed.

To obtain the concrete bounds l_j, u_j for a neuron x_j, the bounds of any non-input neurons are recursively substituted into the linear bounds of x_j until only input nodes $x_1, ..., x_n$ remain. Finally, the concrete input intervals are substituted into the bound to obtain l_j, u_j.

Example We illustrate on the two-layer network in Fig. 2 for the previously defined property. We trivially have $l_1 = l_2 = -1$, $u_1 = u_2 = 1$, $-1 \le x_1 \le 1$, and $-1 \le x_2 \le 1$. We then compute linear bounds for x_3, x_4 in terms of previous layer's neurons x_1, x_2. We multiply x_1, x_2 by the edge weights, obtaining $-x_1+x_2$ as the lower and upper bound for both of x_3 and x_4. Since this bound is already in terms of the input variables, we substitute the concrete bounds into this equation and obtain $l_3 = l_4 = -2$ and $u_3 = u_4 = 2$.

Next, we need to compute the linear bounds for $x_5 = \sigma(x_3)$ and $x_6 = \sigma(x_4)$ after applying the activation function. Solving this challenge has been the focus of many prior works. There are two requirements. First, they need to be *sound*. For example, for x_5 we need to find coefficients $c_1^l, c_2^l, c_1^u, c_2^u$ such that $c_1^l x_3 + c_2^l \le \sigma(x_3) \le c_1^u x_3 + c_2^u$ for all $x_3 \in [l_3, u_3]$, and similarly for x_6. Second, we want them to be *tight*. Generally, this means that volume below the upper bound is minimized, and volume below the lower bound is maximized.

As an example, prior work [36, 48] proposed the following sound and tight bound for $\sigma(x) = max(0, x)$:

$$\forall x_i \in [l_i, u_i] \; . \; \frac{u_i}{u_i - l_i} x_i + \frac{-l_i u_i}{u_i - l_i} \le \sigma(x_i) \le \begin{cases} 0 & -l_i \ge u_i \\ x_i & -l_i < u_i \end{cases}$$

We illustrate the bound for x_5 in Fig. 3. After computing this bound, we recursively substitute variables in the bounds of x_5 with the appropriate bound, and compute l_5, u_5. The process then repeats for x_6, followed by x_7 and x_8. We then check $l_7 > u_8$ to verify the property, which fails in this case.

2.4 Limitations of Existing Methods

Current approaches only support a limited number of activation functions, and designing linear bounds for new activation functions often requires a significant amount of effort even for a domain expert. For example, handcrafted sound and tight linear bounds for activation functions such as ReLU, sigmoid, and tanh [36, 45, 48, 46, 44, 43], convolution layers and pooling operations [6], the two-dimensional activations found in LSTMs [25, 33], and those in transformer networks [34] are worthy of publication. Furthermore, even bounds that are hand-crafted by experts are not always tight. For example, a recent work [46] was able to nearly triple the precision of previous state-of-the-art sigmoid and tanh linear bounds simply by improving tightness.

To the best of our knowledge, AutoLiRPA [47] is the only tool that has the ability to handle more complex activation functions, though it was not originally designed for this. It can do so by decomposing them into simpler operations, and then composing the bounds together. We illustrate with $swish(x) = x \times sigmoid(x)$, where $x \in [-1.5, 5.5]$. AutoLiRPA would first bound $sigmoid(x)$ over the region $[-1.5, 5.5]$, resulting in the bound $.11x + .35 \leq sigmoid(x) \leq .22x + .51$. For the left-hand side of the function, we trivially have $x \leq x \leq x$. AutoLiRPA would then bound a multiplication $y \times z$, where in this case $y = x$ and $z = sigmoid(x)$, resulting in the final bound $-.15x - .495 \leq x \times sigmoid(x) \leq 0.825x + .96$. We illustrate this bound in Fig. 4, and we provide bounds computed by LinSyn as a comparison point. LinSyn provides a slightly better upper bound, and a significantly better lower bound. The reason for the looseness is because when AutoLiRPA bounds $sigmoid(x)$, it necessarily accumulates some approximation error because it is approximating the behavior of a non-linear function with linear bounds. The approximation error effectively "loses some information" about about its input variable x. Then, when bounding the multiplication operation, it has partially lost the information that y and z are related (i.e. they are both derived from x). In contrast, LinSyn overcomes this issue by considering $swish(x)$ as a whole. We explain how in the following sections.

3 Synthesizing the Candidate Linear Bounds

In this section, we describe our method for synthesizing candidate, possibly unsound linear bounds.

3.1 Problem Statement and Challenges

We assume we are given a d-dimensional activation function $z = \sigma(x_1, ..., x_d)$, and an input interval $x_i \in [l_i, u_i]$ for each $i \in \{1..d\}$. Our goal is to synthesize linear coefficients c_i^l, c_i^u, where $i \in \{1..d+1\}$ that are sound, meaning that the following condition holds:

$$\begin{gathered} \forall x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \ldots, x_d \in [l_d, u_d] \\ c_1^l x_1 + c_2^l x_2 + \cdots + c_{d+1}^l \leq \sigma(x_1, x_2, \ldots) \leq c_1^u x_1 + c_2^u x_2 + \cdots + c_{d+1}^u \end{gathered} \tag{2}$$

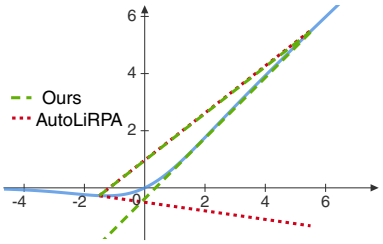

Fig. 4. Bounds computed by LinSyn and AutoLiRPA for $swish(x)$, $x \in [-1.5, 5.5]$.

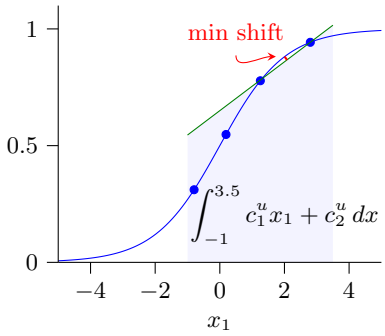

Fig. 5. Candidate plane synthesis.

In addition, we want to ensure that the bounds are *tight*. The ideal definition of tightness would choose linear bounds that maximize the precision of the overall analysis, for example minimizing the width of the output neuron's intervals. Unfortunately, such a measure would involve all of the neurons of the network, and so is impractical to compute. Instead, the common practice is to settle for tightness that's local to the specific neuron we are bounding.

Informally, we say a bound is *tight* if the volume below the upper bound is minimized, and volume below the lower bound is maximized. Prior work [48, 36, 25] has found this to be a good heuristic[1]. Formally, volume is defined as the following integral: $\int_{l_1}^{u_1} \cdots \int_{l_d}^{u_d} \sum_{i=1}^{d} c_i^u x_i + c_{d+1}^u \; dx_1 \ldots dx_d$ which, for the upper bound, should be minimized subject to Equation 2. This integral has the following closed-form solution:

$$\sum_{i=0}^{d} \left[\frac{1}{2} c_i \times \prod_{j=0}^{d} \left(u_i^{1+\mathbf{1}_{i=j}} - l_i^{1+\mathbf{1}_{i=j}} \right) \right] + c_{d+1} * \prod_{i=0}^{d} (u_i - l_i) \tag{3}$$

where $\mathbf{1}_{i=j}$ is the (pseudo Boolean) indicator function that returns 1 when its predicate is true. We omit the proof, but note that the above expression can be derived inductively on d. Also note that, since each l_i, u_i are concrete, the above expression is linear in terms of the coefficients, which will be advantageous in our approach below.

While recent approaches in solving non-linear optimization problems [26, 8] could directly minimize Equation 3 subject to Equation 2 in one step, we find the runtime to be very slow. Instead, we adopt a two-step approach that first uses efficient procedures for computing candidate coefficients that are almost sound (explained in this section), and second, only calls an SMT solver when necessary to verify Equation 2 (explained in the next section). We illustrate the approach on a concrete example.

[1] We also experimented with minimizing the volume between the linear bound and the activation function, which gave almost identical results.

3.2 Synthesizing Candidate Bounds

The first step in our approach computes candidate coefficients for the linear bound. In this step we focus on satisfying the tightness requirement, while making a best effort for soundness. We draw inspiration from prior work [33, 3] that leverages sampling to estimate the curvature of a particular function, and then uses a linear programming (LP) solver to compute a plane that is sound. However, unlike prior work which targeted a fixed function, we target arbitrary (activation) functions, and thus these are special cases of our approach.

The constraints of the LP are determined by a set of sample points $S \subset \mathbb{R}^d$. For the upper bound, we minimize Equation 3, subject to the constraint that the linear bound is above $\sigma(\cdot)$ at the points in S. Using $\mathbf{s}_i$ to refer to the i^{th} element of the vector $\mathbf{s} \in S$, the linear program we solve is:

$$\text{minimize Equation (3)} \;\; \text{subject to} \bigwedge_{\mathbf{s} \in S} c_1\mathbf{s}_1 + c_2\mathbf{s}_2 + \cdots + c_{d+1} \geq \sigma(\mathbf{s}) \tag{4}$$

We generate S by sampling uniformly-spaced points over the input intervals.

Example We demonstrate our approach on the running example illustrated in Fig. 5. For the example, let $\sigma(x_1) = \frac{1}{1+e^{-x_1}}$ (the sigmoid function, shown as the blue curve), where $x_1 \in [-1, 3.5]$. We focus only on the upper bound, but the lower bound is computed analogously.

Plugging in the variables into Equation 3, the objective of the LP that we minimize is: $\int_{-1}^{3.5} c_1^u x_1 + c_2^u \, dx_1 = 6.625c_1^u + 4.5c_2^u$ which is shown as the shaded region in Fig. 5.

We sample the points $S = \{-1, 0.25, 1.5, 2.75\}$, resulting in the following four constraints: $-c_1 + c_2 \geq \sigma(-1) \wedge 0.25c_1 + c_2 \geq \sigma(0.25) \wedge 1.5c_1 + c_s \geq \sigma(1.5) \wedge 2.75c_1 + c_2 \geq \sigma(2.75)$. Solving the LP program results in $c_1 = 0.104, c_2 = 0.649$, which is illustrated by the green line in Fig. 5.

4 Making the Bound Sound

In this section, we present our method for obtaining soundness because the candidate bounds synthesized in the previous section may not be sound. Here, we focus only on making the upper bound sound, but note the procedure for the lower bound is similar.

4.1 Problem Statement and Challenges

We are given the activation function $\sigma(\cdot)$, the input intervals $x_i \in [l_i, u_i]$, and the candidate coefficients $c_1, c_2, \ldots, c_{d+1}$. The goal is to compute an upward shift, if needed, to make the upper bound sound. First, we define the violation of the upper bound as:

$$v(x_1, x_2, \ldots, x_d) := c_1^u x_1 + c_2^u x_2 + \cdots + c_{d+1}^u - \sigma(x_1, x_2, \ldots, x_d) \tag{5}$$

A negative value indicates the upper bound is not sound. We then need to compute a lower bound on $v(\cdot)$, which we term v_l. Then the equation we pass to the verifier is:

$$\begin{gathered} \forall x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \ldots, x_d \in [l_d, u_d] \\ v(x_1, x_2, \ldots, x_d) + (-v_l) \geq 0 \end{gathered} \tag{6}$$

Expanding $v(\cdot)$ with its definition in the above equation results in the soundness definition of Equation 2. Thus, if the verifier proves Equation 6, then shifting the upper bound upward by $-v_l$ ensures its soundness. For our running example, the quantity v_l is shown by the red line in Fig. 5.

This problem is non-trivial because finding a solution for v_l requires a search for a sound global minimum/maximum of a function involving $\sigma(\cdot)$, which may be highly non-linear. State-of-the-art SMT solvers such as Z3 do not support all non-linear operations, and furthermore, since we assume arbitrary $\sigma(\cdot)$, the problem may even be (computationally) undecidable.

4.2 Verifying the Bound

We first assume we have a candidate (possibly unsound) v_l, and explain our verification method. To ensure decidability and tractability, we leverage the δ-*decision procedure* implemented by dReal [14]. To the best of our knowledge this is is the only framework that is decidable for all computable functions.

In this context, instead of verifying Equation 6, the formula is first negated thus changing it into an existentially quantified one, and then applying a δ-*relaxation*. Formally, the formula dReal attempts to solve is:

$$\begin{gathered} \exists x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \ldots, x_d \in [l_d, u_d] \\ v(x_1, x_2, \ldots) + (-v_l) \leq \delta \end{gathered} \tag{7}$$

where δ is a small constant (e.g. 10^{-5}), which we explain in a moment. The above is formulated such that Equation 6 holds if (but not only if) there does *not* exist a solution to Equation 7.

Internally, dReal performs interval constraint propagation (ICP) on the left-hand side of Equation 7 over the intervals defined by each $[l_i, u_i]$ to compute an upper bound, and compares this upper bound with δ. If the upper bound is less than δ, then no solution exists (i.e., Equation 7 is unsatisfiable, and we have proven the original Equation 6 holds). Otherwise a solution *may* exist. In this case, dReal iteratively partitions the input space defined by the $[l_i, u_i]$ and repeats this process on each partition separately.

dReal stops partitioning either when it proves all partitions do not have solutions , or when a partition whose intervals all have width less than some ϵ is found. Here, ϵ is proportional to δ (i.e., smaller δ means smaller ϵ). In the latter case, dReal returns this partition as a "solution".

While Equation 6 holds if there does not exist a solution to Equation 7, the converse does not hold true both because of the error inherent in ICP, and because we "relaxed" the right-hand side of Equation 7. This means that δ controls the *precision* of the analysis. δ controls both the size of the false solution

space, and determines how many times we will sub-divide the input space before giving up on proving Equation 7 to be unsatisfiable.

Practically, this has two implications for our approach. The first one is that our approach naturally inherits a degree of looseness in the linear bounds defined by δ. Specifically, we must shift our plane upward by δ in addition to the true v_l, so that dReal can verify the bound. The second is that we have to make a trade-off between computation and precision. While smaller δ will allow us to verify a tighter bound, it generally will also mean a longer verification time. In our experiments, we find that $\delta = 10^{-7}$ gives tight bounds at an acceptable runtime, though we may be able to achieve a shorter runtime with a larger δ.

4.3 Computing v_l

Now that we have defined how we can verify a candidate bound, we explain our approach for computing v_l. The implementation is outlined in Algorithm 1. Since failed calls to the verifier can be expensive, at lines 1-2, we first use a relatively cheap (and unsound) local optimization procedure to estimate the true v_l. While local optimization may get stuck in local minima, neural network activation functions typically do not have many local minima, so neither will $v(\cdot)$. We use L-BFGS-B [7], the bounded version of L-BFGS, to perform the optimization. At a high-level, L-BFGS-B takes as input $v(\cdot)$, the input bounds $x_i \in [l_i, u_i]$, and an initial guess $\mathbf{g} \in \mathbb{R}^d$ at the location of the local minimum. It then uses the Jacobian matrix (i.e., derivatives) of $v(\cdot)$ to iteratively move towards the local minimum (the Jacobian can be estimated using the finite differences method or provided explicitly – we use Mathematica [21] to obtain it). We find that sampling points uniformly in $v(\cdot)$ can usually find a good $\mathbf{g}$, and thus L-BFGS-B often converges in a small number of iterations. L-BFGS-B typically produces an estimate within 10^{-8} of the true value. To account for estimation error we add an additional 10^{-6}, plus $2 \times \delta$ to account for the δ-relaxation (line 3). Finally, we iteratively decrease v_l by a small amount (10^{-6}) until dReal verifies it (lines 4-9).

Going back to our motivating example, we would estimate v_l with a local minimizer, and then use dReal to verify the following:

$$\forall x_1 \in [-1, 3.5] \; . \; \sigma(x_1) \leq c_1^u x_1 + c_2^u + (-v_l) + 2 \times \delta + 10^{-6}$$

If verification fails, we iteratively decrease the value of v_l by 10^{-6}, and call dReal until the bound is verified. The final value of $c_1^u x_1 + c_2^u + (-v_l) + 2 \times \delta + 10^{-6}$ is the final sound upper bound.

4.4 On the Correctness and Generality of LinSyn

The full LinSyn procedure is shown in Algorithm 2. The correctness (i.e. soundness) of the synthesized bounds is guaranteed if the v_l returned by Algorithm 1 is a true lower bound on $v(\cdot)$. Since Algorithm 1 does not return until dReal verifies v_l at line 6, the correctness is guaranteed.

Both our procedure in Section 3 and L-BFGS-B require only black-box access to $\sigma(\cdot)$, so the only potential limit to the arbitrariness of our approach lies in

Algorithm 1: BoundViolation

Input: Activation $\sigma(x_1, x_2, \dots)$, Candidate Coefficients $c_1^u, c_2^u, \dots, c_{d+1}^u$, Input Bounds $x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \dots$, Jacobian ∇v (optional)
Output: Lower Bound on Violation v_l

$\mathbf{g} \leftarrow$ sample points on $v(x_1, x_2, \dots)$ and take minimum;
$v_l \leftarrow$ **L-BFGS-B**$(v(x_1, x_2, \dots), x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \dots, \mathbf{g}, \nabla v)$;
$v_l \leftarrow v_l - 10^{-6} - 2\delta$;
while ***True*** **do**
  // Call dReal
  if *Equation 2 holds* **then**
    return v_l;
  end
  $v_l \leftarrow v_l - 10^{-6}$;
end

Algorithm 2: SynthesizeUpperBoundCoefficients

Input: Activation $\sigma(x_1, x_2, \dots)$, Input Bounds $x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \dots$, Jacobian ∇v (optional)
Output: Sound Coefficients $c_1^u, c_2^u, \dots, c_{d+1}^u$

$c_1^u, c_2^u, \dots, c_{d+1}^u \leftarrow$ Sampling and LP procedure on $\sigma(x)$ over Input Bounds;
$v_l \leftarrow$ BoundViolation$(c_1^u, c_2^u, \dots, c_{d+1}^u, x_1 \in [l_1, u_1], x_2 \in [l_2, u_2], \dots, \nabla v)$;
$c_{d+1}^u \leftarrow c_{d+1}^u + (-v_l)$;
return $c_1^u, c_2^u, \dots, c_{d+1}^u$;

what elementary operations are supported by dReal. During our investigation, we did not find activations that use operations unsupported by dReal, however if an unsupported operation is encountered, one would only need to define an *interval extension* [28] for the operation, which can be done for any computable function.

5 Evaluation

We have implemented our method in a module called LinSyn, and integrated it into the AutoLiRPA neural network verification framework [47]. A user instantiates LinSyn with a definition of an activation function, which results in an executable software module capable of computing the sound linear lower and upper bounds for the activation function over a given input region. LinSyn uses Gurobi [17] to solve the LP problem described in Section 3, and dReal [14] as the verifier described in 4. In total, LinSyn is implemented in about 1200 lines of Python code.

5.1 Benchmarks

Neural Networks Our benchmarks are nine deep neural networks trained on the three different datasets shown below. In the following, a neuron is a node in the

neural network where a linear bound must be computed, and thus the neuron counts indicate the number of calls to LinSyn that must be made.

- **MNIST:** MNIST is a dataset of hand-written integers labeled with the corresponding integer in the image. The images have 28x28 pixels, with each pixel taking a gray-scale value between 0 to 255. We trained three variants of a 4-layer CNN (convolutional neural network). Each takes as input a 28x28 = 784-dimensional input vector and outputs 10 scores, one for each class. In total, each network has 2,608 neurons – 1568, 784, and 256 in the first, second, and third layers, respectively.
- **CIFAR:** CIFAR is a dataset of RGB images from 10 different classes. The images have 32x32 pixels, with each pixel having an R, G, and B value in the range 0 to 255. We trained three variants of a 5-layer CNN. Each takes a 32x32x3 = 3072-dimensional input vector and outputs 10 scores, one for each class. In total, each network has 5376 neurons, 2048, 2048, 1024, and 256 neurons in the first, second, third, and fourth layers, respectively.
- **SST-2:** The Stanford Sentiment Treebank (SST) dataset consists of sentences taken from movie reviews that are human annotated with either positive or negative, indicating the sentiment expressed in the sentence. We trained three different variants of the standard LSTM architecture. These networks take as input a sequence 64-dimensional word embeddings and output 2 scores, one for positive and one for negative. Each network has a hidden size of 64, which works out to 384 neurons per input in the input sequence.

Activation Functions We experimented with the four activation functions as shown in Fig. 6. *GELU* and *Swish* were recently proposed alternatives to the standard ReLU function due to their desirable theoretical properties [18] such as reduced overfitting [38], and they have seen use in OpenAI's GPT [31] and very deep feed forward networks [32]. Similarly, *Hard-Tanh* is an optimized version of the common tanh function, while the *Log-Log* function [16] is a sigmoid-like function used in forecasting.

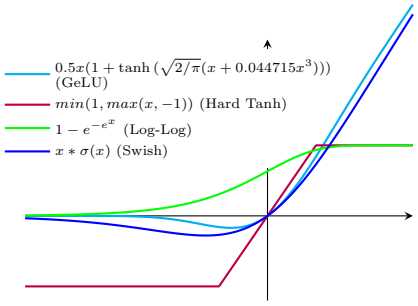

Fig. 6. Nonlinear activation functions.

The Verification Problem The verification problem we consider is to certify that an input is robust to bounded perturbations of magnitude ϵ, where ϵ is a small number. *Certifying* means proving that the classification result of the neural network does not change in the presence of perturbations. We focus on l_∞ robustness, where we take an input $\mathbf{x} \in \mathbb{R}^n$ and allow a bounded perturbation of $+/-\epsilon$ to each element in $\mathbf{x}$. For each network, we take 100 random test inputs, filter out those that are incorrectly classified, apply an ϵ bounded perturbation

Table 1. Comparing certified accuracy and run time of LinSyn and AutoLiRPA.

Network Architecture		AutoLiRPA [47]		Our Method (new)	
		% certified	time (s)	% certified	time(s)
MNIST	4-Layer CNN with Swish	0.34	15	0.76	796
	4-Layer CNN with Gelu	0.01	359	0.72	814
	4-Layer CNN with Log Log	0.00	38	0.24	867
CIFAR	5-Layer CNN with Swish	0.03	69	0.35	1,077
	5-Layer CNN with Gelu	0.00	1,217	0.31	1,163
	5-Layer CNN with Log Log	0.59	98	0.69	717
SST-2	LSTM with sig tanh	0.93	37	0.91	1,074
	LSTM with hard tanh	-	-	0.64	2300
	LSTM with log log	0.16	1,072	0.82	2,859

Table 2. Comparing certified accuracy and run time of LinSyn and POPQORN.

Network Architecture		POPQORN [25]		Our Method (new)	
		% certified	time (s)	% certified	time(s)
SST-2	LSTM with sig tanh	0.93	1517	0.90	1,074

to the correctly classified inputs, and then attempt to prove the classification remains correct. We choose ϵ values common in prior work. For MNIST networks, in particular, we choose $\epsilon = 8/255$. For CIFAR networks, we choose $\epsilon = 1/255$. For SST-2 networks, we choose $\epsilon = 0.04$, and we only apply it to the first word embedding in the input sequence.

5.2 Experimental Results

Our experiments were designed to answer the following two questions: (1) How do LinSyn's linear bounds compare with handcrafted bounds? (2) How does the runtime of LinSyn compare to state-of-the-art linear bounding techniques? To answer these questions, we compare the effectiveness of LinSyn's linear bounds with the state-of-the-art linear bounding technique implemented in AutoLiRPA. To the best of our knowledge this is the only tool that can handle the activation functions we use in our benchmarks. As another comparison point, we also compare with POPQORN, a state-of-the-art linear bounding technique for LSTM networks. POPQORN tackles the challenge of computing tight linear bounds for $sigmoid(x) \times tanh(y)$ and $x \times sigmoid(y)$ using an expensive gradient descent based approach, and thus makes a good comparison point for runtime and accuracy. Our experiments were conducted on a computer with an Intel 2.6 GHz i7-6700 8-core CPU and 32GB RAM. Both AutoLiRPA and LinSyn are engineered to bound individual neurons in parallel. We configure each method to use up to 6 threads.

Overall Comparison First, we compare the overall performance of our new method and the default linear bounding technique in AutoLiRPA. The results are shown in Table 1. Here, Columns 1-2 show the name of the dataset and the type of neural networks. Columns 3-4 show the results of the default AutoLiRPA, including the percentage of inputs certified and the analysis time in seconds. Similarly, Columns 5-6 show the results of our new method.

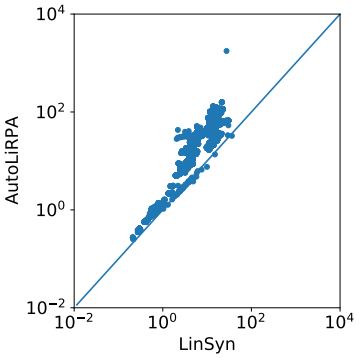

Fig. 7. Scatter plot comparing the final output interval width of LinSyn and AutoLiRPA.

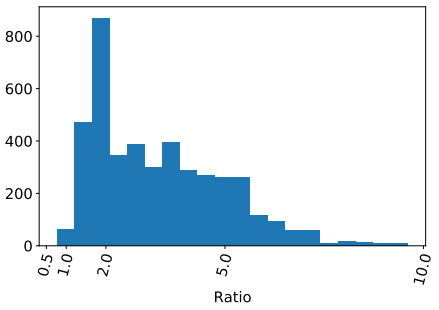

Fig. 8. Histogram of width ratios between AutoLiRPA and LinSyn. Ratio reported as $\frac{\text{AutoLiRPA}}{\text{LinSyn}}$.

The results in Table 1 show that, in terms of the analysis time, our method is slower, primarily due to the use of constraint solvers (namely dReal and the LP solver) but overall, the analysis speed is comparable to AutoLiRPA. However, in terms of accuracy, our method significantly outperforms AutoLiRPA. In almost all cases, our method was able to certify a much higher percentage of the inputs. For example, LinSyn more than quadruples the certified robustness of the *LSTM with log log* benchmark, and handles very well the relatively complex GeLU function. As for *SST-2: LSTM with hard tanh*, AutoLiRPA does not support the general $max(x, y)$ operation, so a comparison is not possible without significant engineering work.

The only exception to the improvement is *SST-2: LSTM with sig tanh*, for which the results are similar (.93 versus .91). In this case, there is likely little to be gained over the default, decomposition-based approach of AutoLiRPA in terms of tightness because the inputs to $sigmoid(x) \times tanh(y)$ and $x \times sigmoid(y)$ are not related, i.e., x and y are two separate variables. This is in contrast to, e.g., $swish(x) = x \times sigmoid(x)$, where the left-hand side and right-hand side of the multiplication *are* related.

In Table 2, we show a comparison between LinSyn and POPQORN. The result shows that our approach achieves similar certified robustness and runtime, even though POPQORN was designed to specifically target this particular type of LSTM architecture, while LinSyn is entirely generic.

Detailed Comparison Next, we perform a more in depth comparison of accuracy by comparing the widths of the final output neuron's intervals that are computed by AutoLiRPA and LinSyn. The results are shown in the scatter plot in Fig. 7 and the histogram in Fig. 8. Each point in the scatter plot represents a single output neuron x_i for a single verification problem. The x-axis is the width of the interval of the output neuron x_i (i.e. $u_i - l_i$) computed by LinSyn, and the y-axis is the width computed by AutoLiRPA. A point above the diagonal

line indicates that LinSyn computed a tighter (smaller) final output interval. In the histogram, we further illustrate the accuracy gain as the width ratio, measured as $\frac{\text{AutoLiRPA}}{\text{LinSyn}}$. Overall, the results show that LinSyn is more accurate in nearly all cases, and LinSyn often produces final output bounds 2-5X tighter than AutoLiRPA.

6 Related Work

Linear Bound-based Neural Network Verification There is a large body of work on using linear-bounding techniques [36, 48, 34, 6, 45, 29, 30, 46, 27] and other abstract domains such as concrete intervals, symbolic intervals [44], and Zonotopes [15], for the purpose of neural network verification. All of these can be thought of as leveraging restricted versions of the polyhedral abstract domain [10, 9]. To the best of our knowledge, these approaches are the most scalable (in terms of network size) due to the use of approximations, but this also means they are less accurate than exact approaches. In addition, all these approaches have the limitation that they depend on bounds that are hand-crafted by an expert.

SMT solver-based Neural Network Verification There is also a large body of work on using exact constraint solving for neural network verification. Early works include solvers specifically designed for neural networks, such as Reluplex and Marabou [23, 24] and others [11], and leveraging existing solvers [12, 20, 5, 20, 4, 40, 19]. While more accurate, the reliance on an SMT solver typically limits their scalability. More recent work often uses solvers to refine the bounds computed by linear bounding [35, 37, 43, 42, 41]. Since the solvers leveraged in these approaches usually involve linear constraint solving techniques, they are usually only applicable to piece-wise linear activation functions such as ReLU and Max/Min-pooling.

7 Conclusions

We have presented LinSyn, a method for synthesizing linear bounds for arbitrary activation functions. The key advantage of LinSyn is that it can handle complex activation functions, such as Swish, GELU, and Log Log as a whole, allowing it to synthesize much tighter linear bounds than existing tools. Our experimental results show this increased tightness leads to drastically increased certified robustness, and tighter final output bounds.

References

1. Eran. https://github.com/eth-sri/eran (2021)
2. Alzantot, M., Sharma, Y., Elgohary, A., Ho, B.J., Srivastava, M., Chang, K.W.: Generating natural language adversarial examples. arXiv preprint arXiv:1804.07998 (2018)
3. Balunović, M., Baader, M., Singh, G., Gehr, T., Vechev, M.: Certifying geometric robustness of neural networks. Advances in Neural Information Processing Systems 32 (2019)
4. Baluta, T., Shen, S., Shinde, S., Meel, K.S., Saxena, P.: Quantitative verification of neural networks and its security applications. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1249–1264 (2019)
5. Bastani, O., Ioannou, Y., Lampropoulos, L., Vytiniotis, D., Nori, A.V., Criminisi, A.: Measuring neural net robustness with constraints. In: Annual Conference on Neural Information Processing Systems. pp. 2613–2621 (2016)
6. Boopathy, A., Weng, T.W., Chen, P.Y., Liu, S., Daniel, L.: Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33, pp. 3240–3247 (2019)
7. Byrd, R.H., Lu, P., Nocedal, J., Zhu, C.: A limited memory algorithm for bound constrained optimization. SIAM Journal on scientific computing **16**(5), 1190–1208 (1995)
8. Chabert, G., Jaulin, L.: Contractor programming. Artificial Intelligence **173**(11), 1079–1100 (2009)
9. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 238–252 (1977)
10. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 84–96 (1978)
11. Dvijotham, K., Stanforth, R., Gowal, S., Mann, T.A., Kohli, P.: A dual approach to scalable verification of deep networks. In: International Conference on Uncertainty in Artificial Intelligence. pp. 550–559 (2018)
12. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings. pp. 269–286 (2017)
13. Engstrom, L., Tran, B., Tsipras, D., Schmidt, L., Madry, A.: Exploring the landscape of spatial robustness. In: International Conference on Machine Learning. pp. 1802–1811. PMLR (2019)
14. Gao, S., Kong, S., Clarke, E.M.: dreal: An smt solver for nonlinear theories over the reals. In: International conference on automated deduction. pp. 208–214. Springer (2013)
15. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: IEEE Symposium on Security and Privacy. pp. 3–18 (2018)
16. Gomes, G.S.d.S., Ludermir, T.B.: Complementary log-log and probit: activation functions implemented in artificial neural networks. In: 2008 Eighth International Conference on Hybrid Intelligent Systems. pp. 939–942. IEEE (2008)
17. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2021), https://www.gurobi.com

18. Hendrycks, D., Gimpel, K.: Gaussian error linear units (gelus). arXiv preprint arXiv:1606.08415 (2016)
19. Hu, H., Fazlyab, M., Morari, M., Pappas, G.J.: Reach-sdp: Reachability analysis of closed-loop systems with neural network controllers via semidefinite programming. In: 2020 59th IEEE Conference on Decision and Control (CDC). pp. 5929–5934. IEEE (2020)
20. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: International Conference on Computer Aided Verification. pp. 3–29 (2017)
21. Inc., W.R.: Mathematica, Version 12.3.1, https://www.wolfram.com/mathematica, champaign, IL, 2021
22. Kanbak, C., Moosavi-Dezfooli, S.M., Frossard, P.: Geometric robustness of deep networks: analysis and improvement. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 4441–4449 (2018)
23. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: International Conference on Computer Aided Verification. pp. 97–117 (2017)
24. Katz, G., Huang, D.A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljic, A., Dill, D.L., Kochenderfer, M.J., Barrett, C.W.: The Marabou framework for verification and analysis of deep neural networks. In: International Conference on Computer Aided Verification. pp. 443–452 (2019)
25. Ko, C.Y., Lyu, Z., Weng, L., Daniel, L., Wong, N., Lin, D.: Popqorn: Quantifying robustness of recurrent neural networks. In: International Conference on Machine Learning. pp. 3468–3477. PMLR (2019)
26. Kong, S., Solar-Lezama, A., Gao, S.: Delta-decision procedures for exists-forall problems over the reals. In: International Conference on Computer Aided Verification. pp. 219–235. Springer (2018)
27. Mohammadinejad, S., Paulsen, B., Wang, C., Deshmukh, J.V.: Diffrnn: Differential verification of recurrent neural networks. arXiv preprint arXiv:2007.10135 (2020)
28. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to interval analysis, vol. 110. Siam (2009)
29. Paulsen, B., Wang, J., Wang, C.: Reludiff: Differential verification of deep neural networks. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). pp. 714–726. IEEE (2020)
30. Paulsen, B., Wang, J., Wang, J., Wang, C.: Neurodiff: scalable differential verification of neural networks using fine-grained approximation. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 784–796. IEEE (2020)
31. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I.: Improving language understanding by generative pre-training (2018)
32. Ramachandran, P., Zoph, B., Le, Q.V.: Searching for activation functions. arXiv preprint arXiv:1710.05941 (2017)
33. Ryou, W., Chen, J., Balunovic, M., Singh, G., Dan, A., Vechev, M.: Scalable polyhedral verification of recurrent neural networks. In: International Conference on Computer Aided Verification. pp. 225–248. Springer (2021)
34. Shi, Z., Zhang, H., Chang, K.W., Huang, M., Hsieh, C.J.: Robustness verification for transformers. International Conference on Learning Representations (2020)
35. Singh, G., Ganvir, R., Pschel, M., Vechev, M.: Beyond the single neuron convex barrier for neural network certification. In: Advances in Neural Information Processing Systems (NeurIPS) (2019)
36. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages pp. 41:1–41:30 (2019)

37. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: Boosting robustness certification of neural networks. In: International Conference on Learning Representations (2019)
38. Singla, V., Singla, S., Feizi, S., Jacobs, D.: Low curvature activations reduce overfitting in adversarial training. In: Proceedings of the IEEE/CVF International Conference on Computer Vision. pp. 16423–16433 (2021)
39. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R.: Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199 (2013)
40. Tjeng, V., Xiao, K., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. International Conference on Learning Representations (2019)
41. Tran, H.D., Bak, S., Xiang, W., Johnson, T.T.: Verification of deep convolutional neural networks using imagestars. In: International Conference on Computer Aided Verification. pp. 18–42. Springer (2020)
42. Tran, H.D., Lopez, D.M., Musau, P., Yang, X., Nguyen, L.V., Xiang, W., Johnson, T.T.: Star-based reachability analysis of deep neural networks. In: International Symposium on Formal Methods. pp. 670–686. Springer (2019)
43. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: Annual Conference on Neural Information Processing Systems. pp. 6369–6379 (2018)
44. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: USENIX Security Symposium. pp. 1599–1614 (2018)
45. Weng, T., Zhang, H., Chen, H., Song, Z., Hsieh, C., Daniel, L., Boning, D.S., Dhillon, I.S.: Towards fast computation of certified robustness for relu networks. In: International Conference on Machine Learning. pp. 5273–5282 (2018)
46. Wu, Y., Zhang, M.: Tightening robustness verification of convolutional neural networks with fine-grained linear approximation. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 35, pp. 11674–11681 (2021)
47. Xu, K., Shi, Z., Zhang, H., Wang, Y., Chang, K.W., Huang, M., Kailkhura, B., Lin, X., Hsieh, C.J.: Automatic perturbation analysis for scalable certified robustness and beyond. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M.F., Lin, H. (eds.) Advances in Neural Information Processing Systems. vol. 33, pp. 1129–1141. Curran Associates, Inc. (2020), https://proceedings.neurips.cc/paper/2020/file/0cbc5671ae26f67871cb914d81ef8fc1-Paper.pdf
48. Zhang, H., Weng, T.W., Chen, P.Y., Hsieh, C.J., Daniel, L.: Efficient neural network robustness certification with general activation functions. In: Advances in neural information processing systems. pp. 4939–4948 (2018)