# Example Guided Synthesis of Linear Approximations for Neural Network Verification

CAV Artifact Evaluation ★ Available

CAV Artifact Evaluation ★★★ Reusable

Brandon Paulsen(✉) and Chao Wang

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

**Abstract.** Linear approximations of nonlinear functions have a wide range of applications such as rigorous global optimization and, recently, verification problems involving neural networks. In the latter case, a linear approximation must be hand-crafted for the neural network's activation functions. This hand-crafting is tedious, potentially error-prone, and requires an expert to prove the soundness of the linear approximation. Such a limitation is at odds with the rapidly advancing deep learning field – current verification tools either lack the necessary linear approximation, or perform poorly on neural networks with state-of-the-art activation functions. In this work, we consider the problem of automatically synthesizing sound linear approximations for a given neural network activation function. Our approach is *example-guided*: we develop a procedure to generate examples, and then we leverage machine learning techniques to learn a (static) function that outputs linear approximations. However, since the machine learning techniques we employ do not come with formal guarantees, the resulting synthesized function may produce linear approximations with violations. To remedy this, we bound the maximum violation using rigorous global optimization techniques, and then adjust the synthesized linear approximation accordingly to ensure soundness. We evaluate our approach on several neural network verification tasks. Our evaluation shows that the automatically synthesized linear approximations greatly improve the accuracy (i.e., in terms of the number of verification problems solved) compared to hand-crafted linear approximations in state-of-the-art neural network verification tools. An artifact with our code and experimental scripts is available at: https://zenodo.org/record/6525186#.Yp51L9LMIzM.

## 1 Introduction

Neural networks have become a popular model choice in machine learning due to their performance across a wide variety of tasks ranging from image classification, natural language processing, and control. However, they are also known

This work was partially funded by the U.S. National Science Foundation grants CNS-1813117 and CNS-1722710.

to misclassify inputs in the presence of both small amounts of input noise and seemingly insignificant perturbations to the inputs [37]. Indeed, many works have shown they are vulnerable to a variety of seemingly benign input transformations [1,9,17], which raises concerns about their deployment in safety-critical systems. As a result, a large number of works have proposed verification techniques to prove that a neural network is not vulnerable to these perturbations [35,43,44], or in general satisfies some specification [15,18,27,28].

Crucial to the precision and scalability of these verification techniques are *linear approximations* of the network's activation functions.

In essence, given some arbitrary activation function $\sigma(x)$, a linear approximation is a *coefficient generator function* $\mathcal{G}(l, u) \rightarrow \langle a_l, b_l, a_u, b_u\rangle$, where $l, u \in \mathbb{R}$ are real values that correspond to the interval $[l, u]$, and $a_l, b_l, a_u, b_u \in \mathbb{R}$ are real-valued coefficients in the linear lower and upper bounds such that the following condition holds:

$$\forall x \in [l, u].\ \ a_l \cdot x + b_l \leq \sigma(x) \leq a_u \cdot x + b_u \tag{1}$$

Indeed, a key contribution in many seminal works on neural network verification was a hand-crafted $\mathcal{G}(l, u)$ [2,7,19,33–35,42–45,47] and follow-up work built off these hand-crafted approximations [36,38]. Furthermore, linear approximations have applications beyond neural network verification, such as rigorous global optimization and verification [21,40].

However, crafting $\mathcal{G}(l, u)$ is tedious, error-prone, and requires an expert. Unfortunately, in the case of neural network activation functions, experts have only crafted approximations for the most common functions, namely ReLU, sigmoid, tanh, max-pooling, and those in vanilla LSTMs. As a result, existing techniques cannot handle new and cutting-edge activation functions, such as Swish [31], GELU [14], Mish [24], and LiSHT [32].

In this work, we consider the problem of automatically synthesizing the coefficient generator function $\mathcal{G}(l, u)$, which can alternatively be viewed as four individual functions $\mathcal{G}_{a_l}(l, u)$, $\mathcal{G}_{b_l}(l, u)$, $\mathcal{G}_{a_u}(l, u)$, and $\mathcal{G}_{b_u}(l, u)$, one for each coefficient. However, synthesizing the generator functions is a challenging task because (1) the search space for each function is very large (in fact, technically infinite), (2) the optimal generator functions are highly nonlinear for all activation functions considered both in our work and prior work, and (3) to prove soundness of the synthesized generator functions, we must show:

$$\begin{gathered}\forall [l, u] \in \mathbb{IR}, x \in [l, u]\ . \\ (\mathcal{G}_{a_l}(l, u) \cdot x + \mathcal{G}_{b_l}(l, u)) \leq \sigma(x) \leq (\mathcal{G}_{a_u}(l, u) \cdot x + \mathcal{G}_{b_u}(l, u))\end{gathered} \tag{2}$$

where $\mathbb{IR} = \{[l, u] \mid l, u \in \mathbb{R}, l \leq u\}$ is the set of all real intervals. The above equation has highly non-linear constraints, which cannot be directly handled by standard verification tools, such as the Z3 [6] SMT solver.

To solve the problem, we propose a novel example-guided synthesis and verification approach, which is applicable to any differentiable, Lipschitz-continuous activation function $\sigma(x)$. (We note that activation functions are typically required to be differentiable and Lipschitz-continuous in order to be trained

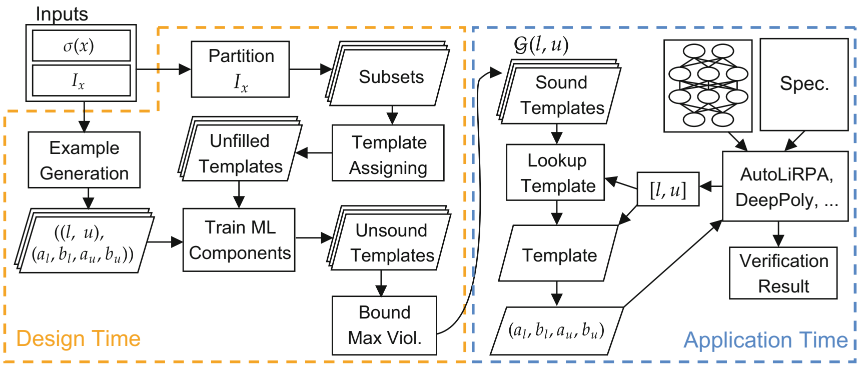

**Fig. 1.** Overview of our method for synthesizing the *coefficient generator function*.

by gradient descent, thus our approach applies to any *practical* activation function). To tackle the potentially infinite search space of $\mathcal{G}(l, u)$, we first propose two *templates* for $\mathcal{G}(l, u)$, which are inspired by the hand-crafted coefficient functions of prior work. The "holes" in each template are filled by a machine learning model, in our case a small neural network or linear regression model. Then, the first step is to partition the input space of $\mathcal{G}(l, u)$, and then assign a single template to each subset in the partition. The second step is to fill in the holes of each template. Our approach leverages an example-generation procedure to produce a large number of training examples of the form $((l, u), (a_l, b_l, a_u, b_u))$, which can then be used to train the machine learning component in the template. However, a template instantiated with a trained model may still violate Eq. 2, specifically the lower bound (resp. upper bound) may be above (resp. below) the activation function over some interval $[l, u]$. To ensure soundness, the final step is to bound the *maximum violation* of a particular template instance using a rigorous global optimization technique based on interval analysis, which is implemented by the tool IbexOpt [5]. We then use the computed maximum violation to adjust the template to ensure Eq. 2 always holds.

The overall flow of our method is shown in Fig. 1. It takes as input the activation function $\sigma(x)$, and the set of input intervals $I_x \subseteq \mathbb{R}$ for which $\mathcal{G}(l, u)$ will be valid. During *design time*, we follow the previously described approach, which outputs a set of sound, instantiated templates which make up $\mathcal{G}(l, u)$. Then the synthesized $\mathcal{G}(l, u)$ is integrated into an existing verification tool such as AutoLiRPA [46] or DeepPoly [35]. These tools take as input a neural network and a specification, and output the verification result (proved, counterexample, or unknown). At *application time* (i.e., when attempting to verify the input specification), when these tools need a linear approximation for $\sigma(x)$ over the interval $[l, u]$, we lookup the appropriate template instance, and use it to compute the linear approximation $(a_l, b_l, a_u, b_u)$, and return it to the tool.

To the best of our knowledge, our method is the first to synthesize a linear approximation generator function $\mathcal{G}(l, u)$ for any given activation function

$\sigma(x)$. Our approach is fundamentally different from the ones used by state-of-the-art neural network verification tools such as AutoLiRPA and DeepPoly, which require an expert to hand-craft the approximations. We note that, while AutoLiRPA can handle activations that it does not explicitly support by *decomposing* $\sigma(x)$ into elementary operations for which it has (hand-crafted) linear approximations, and then combining them, the resulting bounds are often not tight. In contrast, our method synthesizes linear approximations for $\sigma(x)$ as a whole, and we show experimentally that our synthesized approximations significantly outperform AutoLiRPA.

We have implemented our approach and evaluated it on popular neural network verification problems (specifically, robustness verification problems in the presence of input perturbations). Compared against state-of-the-art linear approximation based verification tools, our synthesized linear approximations can drastically outperform these existing tools in terms of the number of problems verified on recently published activation functions such as Swish [31], GELU [14], Mish [24], and LiSHT [32].

To summarize, we make the following contributions:

– We propose the first method for synthesizing the linear approximation generator function $\mathcal{G}(l, u)$ for any given activation function.
– We implement our method, use it to synthesize linear approximations for several novel activation functions, and integrate these approximations into a state-of-the-art neural network verification tool.
– We evaluate our method on a large number of neural network verification problems, and show that our synthesized approximations significantly outperform the state-of-the-art tools.

## 2 Preliminaries

In this section, we discuss background knowledge necessary to understand our work. Throughout the paper, we will use the following notations: for variables or scalars we use lower case letters (e.g., $x \in \mathbb{R}$), for vectors we use bold lower case letters (e.g., $\mathbf{x} \in \mathbb{R}^n$) and for matrices we use bold upper case letters (e.g., $\mathbf{W} \in \mathbb{R}^{n \times m}$). In addition, we use standard interval notation: we let $[l, u] = \{x \in \mathbb{R} | l \leq x \leq u\}$ be a real-valued interval, we denote the set of all real intervals as $\mathbb{IR} = \{[l, u] | l, u \in \mathbb{R}, l \leq u\}$, and finally we define the set of $n$-dimensional intervals as $\mathbb{IR}^n = \{\bigtimes_{i=1}^{n} [l_i, u_i] \mid [l_i, u_i] \in \mathbb{IR}\}$, where $\bigtimes$ is the Cartesian product.

### 2.1 Neural Networks

We consider a neural network to be a function $f : \mathbb{X} \subseteq \mathbb{R}^n \rightarrow \mathbb{Y} \subseteq \mathbb{R}^m$, which has $n$ inputs and $m$ outputs. For ease of presentation, we focus the discussion on *feed-forward, fully-connected* neural networks (although the bounds synthesized by our method apply to all neural network architectures). For $\mathbf{x} \in \mathbb{X}$, such networks compute $f(\mathbf{x})$ by performing an alternating series of matrix multiplications followed by the element-wise application of an activation function $\sigma(x)$.

Formally, an $l$-layer neural network with $k_i$ neurons in each layer (and letting $k_0 = n, k_l = m$) has $l$ weight matrices and bias vectors $\mathbf{W}_i \in \mathbb{R}^{k_{i-1} \times k_i}$ and $\mathbf{b}_i \in \mathbb{R}^{k_i}$ for $i \in \{1..l\}$. The input of the network is $f_0 = \mathbf{x}^T$, and the output of layer $i$ is given by the function: $f_i = \sigma(f_{i-1} \cdot \mathbf{W}_i + \mathbf{b}_i)$ which can be applied recursively until the output layer of the network is reached.

Initially, common choices for the activation function $\sigma(x)$ were $ReLU(x) = max(0, x)$, $sigmoid(x) = \frac{e^x}{e^x+1}$, and $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, however the field has advanced rapidly in recent years and, as a result, automatically discovering novel activations has become a research subfield of its own [31]. Many recently proposed activations, such as Swish and GELU [14,31], have been shown to outperform the common choices in important machine learning tasks.

## 2.2 Existing Neural Network Verification Techniques and Limitations

We consider neural network verification problems of the following form: given a neural network $f : \mathbb{X} \to \mathbb{Y}$ and an input set $X \subseteq \mathbb{X}$, compute an over-approximation $Y$ such that $\{f(\mathbf{x}) \mid \mathbf{x} \in X\} \subseteq Y \subseteq \mathbb{Y}$. The most scalable approaches to neural network verification (where scale is measured by number of neurons in the network) use linear bounding techniques to compute $Y$, which require a *linear approximation* of the network's activation function. This is an extension of *interval analysis* [26] (e.g., intervals with linear lower/upper bounds [35,46]) to compute $Y$, and thus $X$ and $Y$ are represented as elements of $\mathbb{IR}^n$ and $\mathbb{IR}^m$, respectively.

We use Fig. 2 to illustrate a typical neural network verification problem. The network has input neurons $x_1, x_2$, output neurons $x_7, x_8$ and a single hidden layer. We assume the activation function is $swish(x) = x \cdot sigmoid(x)$, which is shown by the blue line in Fig. 3. Our input space is $X = [-1, 1] \times [-1, 1]$ (i.e., $x_1, x_2 \in [-1, 1]$), and we want to prove $x_7 > x_8$, which can be accomplished by first computing the bounds $x_7 \in [l_7, u_7], x_8 \in [l_8, u_8]$, and then showing $l_7 > u_8$. Following the prior work [35] and for simplicity, we split the affine transformation and application of activation function in the hidden layer into two steps, and we assume the neurons $x_i$, where $i \in \{1..8\}$, are ordered such that $i < j$ implies that $x_i$ is in either the same layer as $x_j$, or a layer prior to $x_j$.

Linear bounding based neural network verification techniques work as follows. For each neuron $x_i$, they compute the concrete lower and upper bounds $l_i$ and $u_i$, together with symbolic lower and upper bounds. The symbolic lower and upper bounds are linear constraints $\sum_{j=0}^{i-1} c_j^l \cdot x_j + c_i^l \leq x_i \leq \sum_{j=0}^{i-1} c_j^u \cdot x_j + c_i^u$, where each of $c_i^l, c_i^u$ is a constant. Both bounds are computed in a forward layer-by-layer fashion, using the result of the previous layers to compute bounds for the current layer.

We illustrate the computation in Fig. 2. In the beginning, we have $x_1 \in [-1, 1]$ as the concrete bounds, and $-1 \leq x_1 \leq 1$ as the symbolic bounds, and similarly for $x_2$. To obtain bounds for $x_3, x_4$, we multiply $x_1, x_2$ by the edge weights, which for $x_3$ gives the linear bounds $-x_1 + x_2 \leq x_3 \leq -x_1 + x_2$. Then, to compute $l_3$ and

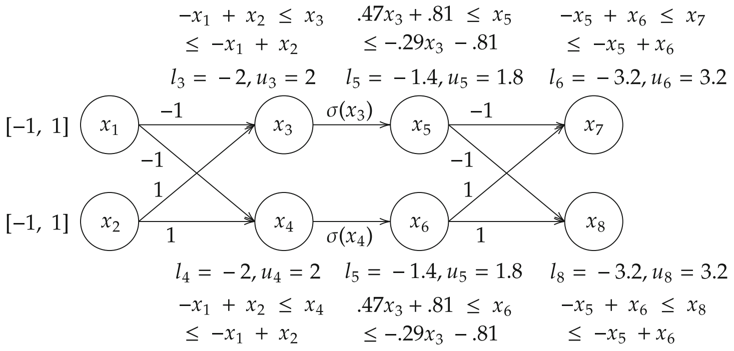

**Fig. 2.** An example of linear bounding for neural network verification.

$u_3$, we minimize and maximize the linear lower and upper bounds, respectively, over $x_1, x_2 \in [-1, 1]$. Doing so results in $l_3 = -2, u_3 = 2$. We obtain the same result for $x_4$.

However, we encounter a key challenge when attempting to bound $x_5$, as we need a linear approximation of $\sigma(x_3)$ over $[l_3, u_3]$ when bounding $x_5$, and similarly for $x_6$. Here, a linear approximation for $x_5$ can be regarded as a set of coefficients $a_l, b_l, a_u, b_u$ such that the following *soundness* condition holds: $\forall x_3 \in [l_3, u_3] \ . \ a_l \cdot x_3 + b_l \le \sigma(x_3) \le a_u \cdot x_3 + b_u$. In addition, a sub goal for the bounds is *tightness*, which typically means the volume between the bounds and $\sigma(x)$ is minimized. Crafting a function to generate these coefficients has been the subject of many prior works. Many seminal papers on neural network verification have focused on solving this problem alone. Broadly speaking, they fall into the following categories.

*Hand-Crafted Approximation Techniques.* The first category of techniques use hand-crafted functions for generating $a_l, b_l, a_u, b_u$. Hand-crafted functions are generally fast because they are static, and tight because an expert designed them. Unfortunately, current works in this category are not *general* – they only considered the most common activation functions, and thus cannot currently handle our motivating example or any recent, novel activation functions. For these works to apply to our motivating example, an expert would need to hand-craft an approximation for the activation function, which is both difficult and error-prone.

*Expensive Solver-Aided Techniques.* The second category use expensive solvers and optimization tools to compute sound and tight bounds in a general way, but at the cost of runtime. Recent works include DiffRNN [25] and POPQORN [19]. The former uses (unsound) optimization to synthesize candidate coefficients and then uses an SMT solver to verify soundness of the bounds. The latter uses constrained-gradient descent to compute coefficients. We note that, while these works do not explicitly target an arbitrary activation function $\sigma(x)$, their techniques can be naturally extended. Their high runtime and computational cost are undesirable and, in general, make them less scalable than the first category.

*Decomposing Based Techniques.* The third category combine hand-crafted approximations with a decomposing based technique to obtain generality and efficiency, but at the cost of tightness. Interestingly, this is similar to the approach used by non-linear SMT solvers and optimizers such as dReal [11] and Ibex [5]. To the best of our knowledge, only one work AutoLiRPA [46] implements this approach for neural network verification. Illustrating on our example, AutoLiRPA does not have a static linear approximation for $\sigma(x_3) = x_3 \cdot sigmoid(x_3)$, but it has static approximations for $sigmoid(x_3)$ and $x_3 \cdot y$. Thus we can bound $sigmoid(x_3)$ over $x_3 \in [-2, 2]$, and then, letting $y = sigmoid(x_3)$, bound $x_3 \cdot y$. Doing so results in the approximation shown as red lines in Fig. 3. While useful, they are suboptimal because they do not minimize the area between the two bounding lines. This suboptimality occurs due to the decomposing, i.e., the static approximations used here were not designed for $swish(x)$ as a whole, but designed for the individual elementary operations.

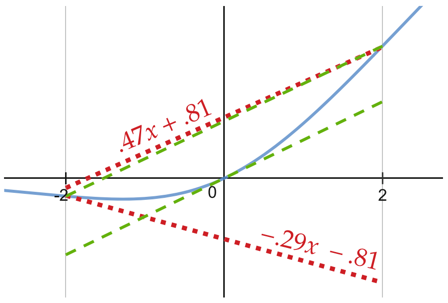

**Fig. 3.** Approximation of AutoLiRPA (red) and our approach (green). (Color figure online)

*Our Work: Synthesizing Static Approximations.* Our work overcomes the limitation of prior work by automatically synthesizing a *static* function specifically for any given activation function $\sigma(x)$ *without* decomposing. Since the synthesis is automatic, and results in a bound generator function, we obtain generality and efficiency, and since the synthesis targets $\sigma(x)$ specifically, we *usually* (demonstrated empirically) obtain tightness. In Fig. 3, for example, the bounds computed by our method are represented by the green lines. The synthesized bound generator function can then be integrated to state-of-the-art neural network verification tools, including AutoLiRPA.

*Wrapping Up the Example.* For our running example, using AutoLiRPA's linear approximation, we would add the linear bounds for $x_5$ shown in Fig. 2. To compute $l_5, u_5$, we would substitute the linear bounds for $x_3$ into $x_5$'s linear bounds, resulting in linear bounds with only $x_1, x_2$ terms that can be minimized/maximized for $l_5, l_6$ respectively. We do the same for $x_6$, and then we repeat the entire process until the output layer is reached.

## 3 Problem Statement and Challenges

In this section, we formally define the synthesis problem and then explain the technical challenges. During the discussion, we focus on synthesizing the generator functions for the upper bound, but in Sect. 3.1, we explain how we can obtain the lower bound functions.

### 3.1 The Synthesis Problem

Given an activation function $\sigma(x)$ and an input universe $x \in [l_x, u_x]$, we define the set of all intervals over $x$ in this universe as $I_x = \{\ [l, u] \mid [l, u] \in \mathbb{IR}, l, u \in [l_x, u_x]\}$. In our experiments, for instance, we use $l_x = -10$ and $u_x = 10$. Note that if we encounter an $[l, u] \notin I_x$, we fall back to a decomposing-based technique.

Our goal is to synthesize a generator function $\mathcal{G}(l, u) \rightarrow \langle a_u, b_u \rangle$, or equivalently, two generator functions $\mathcal{G}_{a_u}(l, u)$ and $\mathcal{G}_{b_u}(l, u)$ such that $\forall [l, u] \in I_x, x \in \mathbb{R}$, the condition $x \in [l, u] \implies \sigma(x) \leq \mathcal{G}_{a_u}(l, u) \cdot x + \mathcal{G}_{b_u}(l, u)$ holds. This is the same as requiring that the following condition does **not** hold (i.e., the formula is unsatisfiable):

$$\exists [l, u] \in I_x, x \in \mathbb{R}\ .\ x \in [l, u] \wedge \sigma(x) > \mathcal{G}_{a_u}(l, u) \cdot x + \mathcal{G}_{b_u}(l, u) \tag{3}$$

The formula above expresses the search for a counterexample, i.e., an input interval $[l, u]$ such that $\mathcal{G}_{a_u}(l, u) \cdot x + \mathcal{G}_{b_u}(l, u)$ is not a sound upper bound of $\sigma(x)$ over the interval $[l, u]$. Thus, if the above formula is unsatisfiable, the soundness of the coefficient functions $\mathcal{G}_{a_u}, \mathcal{G}_{b_u}$ is proved. We note that we can obtain the lower bound generator functions $\mathcal{G}_{a_l}(l, u), \mathcal{G}_{b_l}(l, u)$ by synthesizing upper bound functions $\mathcal{G}_{a_u}(l, u), \mathcal{G}_{b_u}(l, u)$ for $-\sigma(x)$ (i.e. reflecting $\sigma(x)$ across the x-axis), and then letting $\mathcal{G}_{a_l} = -\mathcal{G}_{a_u}(l, u), \mathcal{G}_{b_l} = -\mathcal{G}_{b_u}(l, u)$.

In addition to *soundness*, we want the bound to be *tight*, which in our context has two complementary goals. For a given $[l, u] \in I_x$ we should have (1) $\sigma(z) = \mathcal{G}_{a_u}(l, u) \cdot z + \mathcal{G}_{b_u}(l, u)$ for at least one $z \in [l, u]$ (i.e., the bound touches $\sigma(x)$ at some point $z$), and (2) the volume below $\mathcal{G}_{a_u}(l, u) \cdot x + \mathcal{G}_{b_u}(l, u)$ should be minimized (which we note is equivalent to minimizing the volume between the upper bound and $\sigma(x)$ since $\sigma(x)$ is fixed). We will illustrate the volume by the shaded green region below the dashed bounding line in Fig. 6.

The first goal is intuitive: if the bound does not touch $\sigma(x)$, then it can be shifted downward by some constant. The second goal is a heuristic taken from prior work that has been shown to yield a precise approximation of the neural network's output set.

### 3.2 Challenges and Our Solution

We face three challenges in searching for the generator functions $\mathcal{G}_{a_u}$ and $\mathcal{G}_{b_u}$. First, we must restrict the search space so that a candidate can be found in a reasonable amount of time (i.e., the search is tractable). The second challenge, which is at odds with the first, is that we must have a large enough search space

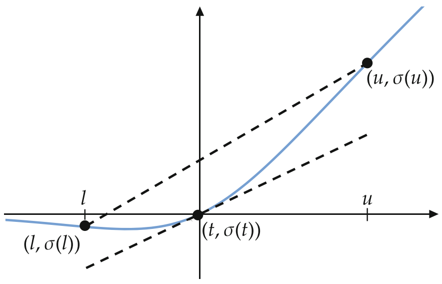

**Fig. 4.** Illustration of the two-point form bound (upper dashed line) and tangent-line form bound (lower dashed line).

such that it permits candidates that represent tight bounds. Finally, the third challenge, which is at odds with the second, is that we must be able to formally verify $\mathcal{G}_{a_u}, \mathcal{G}_{b_u}$ to be sound. While more complex geneator functions ($\mathcal{G}_{a_u}, \mathcal{G}_{b_u}$) will likely produce tighter bounds, they will be more difficult (if not impractical) to verify.

We tackle these challenges by proposing two templates for $\mathcal{G}_{a_u}, \mathcal{G}_{b_u}$ and then developing an approach for selecting the appropriate template. We observe that prior work has always expressed the linear bound for $\sigma(x)$ over an interval $x \in [l, u]$ as either the line connecting the points $(l, \sigma(l)), (u, \sigma(u))$, referred to as the *two-point form*, or as the line tangent to $\sigma(x)$ at a point $t$, referred to as *tangent-line form*. We illustrate both forms in Fig. 4. Assuming that $\sigma'(x)$ is the derivative of $\sigma(x)$, the two templates for $\mathcal{G}_{a_u}$ and $\mathcal{G}_{b_u}$ as follows:

$$\begin{aligned} \mathcal{G}_{a_u}(l, u) &= \frac{\sigma(u) - \sigma(l)}{u - l} \\ \mathcal{G}_{b_u}(l, u) &= -\mathcal{G}_{a_u}(l, u) \cdot l + \sigma(l) + \epsilon \end{aligned} \qquad \text{two-point form template} \tag{4}$$

$$\begin{aligned} \mathcal{G}_{a_u}(l, u) &= \sigma'(g(l, u)) \\ \mathcal{G}_{b_u}(l, u) &= -\mathcal{G}_{a_u}(l, u) \cdot g(l, u) + \sigma(g(l, u)) + \epsilon \end{aligned} \qquad \text{tangent-line form template} \tag{5}$$

In these templates, there are two *holes* to fill during synthesis: $\epsilon$ and $g(l, u)$. Here, $\epsilon$ is a real-valued constant upward (positive) shift that ensures soundness of the linear bounds computed by both templates. We compute $\epsilon$ when we verify the soundness of the template (discussed in Sect. 4.3). In addition to $\epsilon$, for the tangent-line template, we must synthesize a function $g(l, u) = t$, which takes the interval $[l, u]$ as input and returns the tangent point $t$ as output.

These two templates, together, address the previously mentioned three challenges. For the first challenge, the two-point form actually does not have a search space, and thus can be computed efficiently, and for the tangent-line form, we only need to synthesize the function $g(l, u)$. In Sect. 4.2, we will show empirically that $g(l, u)$ tends to be much easier to learn than a function that directly predicts the coefficients $a_u, b_u$. For the second challenge, if the two-point form is sound, then it is also tight since the bound touches $\sigma(x)$ by construction. Similarly, the tangent-line form touches $\sigma(x)$ at $t$. For the third challenge, we will show empirically that these templates can be verified to be sound in a reasonable amount of time (on the order of an hour). prove the soundness of $\mathcal{G}_{a_u}, \mathcal{G}_{b_u}$ for large

At a high level, our approach contains three steps. The first step is to partition $I_x$ into subsets, and then for each subset we assign a fixed template – either the two-point form template or tangent-line form template. The advantage of partitioning is two-fold. First, no single template is a good fit for the entire $I_x$, and thus partitioning results in overall tighter bounds. And second, if the final verified template for a particular subset has a large violation (which results in a large upward shift and thus less tight bounds) the effect is localized to that subset only. Once we have assigned a template to each subset of $I_x$, the second step is to learn a $g(l, u)$ for each subset that was assigned a tangent-line template. We use an example-generation procedure to generate training examples, which are then used to train a machine learning model. After learning each $g(l, u)$, the third step is to compute $\epsilon$ for all of the templates. We phrase the search for a sound $\epsilon$ as a nonlinear global optimization problem, and then use the interval-based solver IbexOpt [5] to bound $\epsilon$.

## 4 Our Approach

In this section, we first present our method for partitioning $I_x$, the input interval space, into disjoint subsets and then assigning a template to each subset. Then, we present the method for synthesizing the bounds-generating function for a subset in the partition of $I_x$ (see Sect. 3.1). Next, we present the method for making the bounds-generating functions sound. Finally, we present the method for efficiently looking up the appropriate template at runtime.

### 4.1 Partitioning the Input Interval Space ($\boldsymbol{I_x}$)

A key consideration when partitioning $I_x$ is how to represent each disjoint subset of input intervals. While we could use a highly expressive representation such as polytope or even use non-linear constraints, for efficiency reasons, we represent each subset (of input intervals) as a box. Since a subset uses either the two-point form template or the tangent-line form template, the input interval space can be divided into $I_x = I_{2pt} \cup I_{tan}$. Each of $I_{2pt}$ and $I_{tan}$ is a set of boxes.

At a high-level, our approach first partitions $I_x$ into uniformly sized disjoint boxes, and then assigns each box to either $I_{2pt}$ or $I_{tan}$. In Fig. 5, we illustrate the partition computed for $swish(x) = x \cdot sigmoid(x)$. The $x$-axis and $y$-axis represent the lower bound $l$ and the upper bound $u$, respectively, and thus a point $(l, u)$ on this graph represents the interval $[l, u]$, and a box on this graph denotes the set of intervals represented by the points contained within it. We give details on computing the partition below.

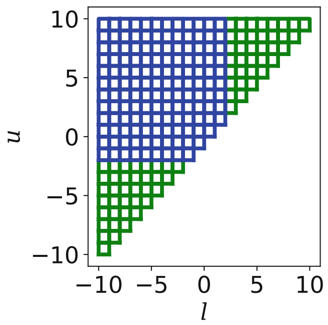

**Fig. 5.** Partition of $I_x$ for the Swish activation function, where the blue boxes belong to $I_{tan}$, and the green boxes belong to $I_{2pt}$. (Color figure online)

*Defining the Boxes.* We first define a constant parameter $c_s$, which is the width and height of each box in the partition of $I_x$. In Fig. 5, $c_s = 1$. The benefits of using a smaller $c_s$ value is two-fold. First, it allows us to more accurately choose the proper template (two-point or tangent) for a given interval $[l, u]$. Second, as mentioned previously, the negative impact of a template with a large violation (i.e., large $\epsilon$) is localized to a smaller set of input intervals.

Assuming that $(u_x - l_x)$ can be divided by $c_s$, then we have $(\frac{u_x - l_x}{c_s})^2$ disjoint boxes in the partition of $I_x$, which we represent by $I_{i,j}$ where $i, j \in \{1..\frac{u_x - l_x}{c_s}\}$. $I_{i,j}$ represents the box whose lower-left corner is located at $(l_x + i \cdot c_s, l_x + j \cdot c_s)$, or alternatively we have $I_{i,j} = \{[l, u] \mid l \in [l_x + i \cdot c_s, l_x + i \cdot c_s + c_s], u \in [l_x + j \cdot c_s, l_x + j \cdot c_s + c_s]\}$.

To determine which boxes $I_{i,j}$ belong to the subset $I_{2pt}$, we uniformly sample intervals $[l, u] \in I_{i,j}$. Then, for each sampled interval $[l, u]$, we compute the two-point form for $[l, u]$, and attempt to search for a counter-example to the equation $\sigma(x) \leq \mathcal{G}_{a_u}(l, u)x + \mathcal{G}_{b_u}(l, u)$ by sampling $x \in [l, u]$. If a counter-example is not found for more than half of the sampled $[l, u] \in I_{i,j}$, we add the box $I_{i,j}$ to $I_{2pt}$, otherwise we add the box to $I_{tan}$.

We note that more sophisticated (probably more expensive) strategies for assigning templates exist. We use this strategy simply because it is efficient. We also note that some boxes in the partition may contain invalid intervals (i.e., we have $[l, u] \in I_{i,j}$ where $u < l$). These invalid intervals are filtered out during the final verification step described in Sect. 4.3, and thus do not affect the soundness of our algorithm.

### 4.2 Learning the Function $g(l, u)$

In this step, for each box $I_{i,j} \in I_{tan}$, we want to learn a function $g(l, u) = t$ that returns the tangent point for any given interval $[l, u] \in I_{i,j}$, where $t$ will be used to compute the tangent-line form upper bound as defined in Eq. 5. This process is done for all boxes in $I_{tan}$, resulting in a separate $g(l, u)$ for each box $I_{i,j}$. A sub-goal when learning $g(l, u)$ is to maximize the tightness of the resulting upper bound, which in our case means minimizing the volume below the tangent line.

We leverage machine learning techniques (specifically linear regression or a small neural network with ReLU activation) to learn $g(l, u)$, which means we need a procedure to generate training examples. The examples must have the form $((l, u), t)$. To generate the training examples, we (uniformly) sample $[l, u] \in I_{i,j}$, and for each sampled $[l, u]$, we attempt to find a tangent point $t$ whose tangent line represents a tight upper bound of $\sigma(x)$. Then, given the training examples, we use standard machine learning techniques to learn $g(l, u)$.

The crux of our approach is generating the training examples. To generate a single example for a fixed $[l, u]$, we follows two steps: (1) generate upper bound coefficients $a_u, b_u$, and then (2) find a tangent point $t$ whose tangent line is close to $a_u, b_u$. In the following paragraphs, we describe the process for a fixed $[l, u]$, and then discuss the machine learning procedure.

**Generating Example Coefficients $\boldsymbol{a_u}, \boldsymbol{b_u}$.** Given a fixed $[l, u]$, we aim to generate upper bound coefficients $a_u, b_u$. A good generation procedure has three criteria: (1) the coefficients should be tight for the input interval $[l, u]$, (2) the coefficients should be sound, and (3) the generation should be fast. The first two criteria are intuitive: good training examples will result in a good learned model. The third is to ensure that we can generate a large number of examples in a reasonable amount of time. Unfortunately, the second and third criteria are at odds, because proving soundness is inherently expensive. To ensure a reasonable runtime, we relax the second criteria to *probably* sound. Thus our final goal is to minimize volume below $a_u, b_u$ such that $\sigma(x) \le a_u \cdot x + b_u$ *probably* holds for $x \in [l, u]$.

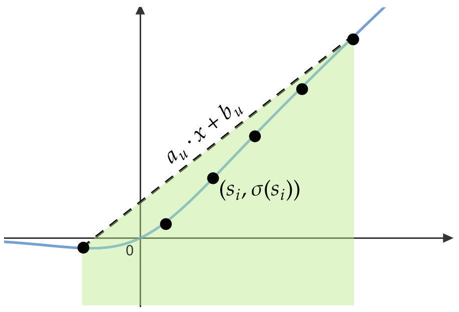

**Fig. 6.** Illustration of the sampling and linear programming procedure for computing an upper bound. Shaded green region illustrates the volume below the upper bound. (Color figure online)

Our approach is inspired by a prior work [2,33], which formulates the goal of a non-linear optimization problem as a linear program that can be solved efficiently. Our approach samples points $(s_i, \sigma(s_i))$ on the activation function for $s_i \in [l, u]$, which are used to to convert the nonlinear constraint $\sigma(x) \le a_u \cdot x + b_u$ into a linear one, and then uses volume as the objective (which is linear). For a set $S$ of sample points $s_i \in [l, u]$, the linear program we solve is:

$$\text{minimize : volume below } \; a_u \cdot x + b_u$$
$$\text{subj. to : } \bigwedge_{s_i \in S} \sigma(s_i) \le a_u \cdot s_i + b_u$$

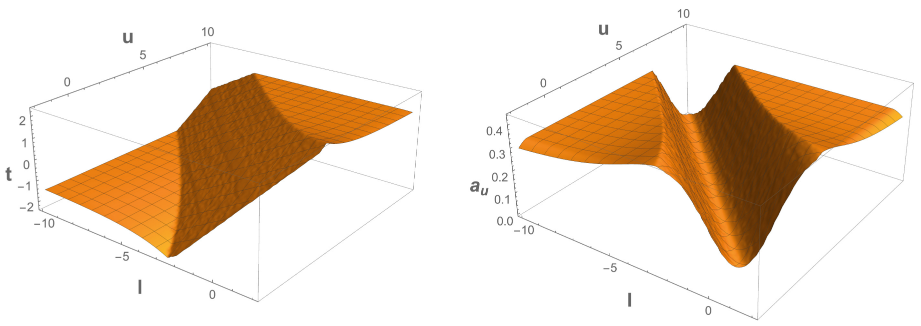

**Fig. 7.** Plots of the training examples, smoothed with linear interpolation. On the left: a plot of $((l, u), (t))$, and on the right: a plot of $((l, u), (a_u))$.

We illustrate this in Fig. 6. Solving the above problem results in $a_u, b_u$, and the prior work [2,33] proved that the solution (theoretically) approaches the optimal and sound $a_u, b_u$ as the number of samples goes to infinity. We use Gurobi [13] to solve the linear program.

**Converting $\boldsymbol{a_u, b_u}$ to a Tangent Line.** To use the generated $a_u, b_u$ in the tangent-line form template, we must find a point $t$ whose tangent line is close to $a_u, b_u$. That is, we require that the following condition (almost) holds:

$$(\sigma'(t) = a_u) \wedge (-\sigma'(t) \cdot t + \sigma(t) = b_u)$$

To solve the above problem, we use local optimization techniques (specifically a modified Powell's method [29] implemented in SciPy [41], but most common techniques would work) to find a solution to $\sigma'(t) = a_u$.

We then check that the right side of the above formula almost holds (specifically, we check $(|(\sigma'(t) \cdot t + \sigma(t)) - b_u| \leq 0.01)$. If the local optimization does not converge (i.e., it does not find a $t$ such that $\sigma'(t) = a_u$), or the check on $b_u$ fails, we throw away the example and do not use it in training.

One may ask the question: could we simply train a model to directly predict the coefficients $a_u$ and $b_u$, instead of predicting a tangent point and then converting it to the tangent line? The answer is yes, however this approach has two caveats. The first caveat is that we will lose the inherent tightness that we gain with the tangent-line form – we no longer have a guarantee that the computed linear bound will touch $\sigma(x)$ at any point. The second caveat is that the relationship between $l, u$ and $t$ tends to be close to linear, and thus easier to learn, whereas the relationship between $l, u$ and $a_u$, or between $l, u$ and $b_u$, is highly nonlinear. We illustrate these relationships as plots in Fig. 7. The left graph plots the generated training examples $((l, u), t)$, converted to a smooth function using linear interpolation. We can see most regions are linear, as shown by the flat sections. The right plot shows $((l, u), a_u)$, where we can see the center region is highly nonlinear.

**Training on the Examples.** Using the procedure presented so far, we sample $[l, u]$ uniformly from $I_{i,j}$ and generate the corresponding $t$ for each of them. This results in a training dataset of $r$ examples $\mathcal{D}_{train} = \{((l_i, u_i), t_i) \mid i \in \{1..r\}\}$. We then choose between one of two models – a linear regression model or a 2-layer, 50-hidden-neuron, ReLU network – to become the final function $g(l, u)$. To decide, we train both model types, and choose the one with the lowest error, where error is measured as the mean absolute error. We give details below.

A linear regression model is a function $g(l, u) = c_1 \cdot l + c_2 \cdot u + c_3$, where $c_i \in \mathbb{R}$ are coefficients learned by minimizing the *squared error*, which formally is:

$$\sum_{((l_i,u_i),t_i)\in D_{train}} (g(l_i, u_i) - t_i)^2 \tag{6}$$

Finding the coefficients $c_i$ that minimize the above constraint has a closed-form solution, thus convergence is guaranteed and optimal, which is desirable.

However, sometimes the relationship between $(l, u)$ and $t$ is nonlinear, and thus using a linear regression model may result in a poor-performing $g(l, u)$, even though the solution is optimal. To capture more complex relationships, we also consider a 2-layer ReLU network where $\mathbf{W}_0 \in \mathbb{R}^{2\times 50}$, $\mathbf{W}_1 \in \mathbb{R}^{50\times 1}$, $\mathbf{b}_0 \in \mathbb{R}^{50}$, $\mathbf{b}_1 \in \mathbb{R}$, and we have $g(l, u) = \text{ReLU}(\langle l, u\rangle^T \cdot \mathbf{W}_0 + \mathbf{b}_0) \cdot \mathbf{W}_1 + \mathbf{b}_1$. The weights and biases are initialized randomly, and then we minimize the squared error (Eq. 6) using gradient descent. While convergence to the optimal weights is not guaranteed in theory, we find in practice it usually converges.

We choose these two models because they can capture a diverse set of $g(l, u)$ functions. While we could use other prediction models, such as polynomial regression, generally, a neural network will be equally (if not more) expressive. However, we believe exploring other model types or architectures of neural networks would be an interesting direction to explore.

## 4.3 Ensuring Soundness of the Linear Approximations

For a given $I_{i,j}$, we must ensure that its corresponding coefficient generator functions $\mathcal{G}_{a_u}(l, u)$ and $\mathcal{G}_{b_u}(l, u)$ are sound, or in other words, that the following condition does **not** hold:

$$\exists [l, u] \in I_{i,j},\ x \in [l, u]\ .\ \sigma(x) > \mathcal{G}_{a_u}(l, u) \cdot x + \mathcal{G}_{b_u}(l, u)$$

We ensure the above condition does not hold (the formula is unsatisfiable) by bounding the *maximum violation* on the clause $\sigma(x) > \mathcal{G}_{a_u}(l, u) \cdot x + \mathcal{G}_{b_u}(l, u)$, which we formally define as $\Delta(l, u, x) = \sigma(x) - (\mathcal{G}_{a_u}(l, u) \cdot x + \mathcal{G}_{b_u}(l, u))$. $\Delta$ is positive when the previous clause holds. Thus, if we can compute an upper bound $\Delta_u$, we can set the $\epsilon$ term in $\mathcal{G}_{b_u}(l, u)$ to $\Delta_u$ to ensure the clause does not hold, thus making the coefficient generator functions sound.

To compute $\Delta_u$, we solve (i.e., bound) the following optimization problem:

$$\begin{aligned} \text{for}: &\ l, u, x \in [l_{i,j}, u_{i,j}] \\ \text{maximize}: &\ \Delta(l, u, x) \\ \text{subj. to}: &\ l < u \wedge l \leq x \wedge x \leq u \end{aligned}$$

where $l_{i,j}, u_{i,j}$ are the minimum lower bound and maximum upper bound, respectively, for any interval in $I_{i,j}$. The above problem can be solved using the general framework of interval analysis [26] and branch-and-prune algorithms [4].

Letting $\Delta_{search} = \{(l, u, x) | l, u, x \in [l_{i,j}, u_{i,j}]\}$ be the domain over which we want to bound $\Delta$, we can bound $\Delta$ over $\Delta_{search}$ using interval analysis. In addition, we can improve the bound in two ways: *branching* (i.e., partitioning $\Delta_{search}$ and bounding $\Delta$ on each subset separately) and *pruning* (i.e., removing from $\Delta_{search}$ values that violate the constraints $l < u \wedge l \leq x \wedge x \leq u$). The tool IbexOpt [5] implements such an algorithm, and we use it solve the above optimization problem.

One practical consideration when solving the above optimization problem is the presence of division by zero error. In the two-point template, we have $\mathcal{G}_{a_u}(l, u) = \frac{\sigma(u)-\sigma(l)}{u-l}$. While we have the constraint $l < u$, from an interval analysis perspective, $\mathcal{G}_{a_u}(l, u)$ goes to infinity as $u - l$ goes to 0, and indeed, if we gave the above problem to IbexOpt, it would report that $\Delta$ is unbounded. To account for this, we enforce a minimum interval width of 0.01 by changing $l < u$ to $0.01 < u - l$.

### 4.4 Efficient Lookup of the Linear Bounds

Due to partitioning $I_x$, we must have a procedure for looking up the appropriate template instance for a given $[l, u]$ at the application time. Formally, we need to find the box $I_{i,j}$, which we denote $[l_l, u_l] \times [l_u, u_u]$, such that $l \in [l_l, u_l]$ and $u \in [l_u, u_u]$, and retrieve the corresponding template. Lookup can actually present a significant runtime overhead if not done with care. One approach is to use a data structure similar to an interval tree or a quadtree [10], the latter of which has $\mathcal{O}(log(n))$ complexity. While the quadtree would be the most efficient for an arbitrary partition of $I_x$ into boxes, we can in fact obtain $\mathcal{O}(1)$ lookup for our partition strategy.

We first note that each box, $I_{i,j}$, can be uniquely identified by $l_l$ and $u_u$. The point $(l_l, u_u)$ corresponds to the top-left corner of a box in Fig. 5. Thus we build a lookup dictionary keyed by $(l_l, u_u)$ for each box that maps to the corresponding linear bound template. To perform lookup, we exploit the structure of the partition: specifically, each box in the partition is aligned to a multiple of $c_s$. Thus, to lookup $I_{i,j}$ for a given $[l, u]$, we view $(l, u)$ as a point on the graph of Fig. 5, and the lookup corresponds to moving left-ward and upward from the point $(l, u)$ to the nearest upper-left corner of a box. More formally, we perform lookup by rounding $l$ down to the nearest multiple of $c_s$, and $u$ upward to the nearest multiple of $c_s$. The top-left corner can then be used to lookup the appropriate template.

## 5 Evaluation

We have implemented our approach as a software tool that synthesizes a linear bound generator function $\mathcal{G}(l, u)$ for any given activation function $\sigma(x)$ in the input universe $x \in [l_x, u_x]$. The output is a function that takes as input $[l, u]$ and returns coefficients $a_l, b_l, a_u, b_u$ as output. For all experiments, we use $l_x = -10, u_x = 10$, $c_s = 0.25$, and a minimum interval width of 0.01. If we encounter an $[l, u] \not\subseteq [l_x, u_x]$, we fall back to the interval bound propagation of dReal [11]. After the generator function is synthesized, we integrate it into AutoLiRPA, a state-of-the-art neural network verification tool, which allows us to analyze neural networks with $\sigma(x)$ as activation functions.

## 5.1 Benchmarks

**Neural Networks and Datasets.** Our benchmarks are eight deep neural networks trained on the following two datasets.

*MNIST.* MNIST [22] is a set of images of hand-written digits each of which are labeled with the corresponding written digit. The images are $28 \times 28$ grayscale images with one of ten written digits. We use a convolutional network architecture with 1568, 784, and 256 neurons in its first, second, and third layer, respectively. We train a model for each of the activation functions described below.

*CIFAR.* CIFAR [20] is a set of images depicting one of 10 objects (a dog, a truck, etc.), which are hand labeled with the corresponding object. The images are $32 \times 32$ pixel RGB images. We use a convolutional architecture with 2048, 2048, 1024, and 256 neurons in the first, second, third, and fourth layers, respectively. We train a model for each of the activation functions described below.

**Activation Functions.** Our neural networks use one of the activation functions shown Fig. 8 and defined in Table 1. They are Swish [14,31], GELU [14], Mish [24], LiSHT [32], and AtanSq [31]. The first two are used in language models such as GPT [30], and have been shown to achieve the best performance for some image classification tasks [31]. The third and fourth two are variants of the first two, which are shown to have desirable theoretical properties. The last was discovered using automatic search techniques [31], and found to perform on par with the state-of-the-art. We chose these activations because they are representative of recent developments in deep learning research.

**Robustness Verification.** We evaluate our approach on *robustness* verification problems. Given a neural network $f : \mathbb{X} \subseteq \mathbb{R}^n \to \mathbb{Y} \subseteq \mathbb{R}^m$ and an input $\mathbf{x} \in \mathbb{X}$, we verify robustness by proving that making a small $p$-bounded perturbation ($p \in \mathbb{R}$) to $\mathbf{x}$ does not change the classification. Letting $\mathbf{x}[i] \in \mathbb{R}$ be the $i^{th}$ element in $\mathbf{x}$, we represent the set of all perturbations as $X \in \mathbb{IR}^n$, where $X = \bigtimes_{i=1}^{n} [\mathbf{x}[i] - p, \mathbf{x}[i] + p]$. We then compute $Y \in \mathbb{IR}^m$ where $Y = \bigtimes_{i=1}^{m} [l_i, u_i]$, and, assuming the target class of $\mathbf{x}$ is $j$, where $j \in \{1..m\}$, we prove robustness by checking $(l_j > u_i)$ for all $i \neq j$ and $i \in \{1..m\}$.

**Table 1.** Definitions of activation functions used in our experiments.

| Name | Definition |
|---|---|
| Swish | $x \cdot sigmoid(x)$ |
| GELU | $0.5x(1 + \tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$ |
| Mish | $x \cdot \tanh[\ln(1 + e^x)]$ |
| LiSHT | $x \cdot \tanh(x)$ |
| AtanSq | $(\tan^{-1}(x))^2 - x$ |

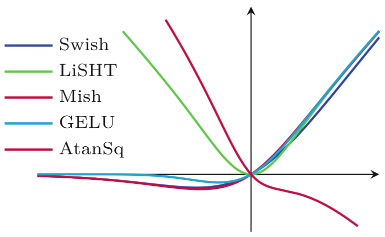

**Fig. 8.** Activation functions used in our experiments.

For each network, we take 100 random test images, and following prior work [12], we filter out misclassified images. We then take the remaining images, and create a robustness verification problem for each one. Again following prior work, we use $p = 8/255$ for MNIST networks and $p = 1/255$ for CIFAR networks.

## 5.2 Experimental Results

Our experiments were designed to answer the following question: How do our synthesized linear approximations compare with other state-of-the-art, hand-crafted linear approximation techniques on novel activation functions? To the best of our knowledge, AutoLiRPA [46] is the only neural network verification tool capable of handling the activation functions we considered here using static, hand-crafted approximations. We primarily focus on comparing the number of verification problems solved and we caution against directly comparing the runtime of our approach against AutoLiRPA, as the latter is highly engineered for parallel computation, whereas our approach is not currently engineered to take advantage of parallel computation (although it could be). We conducted all experiments on an 8-core 2.7 GHz processor with 32 GB of RAM.

We present results on robustness verification problems in Table 2. The first column shows the dataset and architecture. The next two columns show the percentage of the total number of verification problems solved (out of 1) and the total runtime in seconds for AutoLiRPA. The next two columns show the same statistics for our approach. The final column compares the output set sizes of AutoLiRPA and our approach. We first define $|Y|$ as the volume of the (hyper)box $Y$. Then letting $Y_{auto}$ and $Y_{ours}$ be the output set computed by AutoLiRPA and our approach, respectively, $\frac{|Y_{ours}|}{|Y_{auto}|}$ measures the reduction in output set size. In general, $|Y_{ours}| < |Y_{auto}|$ indicates our approach is better because it implies that our approach has more accurately approximated the true output set, and thus $\frac{|Y_{ours}|}{|Y_{auto}|} < 1$ indicates our approach is more accurate.

We point out three trends in the results. First, our automatically synthesized linear approximations always result in more verification problems solved. This is because our approach synthesizes a linear approximation specifically for $\sigma(x)$, which results in tighter bounds. Second, AutoLiRPA takes longer on more complex activations such as GELU and Mish, which have more elementary

**Table 2.** Comparison of the verification results of our approach and AutoLiRPA.

| Network Architecture | | AutoLiPRA [46] | | Our Approach | | $\frac{\lvert Y_{ours}\rvert}{\lvert Y_{auto}\rvert}$ |
|---|---|---|---|---|---|---|
| | | % certified | time (s) | % certified | time (s) | |
| MNIST | 4-Layer CNN with Swish | 0.34 | 15 | 0.74 | 195 | 0.59 |
| | 4-Layer CNN with Gelu | 0.01 | 359 | 0.70 | 289 | 0.22 |
| | 4-Layer CNN with Mish | 0.00 | 50 | 0.28 | 236 | 0.29 |
| | 4-Layer CNN with LiSHT | 0.00 | 15 | 0.11 | 289 | 0.32 |
| | 4-Layer CNN with AtanSq$^1$ | - | - | 0.16 | 233 | - |
| CIFAR | 5-Layer CNN with Swish | 0.03 | 69 | 0.35 | 300 | 0.42 |
| | 5-Layer CNN with Gelu | 0.00 | 1,217 | 0.29 | 419 | 0.21 |
| | 5-Layer CNN with Mish | 0.00 | 202 | 0.29 | 363 | 0.17 |
| | 5-Layer CNN with LiSHT | 0.00 | 68 | 0.00 | 303 | 0.09 |
| | 5-Layer CNN with AtanSq$^1$ | - | - | 0.22 | 347 | - |

$^1$AutoLiRPA does not have an approximation for $\tan^{-1}$.

operations than Swish and LiSHT. This occurs because AutoLiRPA has more linear approximations to compute (it must compute one for every elementary operation before composing the results together). On the other hand, our approach computes the linear approximation in one step, and thus does not have the additional overhead for the more complex activation functions. Third, our approach always computes a much smaller output set, in the range of 2-10X smaller, which again is a reflection of the tighter linear bounds.

*Synthesis Results.* We also report some key metrics about the synthesis procedure. Results are shown in Table 3. The first three columns show the total CPU time for the three steps in our synthesis procedure. We note that all three steps can be heavily parallelized, thus the wall clock time is roughly 1/8 the reported times on our 8-core machine. The final column shows the percentage of boxes in the partition that were assigned a two-point template (we can take the complement to get the percentage of tangent-line templates).

## 6 Related Work

Most closely related to our work are those that leverage interval-bounding techniques to conduct neural network verification. Seminal works in this area can either be thought of as explicit linear bounding, or linear bounding with some type of restriction (usually for efficiency). Among the explicit linear bounding techniques are the ones used in DeepPoly [35], AutoLiRPA [46], Neurify [42], and similar tools [2,7,19,33,34,44,45,47]. On the other hand, techniques using Zonotopes [12,23] and symbolic intervals [43] can be thought of as restricted linear bounding. Such approaches have an advantage in scalability, although they may sacrifice completeness and accuracy. In addition, recent

**Table 3.** Statistics of the synthesis step in our method.

| Activation $\sigma(x)$ | Partition Time (s) | Learning Time (s) | Verification Time (s) | $\frac{\lvert I_{2pt}\rvert}{\lvert I_x\rvert}$ |
|---|---|---|---|---|
| Swish | 81 | 1,762 | 20,815 | 0.45 |
| GELU | 104 | 2,113 | 45,504 | 0.57 |
| Mish | 96 | 2,052 | 38,156 | 0.45 |
| LiSHT | 83 | 1,650 | 61,910 | 0.46 |
| AtanSq | 85 | 1,701 | 18,251 | 0.38 |

work leverages semi-definite approximations [15], which allow for more expressive, nonlinear lower and upper bounds. In addition, linear approximations are used in nonlinear programming and optimization [5,40]. However, to the best of our knowledge, none of these prior works attempt to automate the process of crafting the bound generator function $\mathcal{G}(l, u)$.

Less closely related are neural network verification approaches based on solving systems of linear constraints [3,8,16,18,38]. Such approaches typically only apply to networks with piece-wise-linear activations such as ReLU and max pooling, for which there is little need to automate any part of the verification algorithm's design (at least with respect to the activation functions). They do not handle novel activation functions such as the ones concerned in our work. These approaches have the advantage of being complete, although they tend to be less scalable than interval analysis based approaches.

Finally, we note that there are many works built off the initial linear approximation approaches, thus highlighting the importance of designing tight and sound linear approximations in general [36,39,42].

## 7 Conclusions

We have presented the first method for statically synthesizing a function that can generate tight and sound linear approximations for neural network activation functions. Our approach is example-guided, in that we first generate example linear approximations, and then use these approximations to train a prediction model for linear approximations at run time. We leverage nonlinear global optimization techniques to ensure the soundness of the synthesized approximations. Our evaluation on popular neural network verification tasks shows that our approach significantly outperforms state-of-the-art verification tools.

## References

1. Alzantot, M., Sharma, Y., Elgohary, A., Ho, B.J., Srivastava, M., Chang, K.W.: Generating natural language adversarial examples. arXiv:1804.07998 (2018)

2. Balunović, M., Baader, M., Singh, G., Gehr, T., Vechev, M.: Certifying geometric robustness of neural networks. NIPS (2019)
3. Baluta, T., Shen, S., Shinde, S., Meel, K.S., Saxena, P.: Quantitative verification of neural networks and its security applications. In: CCS (2019)
4. Benhamou, F., Granvilliers, L.: Continuous and interval constraints. Foundations of Artificial Intelligence (2006)
5. Chabert, G., Jaulin, L.: Contractor programming. Artificial Intelligence 173(11) (2009)
6. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: TACAS (2008)
7. Du, T., et al.: Cert-RNN: towards certifying the robustness of recurrent neural networks (2021)
8. Ehlers, R.: Formal verification of piece-wise linear feed-forward neural networks. In: ATVA (2017)
9. Engstrom, L., Tran, B., Tsipras, D., Schmidt, L., Madry, A.: Exploring the landscape of spatial robustness. In: ICML (2019)
10. Finkel, R.A., Bentley, J.L.: Quad trees a data structure for retrieval on composite keys. Acta informatica (1974)
11. Gao, S., Kong, S., Clarke, E.M.: dReal: An SMT solver for nonlinear theories over the reals. In: International Conference on Automated Deduction (2013)
12. 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)
13. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2021). https://www.gurobi.com
14. Hendrycks, D., Gimpel, K.: Gaussian error linear units (gelus). arXiv:1606.08415 (2016)
15. 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: CDC (2020)
16. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: CAV (2017)
17. Kanbak, C., Moosavi-Dezfooli, S.M., Frossard, P.: Geometric robustness of deep networks: analysis and improvement. In: CVPR (2018)
18. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: CAV (2017)
19. Ko, C.Y., Lyu, Z., Weng, L., Daniel, L., Wong, N., Lin, D.: POPQORN: quantifying robustness of recurrent neural networks. In: ICML (2019)
20. Krizhevsky, A., Hinton, G., et al.: Learning multiple layers of features from tiny images (2009)
21. Lebbah, Y., Michel, C., Rueher, M.: An efficient and safe framework for solving optimization problems. J. Comput. Appl. Math. (2007)
22. Lecun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE (1998)
23. Mirman, M., Gehr, T., Vechev, M.T.: Differentiable abstract interpretation for provably robust neural networks. In: ICML (2018)
24. Misra, D.: Mish: a self regularized non-monotonic neural activation function. arXiv:1908.08681 (2019)
25. Mohammadinejad, S., Paulsen, B., Deshmukh, J.V., Wang, C.: DiffRNN: Differential verification of recurrent neural networks. In: FORMATS (2021)
26. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to interval analysis. SIAM (2009)

27. Paulsen, B., Wang, J., Wang, C.: Reludiff: differential verification of deep neural networks. In: ICSE (2020)
28. Paulsen, B., Wang, J., Wang, J., Wang, C.: NeuroDiff: scalable differential verification of neural networks using fine-grained approximation. In: ASE (2020)
29. Powell, M.J.: An efficient method for finding the minimum of a function of several variables without calculating derivatives. The Computer Journal (1964)
30. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I.: Improving language understanding by generative pre-training (2018)
31. Ramachandran, P., Zoph, B., Le, Q.V.: Searching for activation functions. arXiv:1710.05941 (2017)
32. Roy, S.K., Manna, S., Dubey, S.R., Chaudhuri, B.B.: LiSHT: Non-parametric linearly scaled hyperbolic tangent activation function for neural networks. arXiv:1901.05894 (2019)
33. Ryou, W., Chen, J., Balunovic, M., Singh, G., Dan, A., Vechev, M.: Scalable polyhedral verification of recurrent neural networks. In: CAV (2021)
34. Shi, Z., Zhang, H., Chang, K.W., Huang, M., Hsieh, C.J.: Robustness verification for transformers. ICLR (2020)
35. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: An abstract domain for certifying neural networks. POPL (2019)
36. Singh, G., Gehr, T., Püschel, M., Vechev, M.T.: Boosting robustness certification of neural networks. In: ICLR (2019)
37. Szegedy, C., et al.: Intriguing properties of neural networks. arXiv:1312.6199 (2013)
38. Tjeng, V., Xiao, K., Tedrake, R.: Evaluating robustness of neural networks with mixed integer programming. ICLR (2019)
39. Tran, H.D., et al.: Star-based reachability analysis of deep neural networks. In: FM (2019)
40. Trombettoni, G., Araya, I., Neveu, B., Chabert, G.: Inner regions and interval linearizations for global optimization. In: AAAI (2011)
41. Virtanen, P.: SciPy 1.0 Contributors: SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods (2020)
42. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: NIPS (2018)
43. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: USENIX Security (2018)
44. Weng, T., et al.: Towards fast computation of certified robustness for relu networks. In: ICML (2018)
45. Wu, Y., Zhang, M.: Tightening robustness verification of convolutional neural networks with fine-grained linear approximation. In: AAAI (2021)
46. Xu, K., et al.: Automatic perturbation analysis for scalable certified robustness and beyond. In: NIPS (2020)
47. Zhang, H., Weng, T.W., Chen, P.Y., Hsieh, C.J., Daniel, L.: Efficient neural network robustness certification with general activation functions. In: NIPS (2018)