On Optimizing Back-Substitution Methods for Neural Network Verification

Tom Zelazny*, Haoze Wu†, Clark Barrett†, and Guy Katz*
*The Hebrew University of Jerusalem, Jerusalem, Israel †Stanford University, Stanford, California
*{tomz, g.katz}@mail.huji.ac.il †{haozewu, barrett}@cs.stanford.edu

***Abstract*—With the increasing application of deep learning in mission-critical systems, there is a growing need to obtain formal guarantees about the behaviors of neural networks. Indeed, many approaches for verifying neural networks have been recently proposed, but these generally struggle with limited scalability or insufficient accuracy. A key component in many state-of-the-art verification schemes is computing lower and upper bounds on the values that neurons in the network can obtain for a specific input domain — and the tighter these bounds, the more likely the verification is to succeed. Many common algorithms for computing these bounds are variations of the symbolic-bound propagation method; and among these, approaches that utilize a process called back-substitution are particularly successful. In this paper, we present an approach for making back-substitution produce tighter bounds. To achieve this, we formulate and then minimize the imprecision errors incurred during back-substitution. Our technique is general, in the sense that it can be integrated into numerous existing symbolic-bound propagation techniques, with only minor modifications. We implement our approach as a proof-of-concept tool, and present favorable results compared to state-of-the-art verifiers that perform back-substitution.**

I. Introduction

Deep neural networks (DNNs) are dramatically changing the way modern software is written. In many domains, such as image recognition [43], game playing [42], protein folding [2] and autonomous vehicle control [12], [30], state-of-the-art solutions involve deep neural networks — which are artifacts learned automatically from a finite set of examples, and which often outperform carefully handcrafted software.

Along with their impressive success, DNNs present a significant new challenge when it comes to quality assurance. Whereas many best practices exist for writing, testing, verifying and maintaining hand-crafted code, DNNs are automatically generated, and are mostly opaque to humans [24], [25]. Consequently, it is difficult for human engineers to reason about them and ensure their correctness and safety — as most existing approaches are ill-suited for this task. This challenge is becoming a significant concern, with various faults being observed in modern DNNs [5]. One notable example is that of *adversarial perturbations* — small perturbation that, when added to inputs that are correctly classified by the DNN, result in severe errors [20], [48]. This issue, and others, call into question the safety, security and interpretability of DNNs, and could hinder their adoption by various stakeholders.

In order to mitigate this challenge, the formal methods community has taken up interest in DNN verification. In the past few years, a plethora of approaches have been proposed for tackling the *DNN verification problem*, in which we are given a DNN and a condition abouts its inputs and outputs; and seek to either find an input assignment to the DNN that satisfies this condition, or prove that it is not satisfiable [1], [8], [10], [14], [21], [27], [29], [31], [33], [39], [51], [57]. The usefulness of DNN verification has been demonstrated in several settings and domains [21], [27], [31], [47], but most existing approaches still struggle with various limitations, specifically relating to scalability.

A key technical challenge in verifying neural networks is to reason about *activation functions*, which are non-linear (e.g., piece-wise linear) transformations applied to the output of each layer in the neural network. Precisely reasoning about such non-linear behaviors requires a case-by-case analysis of the activation phase of each activation function, which quickly becomes infeasible as the number of non-linear activations increases. Instead, before performing such a search procedure, state-of-the-art solvers typically first consider linear abstractions of activation functions, and use these abstractions to over-approximate the values that the activation functions can take in the neural network. Often, these over-approximations significantly curtail the search space that later needs to be explored, and expedite the verification procedure as a whole.

A key operation that is repeatedly invoked in this computation of over-approximations is called *back-substitution* [45], where the goal is to compute, for each neuron in the DNN, lower and upper bounds on the values it can take with respect to the input region of interest. This is done by first expressing the lower and upper bounds of a neuron symbolically as a function of neurons from previous layers, and then concretizing these symbolic bounds with the known bounds of neurons in those previous layers. Such a technique is essential in state-of-the-art solvers (e.g., [32], [45], [54]) and is often able to obtain sufficiently tight bounds for proving the properties with respect to small input regions. However, it tends to significantly lose precision when the input region (i.e., perturbation radius) grows, preventing one from efficiently verifying more challenging problems.

In this work, we seek to improve the precision and scalability of DNN verification techniques, by reducing the over-approximation error in the back-substitution process. Our key insight is that, as part of the symbolic-bound propagation, one can measure the error accumulated by the over-approximations used in back-substitution. Often, the currently computed bound can then be significantly improved by "pushing" it towards the

https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_7

This article is licensed under a Creative Commons Attribution 4.0 International License

true function, in a way that maintains its validity. For example, suppose that we upper-bound a function f with a function g, i.e. $\forall x.\ g(x) \geq f(x)$. If we discover that the minimal approximation error is 5, i.e. $\min_x\{g(x) - f(x)\} = 5$, then $g(x) - 5$ can be used as a better upper bound for f than the original g. By integrating this simple principle into the back-substitution process, we show that we can obtain much tighter bounds, which eventually translates to the ability to verify more difficult properties.

We propose here a verification approach, called *DeepMIP*, that uses symbolic-bound tightening enhanced with our error-optimization method. At each iteration of the back-substitution, DeepMIP invokes an external MIP solver [26] to compute bounds on the error of the current approximation, and then uses these bounds to improve that approximation. As we show, this leads to an improved ability to solve verification benchmarks when compared to state-of-the-art, symbolic-bound tightening techniques. We discuss the different advantages of the approach, as well as the extra overhead that it incurs, and various enhancements that could be used to expedite it further.

The rest of the paper is organized as follows. We begin by presenting the necessary background on DNNs, DNN verification, and on symbolic-bound propagation in Sec. II. Next, in Sec. III we show how one can express the approximation error incurred as part of the back-substitution process. In Sec. IV we present the DeepMIP algorithm, followed by its evaluation in Sec. V. Related work is discussed in Sec. VI, and we conclude in Sec. VII.

II. Background

Neural networks. A fully-connected feed-forward neural network with $k+1$ layers is a function $N : \mathbb{R}^m \rightarrow \mathbb{R}^n$. Given an input $\boldsymbol{x} \in \mathbb{R}^m$, we use $N_i(\boldsymbol{x})$ to denote the values of neurons in the i^{th} layer ($0 \leq i \leq k$). The output of the neural network $N(\boldsymbol{x})$ is defined as $N_k(\boldsymbol{x})$, which we refer to as the output layer. More concretely, for $1 \leq i \leq k$,

$$N_i(\boldsymbol{x}) = \sigma(W^{i-1}N_{i-1}(\boldsymbol{x}) + b^{i-1})$$

where W^{i-1} is a *weight matrix*, b^{i-1} is a *bias vector*, σ is an activation function (in this paper, we focus on the ReLU activation function, defined as $\text{ReLU}(x) = \max\{0, x\}$ and use σ and ReLU interchangeably unless otherwise specified) and $N_0(\boldsymbol{x}) = \boldsymbol{x}$. We refer to N_0 as the input layer. Typically, non-linear activations are not applied to the output layer. Thus, when $i = k$, we let σ be the identity function. We note that our techniques are general, and apply to other activation functions (MaxPool, LeakyReLU) and architectures (e.g., convolutional, residual).

Neural network verification. The *neural network verification problem* [31], [39] is defined as follows: given an input domain $\mathcal{D}_i \subseteq \mathbb{R}^m$ and an output domain domain $\mathcal{D}_o \subseteq \mathbb{R}^n$, the goal is to determine whether $\forall \boldsymbol{x} \in \mathcal{D}_i, N(\boldsymbol{x}) \in \mathcal{D}_o$. If the answer is affirmative, we say that the verification property pair $\langle \mathcal{D}_i, \mathcal{D}_o \rangle$ holds. In this paper, we assume that the neural network has a single output neuron and that the verification problem can be reduced to the problem of finding the minimum and/or maximum values for that single output neuron:

$$\min_{\boldsymbol{x} \in \mathcal{D}_i}(N(\boldsymbol{x})) \qquad \max_{\boldsymbol{x} \in \mathcal{D}_i}(N(\boldsymbol{x})) \tag{1}$$

For example, if $\mathcal{D}_o$ is the interval $[-2, 7]$ and we discover that $\min_{x \in \mathcal{D}_i}(N(x)) = 1$ and $\max_{x \in \mathcal{D}_i}(N(x)) = 3$, then we are guaranteed that the property holds. We will focus on solving just the maximization problem, although the method that we present next can just as readily be applied towards the minimization problem.

A straightforward way to solve the optimization problem in Eq. 1 is to encode the neural network as a mixed integer programming (MIP) instance [11], [31], [49], and then solve the problem using a MIP solver, which often employs a branch-and-bound procedure. While this approach has proven effective at verifying small DNNs, it faces a scalability barrier when it comes to larger networks. Therefore, before invoking the branch-and-bound procedure, existing solvers typically first seek to prove the property with abstraction-based techniques (symbolic-bound propagation), which have more tractable runtime.

Symbolic-bound propagation. Symbolic-bound propagation [21], [51] is a method of obtaining bounds on the concrete values a neuron may obtain. When applied to a network's output neuron, it enables us to obtain an approximate solution to the optimization problems from Eq. 1, which may be sufficient to determine that the property holds. For example, continuing the example from before, if we are unable to exactly compute that $\max_{x \in \mathcal{D}_i}(N(x)) = 3$ but can determine that $\max_{x \in \mathcal{D}_i}(N(x)) < 5$, this is enough for concluding that the property in question holds. The idea underlying symbolic-bound propagation is to start from the bounds for the input layer provided in $\mathcal{D}_i$, and then propagate them, layer-by-layer, up to the output layer. It has been observed that while affine transformations allow us to precisely propagate bounds from a layer to its successor, activation functions introduce inaccuracies [45].

Before formally defining symbolic bound propagation, we start with an intuitive example using the network in Fig. 1. Let $\boldsymbol{x}^i$ denote the *pre-activation* values of the neurons in layer i, and let $\boldsymbol{y}^i = \sigma(\boldsymbol{x}^i)$ denote their *post-activation* values; similarly, let x^i_j and $y^i_j = \sigma(x^i_j)$ denote the pre- and post-activation values of neuron j in layer i; and let l^i_j, u^i_j denote the concrete (scalar) lower- and upper-bound for x^i_j, i.e. $l^i_j \leq x^i_j \leq u^i_j$ when the DNN is evaluated on any input from $\mathcal{D}_i$. Assume that $\mathcal{D}_i$ is the following box domain:

$$\mathcal{D}_i = \{-1 \leq x^0_i \leq 1 \mid i \in \{0, 1, 2\}\}$$

and that we wish to compute bounds for the single output neuron, x^3_0.

We begin by propagating the bounds through the first affine layer. According to the network's weights and biases, we get:

$$x^1_0 = x^0_0 + x^0_1, \qquad x^1_1 = x^0_0 - x^0_1, \qquad x^1_2 = x^0_2$$

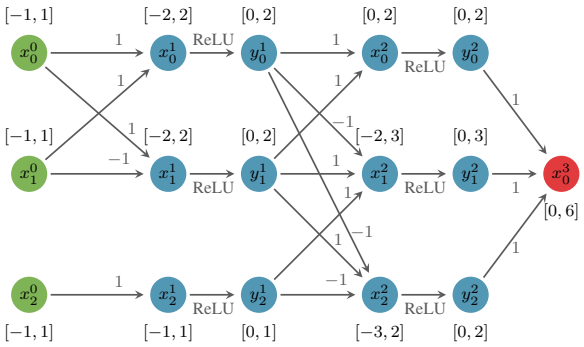

Fig. 1: A neural network.

these equations allow us to compute concrete lower and upper bounds for each of these neurons, by substituting the input neurons (x_0^0, x_1^0, x_2^0) with their corresponding concrete bounds (according to the sign of their coefficients). Using this process, we obtain:

$$x_0^1 \in [-2, 2], \qquad x_1^1 \in [-2, 2], \qquad x_2^1 \in [-1, 1]$$

this propagation, often referred to as *interval arithmetic* [15], is precise for individual neurons: indeed, x_0^1, x_1^1 and x_2^1 can each take on any value in their respective computed ranges. However, much important information is lost when using just interval arithmetic: for example, it is impossible for x_0^1 and x_1^1 to *simultaneously* be assigned 2. As we will later see, symbolic-bound propagation addresses this issue by capturing some of the dependencies between neurons, and using these dependencies in producing tighter bounds.

For now, we continue propagating our computed bounds to neurons y_0^1, y_1^1 and y_2^1. The output range of a ReLU is the non-negative part of its input range, which yields:

$$y_0^1 \in [0, 2], \qquad y_1^1 \in [0, 2], \qquad y_2^1 \in [0, 1]$$

and the next, affine layer is again handled using interval arithmetic. Using the expressions

$$x_0^2 = y_0^1 + y_1^1, \qquad x_1^2 = -y_0^1 + y_1^1 + y_2^0, \qquad x_2^2 = -y_0^1 + y_1^1 - y_2^0$$

and substituting each y_i^1 with the appropriate bound, we obtain:

$$x_0^2 \in [0, 4], \qquad x_1^2 \in [-2, 4], \qquad x_0^2 \in [-4, 2]$$

Unfortunately, as we soon show, the bounds computed for x_0^2, x_1^2, x_2^2 are not tight. A better approach is to compute *symbolic bounds*, as opposed to concrete ones, in a way that lets us carry additional information about the dependencies between neurons. In symbolic-bound propagation, we seek to express the upper and lower bounds of each neuron as a linear combination of neurons from earlier layers, using a process known as *back-substitution*. The main difficulty is to propagate these bounds across ReLU layers, which are not convex; and this is performed by using a *triangle relaxation* of the ReLU function, illustrated in Fig. 2. Assume $x \in [l, u]$; then, using this relaxation, we can deduce the following bounds:

$$\begin{cases} 0 \le \sigma(x) \le 0 & \text{if } u \le 0 \\ x \le \sigma(x) \le x & \text{if } l \ge 0 \\ \alpha x \le \sigma(x) \le \frac{u}{u-l}(x-l) & \text{otherwise, for any } 0 \le \alpha \le 1 \end{cases}$$

Different symbolic bound propagation methods use different heuristics for choosing α [45], [54]; but this is beyond our scope here, and our proposed technique is compatible with any such heuristic. For our running example, we arbitrarily choose the values of α; and for our implementation, we use an existing heuristic [54].

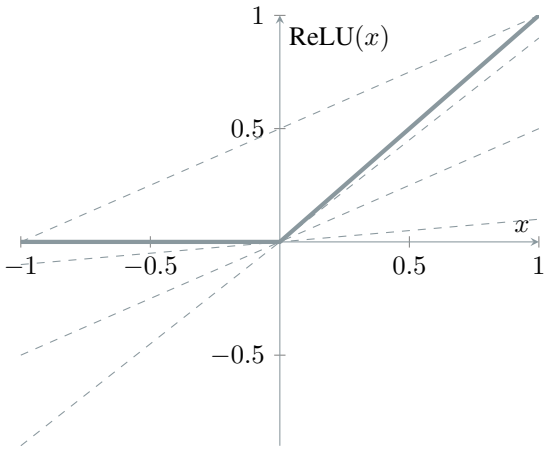

Fig. 2: A triangle relaxation of a ReLU function for $x \in [-1, 1]$. The solid lines correspond to the exact ReLU function, and the dotted lines represent the relaxed lower and upper bounds, for different values of α.

Using this relaxation, we show how to compute symbolic bounds that yield tighter bounds for the x_i^2 neurons. First observe neuron x_0^2, given as $x_0^2 = y_0^1 + y_1^1 = \sigma(x_0^1) + \sigma(x_1^1)$. To obtain its lower bound we first substitute both $y_0^1 = \sigma(x_0^1)$ and $y_1^1 = \sigma(x_1^1)$ with their corresponding triangle relaxation lower bounds, with the choice of $\alpha = 0$ for both (we note that it is possible to choose different α values for different variables). For the upper bound, we use the linear upper bound from the triangle relaxation. By using the bounds we already know for nodes in previous layers, we get that:

$$\begin{aligned} x_0^2 &\ge 0 \cdot x_0^1 + 0 \cdot x_1^1 = 0 \\ x_0^2 &\le \frac{1}{2}\left(x_0^1 + 2\right) + \frac{1}{2}\left(x_1^1 + 2\right) = \frac{1}{2}\left(x_0^1 + x_1^1\right) + 2 \\ &= \frac{1}{2}\left(\left(x_0^0 + x_1^0\right) + \left(x_0^0 - x_1^0\right)\right) + 2 = x_0^0 + 2 \le 3 \end{aligned}$$

which indeed produces a tighter upper bound than the one obtained for x_0^2 using interval propagation. Similarly, we get

that for x_1^2:

$$\begin{aligned}
x_1^2 &\geq -\frac{1}{2}\left(x_0^1+2\right)+0\cdot x_1^1+0\cdot x_2^1\\
&= -\frac{1}{2}\left(x_0^0+x_1^0\right)-1=-2\\
x_1^2 &\leq -0\cdot x_0^1+\frac{1}{2}\left(x_1^1+2\right)+\frac{1}{2}\left(x_2^1+1\right)\\
&= \frac{1}{2}\left(x_1^1+x_2^1\right)+1.5=\frac{1}{2}\left(x_0^0-x_1^0+x_2^0\right)+1.5\leq 3
\end{aligned}$$

and for x_2^2:

$$\begin{aligned}
x_2^2 &\geq -\frac{1}{2}\left(x_0^1+2\right)+0\cdot\left(x_1^1\right)-\frac{1}{2}\left(x_2^1+1\right)\\
&= -\frac{1}{2}\left(x_0^1+x_2^1\right)-1.5=-\frac{1}{2}\left(x_0^0+x_1^0+x_2^0\right)-1.5\geq -3\\
x_2^2 &\leq -0\cdot x_0^1+\frac{1}{2}\left(x_1^1+2\right)-0\cdot x_2^1\\
&= \frac{1}{2}x_1^1+1=\frac{1}{2}\left(x_0^0-x_1^0\right)+1\leq 2
\end{aligned}$$

We have thus obtained the following bounds:

$$x_0^2\in[0,3],\qquad x_1^2\in[-2,3],\qquad x_2^2\in[-3,2]$$

We note that while these bounds are tighter than the ones produced by interval propagation, and are in fact optimal for x_1^2, x_2^2, this is not the case for x_0^2 (the optimal bounds are displayed in square brackets in Fig. 1). The reason for this sub-optimality is discussed in Section III.

We continue to propagate our bounds through the next layer, obtaining:

$$y_0^2\in[0,3],\qquad y_1^2\in[0,3],\qquad y_2^2\in[0,2]$$

and finally reach:

$$\begin{aligned}
x_0^3 &= y_0^2+y_1^2+y_2^2=\sigma(x_0^2)+\sigma(x_1^2)+\sigma(x_2^2)\\
&\leq x_0^2+\frac{3}{5}\left(x_1^2+2\right)+\frac{2}{5}\left(x_2^2+3\right)\\
&= 2y_1^1+\frac{1}{5}y_2^1+\frac{12}{5}=2\sigma(x_1^1)+\frac{1}{5}\sigma(x_2^1)+\frac{12}{5}\\
&\leq 2\cdot\frac{1}{2}\left(x_1^1+2\right)+\frac{1}{5}\cdot\frac{1}{2}\left(x_2^1+1\right)+\frac{12}{5}\\
&= x_0^0-x_1^0+\frac{1}{10}x_2^0+4.5\leq 6.6
\end{aligned}$$

More generally, the back-substitution process for upper-bounding a neuron x_i^k (assuming we already have valid bounds for all neurons in earlier layers) is iteratively defined as:

$$\begin{aligned}
\max(x_i^k) &= \max(W_i^{k-1}\sigma(\boldsymbol{x}^{k-1}))\\
&\leq \max(W_i^{k-1}R_U^{k-2}\boldsymbol{x}^{k-1})\\
&= \max(W_i^{k-1}R_U^{k-2}W^{k-2}\sigma(\boldsymbol{x}^{k-2}))\\
&\leq \max(W_i^{k-1}R_U^{k-2}W^{k-2}R_U^{k-3}\boldsymbol{x}^{k-2})\\
&= \ldots\leq \max(W_i^{k-1}\prod_{j=k-2}^{0}\left(R_U^jW^j\right)\boldsymbol{x}^0)
\end{aligned}$$

(Biases and constants are handled similarly, and are omitted for clarity.) At each step, we can replace the variables of $\boldsymbol{x}^i$ by their respective concrete bounds $[l_j^i, u_j^i]$, in an interval-arithmetic fashion, to obtain a valid concrete upper bound for the value of $\max(x_i^k)$. We refer to this operation as *concretization*. We call the matrices R_L^i, R_U^i the respective lower- and upper-bound *relaxation* matrices [54]. These matrices apply the appropriate triangle relaxation to each ReLU, allowing us to replace it with a linear bound, and are defined using the current symbolic bounds for each ReLU as well as the weight matrix of the layer the precedes it. The two matrices are defined such that $\forall\boldsymbol{x}\in\mathcal{D}_i$:

$$\boldsymbol{\omega_i}R_L^i\boldsymbol{x}+c_L\leq\boldsymbol{\omega_i}\sigma(\boldsymbol{x})\leq\boldsymbol{\omega_i}R_U^i\boldsymbol{x}+c_U$$

where c_L and c_U are scalar constants; and $\boldsymbol{\omega_i}$ is a row vector containing the coefficients of each $\sigma(x_j)$, resulting in linear bounds for the sum of ReLUs. A precise definition of these matrices appears in Sec. A of the Appendix; and a similar procedure can be applied for lower-bounding x_i^k.

At first glance, the iterative back-substitution process may seem counter productive; indeed, in each iteration where we move to an earlier layer of the network, we use a less-than-equals transition, which seems to indicate that the upper bound that we will eventually reach is more loose than the present bound. This, however, is not so; and the reason is the *concretization* process. When we concretize the bounds in some later iteration, it is possible that the known bounds for the variables in that layer of the network will lead to a tighter upper bound than the one that can be derived presently. More generally, this process can be regarded as a trade-off between computing looser expressions for the bound, but being able to concretize them over more exact domains — which could result in tighter bounds [45].

III. Errors in Back-Substitution

As previously mentioned, although symbolic-bound computation using back-substitution can derive tighter bounds than naïve interval propagation, there are cases in which the computed bounds are sub-optimal: for example, while the bounds computed for x_1^2 and x_2^2 were tight (i.e., there exists an input in $\mathcal{D}_i$ for which they are met), the bounds for x_0^2 and x_0^3 were not. In this section, we analyze the reasons behind such sub-optimal bounds. We begin with the following definitions:

***Definition** 1 (Optimal bias for bound):* let $f:\mathbb{R}^n\to\mathbb{R}$ be a function and let $U_f(\boldsymbol{x})\equiv\boldsymbol{\omega x}+b$ ($\boldsymbol{\omega}\in\mathbb{R}^n, b\in\mathbb{R}$) be a valid linear upper bound for f over the domain $\mathcal{D}$, i.e., $\forall\boldsymbol{x}\in\mathcal{D}: U_f(\boldsymbol{x})\geq f(\boldsymbol{x})$. We say that b is the *optimal bias* for $U_f(\boldsymbol{x})$ if $\forall b^*: b^*<b$, it holds that $U_f^*(\boldsymbol{x})\equiv\boldsymbol{\omega x}+b^*$ is no longer a valid upper bound for f. The definition for the optimal bias for f's lower bound is symmetrical.

An example of optimal and sub-optimal upper bounds appears in Fig. 3. In the graph depicted therein, we plot an upper bound for the function $\mathrm{ReLU}(x)$. The bias value of the first bound (in red) is 1; and as we can see, the resulting bound is not tight. When we set the bias value to $1/2$, the bound becomes tight, equaling the function at points $x=-1$ and $x=1$, and so that is the optimal bias value for that bound.

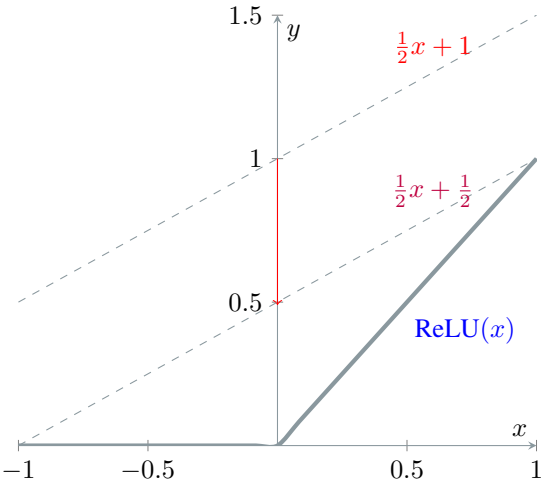

Fig. 3: A simplified illustration of an optimal and sub-optimal bounds for a ReLU function over $x \in [-1, 1]$.

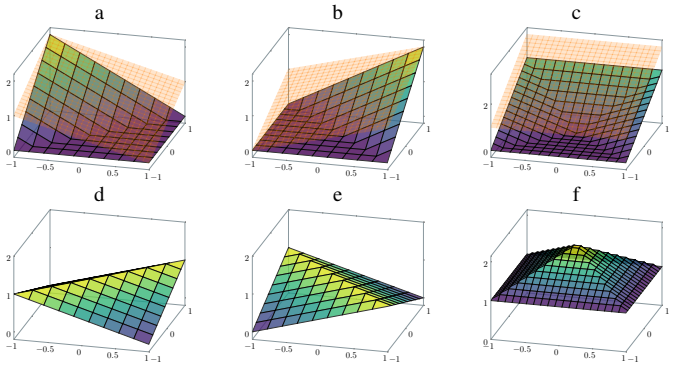

Fig. 4: Illustration of the formation of detached bounds as a result of summed errors. Sub-figures a and b correspond to $y_0^1 = \text{ReLU}(x_0^0 + x_1^0)$, $y_1^1 = \text{ReLU}(x_0^0 - x_1^0)$ and their relaxed upper bounds (in orange); and sub-figure c corresponds to $x_0^2 = y_0^1 + y_1^1$ and its symbolic upper bound, computed using back-substitution.

***Definition** 2 (Bound error):* Let $f : \mathbb{R}^n \to \mathbb{R}$, and let $g(\boldsymbol{x})$ be an upper bound for f over domain $\mathcal{D}$, such that we have: $\forall \boldsymbol{x} \in \mathcal{D} : g(\boldsymbol{x}) \geq f(\boldsymbol{x})$. We define the error of g with respect to f as the function: $E(\boldsymbol{x}) = g(\boldsymbol{x}) - f(\boldsymbol{x})$. The case for a lower bound is symmetrical.

We observe that a linear bound g for f over the domain $\mathcal{D}_i$ has *optimal bias* iff $\exists \boldsymbol{x} \in \mathcal{D}_i : E(\boldsymbol{x}) = 0$. We refer to any bound that has a sub-optimal bias, i.e. $\forall \boldsymbol{x} \in \mathcal{D}_i : E(\boldsymbol{x}) > 0$, as a *detached bound*. We show that these detachments occur naturally as part of the back-substitution process, and are partially responsible for the discovery of sub-optimal concrete bounds.

It is straightforward to see that the aforementioned triangle relaxation for ReLUs produces linear bounds that are bias-optimal for each individual ReLU. However, as it turns out, this may not be the case when multiple ReLUs are involved. In a typical DNN, a neuron's value is computed as a weighted sum of the ReLUs of values from its preceding layer. Consequently, when we calculate an upper bound for the neuron using back-substitution, we are in fact upper-bounding a sum of ReLUs by summing their individual upper bounds. This can result in a *detached bound*, where, despite the fact that each ReLU was approximated using a bound with an optimal bias, the resulting combined bound does not have optimal bias.

An illustration of this phenomenon appears in Fig. 4. Sub-figures a and b therein show the graph of ReLU functions, plotted along their triangle-relaxation upper bound (in orange). Sub-figure c then shows the graph of the *sum* of the two ReLU functions from sub-figures a and b, along with the sum of their individual upper bounds (again, in orange). As we can see, although the upper bounds in a and b touch the functions they are approximating in at least one point (and are hence bias-optimal), the bound in c is detached, and is hence not bias-optimal. Each figure in the lower row of Fig. 4 shows the over-approximation error of the figure directly above it.

More formally, the error of the upper bound for ReLU(x) with current bounds $l < 0 < u$ is:

$$E(x) = \frac{u}{u-l}(x-l) - \sigma(x) \qquad x \in [l, u]$$

and we note that $E(l) = E(u) = 0$. In more complex cases, such as the case of the multivariate function $x_0^2 = y_0^1 + y_1^1$ depicted in Fig. 4, the coordinates where the bound error equals zero could be different for y_0^1 and y_1^1 — resulting in the bound obtained for x_0^2, their sum, becoming detached from the true value of the function. We now show it for the case of x_0^2 in greater detail:

$$x_0^2 = \sigma(x_0^1) + \sigma(x_1^1) = \sigma(x_0^0 + x_1^0) + \sigma(x_0^0 - x_1^0)$$

An upper bound is computed using the relaxations:

$$\sigma(x_0^0 + x_1^0) \leq \frac{1}{2}\left(x_0^0 + x_1^0 + 2\right)$$
$$\sigma(x_0^0 - x_1^0) \leq \frac{1}{2}\left(x_0^0 + x_1^0 + 2\right)$$

where each relaxation has its own relaxation error:

$$E_0^1(x_0^0, x_1^0) = \frac{1}{2}(x_0^0 + x_1^0 + 2) - \sigma(x_0^0 + x_1^0)$$
$$E_1^1(x_0^0, x_1^0) = \frac{1}{2}(x_0^0 + x_1^0 + 2) - \sigma(x_0^0 - x_1^0)$$

The relaxed linear bound obtained is:

$$x_0^2 \leq \frac{1}{2}(x_0^0 + x_1^0 + 2) + \frac{1}{2}(x_0^0 + x_1^0 + 2) = x_0^0 + 2$$

And its error is the sum of the errors of its summands:

$$\begin{aligned} E_{\text{total}}(x_0^0, x_1^0) &\equiv E_0^1 + E_1^1 \\ &= x_0^0 + 2 - \sigma(x_0^0 + x_1^0) - \sigma(x_0^0 - x_1^0) \end{aligned}$$

We note that:

$$\min(E_0^1) = E_0^1(-1, -1) = E_0^1(1, 1) = 0$$
$$\min(E_1^1) = E_1^1(-1, 1) = E_1^1(1, -1) = 0$$

However:

$$\min(E_{\text{total}}) = E_{\text{total}}\left(-1, x_1^0\right) = 1$$

The reason for this is that at the coordinates $\langle -1, -1\rangle$ and $\langle 1, 1\rangle$ where $E_0^1(-1,-1) = E_0^1(1,1) = 0$, we have that $E_1^1(-1,-1) = E_1^1(1,1) = 1$; and vice-versa, for the coordinates $\langle -1, 1\rangle$ and $\langle 1, -1\rangle$, where $E_1^1(-1,1) = E_1^1(1,-1) = 0$ and $E_0^1(-1,1) = E_0^1(1,-1) = 1$. The optimal linear bound for

$$x_0^2 = \sigma(x_0^0 + x_1^0) + \sigma(x_0^0 - x_1^0)$$

is in fact $x_0^2 \leq x_0^0 + 1$, which is the bias-optimal version of the existing linear bound of $x_0^2 \leq x_0^0 + 2$.

IV. DeepMIP: Minimizing Back-Substitution Errors

During a back-propagation execution, the over-approximations of individual ReLUs are repeatedly summed up, which leads to bounds that become increasingly more detached with each iteration — and this results in very loose concrete bounds that hamper verification. We now describe our method, which we term *DeepMIP*, for "tightening" detached bounds, with the goal of eventually obtaining tighter concrete bounds. The idea is to alter the back-propagation mechanism, so that in each iteration it *minimizes* the sum of errors that result from the relaxation of the current activation layer — effectively pushing loose upper bounds down towards the function, by decreasing their bias values (a symmetrical mechanism can be applied for lower bounds). More specifically, we propose to rewrite the general back-substitution rule for a single iteration as follows:

$$\begin{aligned}
\max(x_i^k) &= \max(W_i^{k-1}\sigma(\boldsymbol{x}^{k-1})) \\
&= \max\left(W_i^{k-1}R_U^{k-2}\boldsymbol{x}^{k-1}\right. \\
&\qquad \left. - \left(W_i^{k-1}R_U^{k-2}\boldsymbol{x}^{k-1} - W_i^{k-1}\sigma(\boldsymbol{x}^{k-1})\right)\right) \\
&= \max(W_i^{k-1}R_U^{k-2}\boldsymbol{x}^{k-1} - E^{k-1}) \\
&\leq \max(W_i^{k-1}R_U^{k-2}\boldsymbol{x}^{k-1}) - \min(E^{k-1})
\end{aligned}$$

Observe that while $\min(E^{k-1})$ is non-convex, it contains no nested ReLUs, and can often be efficiently solved by MIP solvers [49]. Thus, as DeepMIP performs the iterative back-substitution process, it can invoke a MIP solver to minimize the error in each iteration, and use it to improve the deduced bounds. The pseudo-code for the algorithm appears in the full version of this paper [56]. Observe that MiniMIP can be regarded as a generalization of modern back-substitution methods [45], [54], in the sense that they only use the non-negativity of the error to produce a trivial bound:

$$\min(E^{k-1}) = \min(W_i^{k-1}R_U^{k-2}\boldsymbol{x}^{k-1} - W_i^{k-1}\sigma(\boldsymbol{x}^{k-1})) \geq 0$$

which is correct, since the error of an upper bound is non-negative by definition (in the lower bound case, the error is non-positive, and so 0 can be used as a trivial upper bound).

To continue our computation we denote the error caused by the over-approximation of the activation of layer t during back-substitution as:

$$E^t \equiv W_i^{k-1}\prod_{j=k-2}^{t-1}(R_U^j W^j)(R_U^{t-1}\boldsymbol{x}^t - \sigma(\boldsymbol{x}^t)) \qquad (2)$$

In the definition above, i is the index of the neuron being bounded by the back-substitution. We get:

$$\begin{aligned}
\max(x_i^k) &\leq \max(W_i^{k-1}R_U^{k-2}\boldsymbol{x}^{k-1}) - \min(E^{k-1}) \\
&= \max(W_i^{k-1}R_U^{k-2}W^{k-2}\sigma(\boldsymbol{x}^{k-2})) - \min(E^{k-1}) \\
&= \max(W_i^{k-1}R_U^{k-2}W^{k-2}R_U^{k-2}\boldsymbol{x}^{k-2} - E^{k-2}) \\
&\quad - \min(E^{k-1}) \\
&\leq \max(W_i^{k-1}R_U^{k-2}W^{k-2}R_U^{k-2}\boldsymbol{x}^{k-2}) \\
&\quad - \min(E^{k-2}) - \min(E^{k-1}) \\
&= \dots \\
&\leq \max(W_i^{k-1}\prod_{j=k-2}^{0}(R_U^j W^j)\boldsymbol{x}^0) - \sum_{j=k-1}^{0}\min(E^j)
\end{aligned}$$

Finally, the maximization problem is transformed into a linear sum over a box domain, which is easy to solve. Since each E^j is shallow (contains no nested ReLUs), it can be minimized efficiently using MIP solvers, and each non-trivial minimum that is found will improve the tightness of the final upper bound. However, we note that the number of MIP problems generated by this process increases linearly with the *depth* of the neuron within the network — i.e., for a neuron in layer k, there are k minimization problems to solve. For deeper networks, especially ones with large domains or ones where many layers only have very loose bounds, minimizing the error terms could become computationally expensive.

Optimization: Direct MIP encoding. As part of its operation, DeepMIP dispatches MIP problems, each corresponding to the over-approximation error of a particular layer. Specifically when it over-approximates the first layer:

$$\begin{aligned}
&\max(W_i^{k-1}\prod_{j=k-2}^{1}(R_U^j W^j)\sigma(W^0\boldsymbol{x}^0)) - \sum_{j=k-2}^{1}\min(E^j) \\
&\leq \max(W_i^{k-1}\prod_{j=k-2}^{0}(R_U^j W^j)\boldsymbol{x}^0) - \min(E^0) \\
&\quad - \sum_{j=k-2}^{1}\min(E^j)
\end{aligned}$$

it will directly solve the linear optimization problem:

$$\max(W_i^{k-1}\prod_{j=k-2}^{0}(R_U^j W^j)\boldsymbol{x}^0)$$

and use a MIP solver to solve:

$$\min(E^0) = \min\left(W_i^{k-1}\prod_{j=k-2}^{1}(R_U^j W^j)(R_U^0\boldsymbol{x}^t - \sigma(\boldsymbol{x}^0)\right)$$

We observe that in this particular case, since we reached the input layer, the initial term can instead be directly solved as a separate MIP query:

$$\max(W_i^{k-1} \prod_{j=k-2}^{1} (R_U^j W^j)\sigma(W^0 \boldsymbol{x}^0))$$

which may result in tighter bounds, since it prevents any additional imprecision. We note that this optimization to DeepMIP generalizes the common practice of directly finding the concrete bounds of the neurons in the first layer using MIP solvers, and only applying back-substitution from the second layer onward [37], [54].

We illustrate this approach by repeating the back-substitution process for x_0^3 from our running example:

$$\begin{aligned}\max(x_0^3) &= \max(y_0^2 + y_1^2 + y_2^2)\\ &= \max(\sigma(x_0^2) + \sigma(x_1^2) + \sigma(x_2^2))\\ &= \max\Big(\sigma(y_0^1 + y_1^1) + \sigma(-y_0^1 + y_1^1 + y_2^1)\\ &\qquad + \sigma(-y_0^1 + y_1^1 - y_2^1)\Big)\\ &= \max(A - E_U^2) \leq \max(A) - \min(E_U^2)\end{aligned}$$

where

$$\begin{aligned}A = (y_0^1+y_1^1)+\frac{3}{5}(-y_0^1+y_1^1+y_2^1)+\frac{2}{5}(-y_0^1+y_1^1-y_2^1)+\frac{12}{5}\\ = 2y_1^1 + \frac{1}{5}y_2^1 + \frac{12}{5}\end{aligned}$$

and E_U^2 is defined as per Eq. 2:

$$\begin{aligned}E_U^2 &= (y_0^1 + y_1^1) + \frac{3}{5}(-y_0^1 + y_1^1 + y_2^1)\\ &\quad + \frac{2}{5}(-y_0^1 + y_1^1 - y_2^1) + \frac{12}{5} - \sigma(y_0^1 + y_1^1)\\ &\quad - \sigma(-y_0^1 + y_1^1 + y_2^1) - \sigma(-y_0^1 + y_1^1 - y_2^1)\\ &= 2y_1^1 + \frac{1}{5}y_2^1 + \frac{12}{5} - \sigma(y_0^1 + y_1^1)\\ &\quad - \sigma(-y_0^1 + y_1^1 + y_2^1) - \sigma(-y_0^1 + y_1^1 - y_2^1)\end{aligned}$$

Simplifying these expressions, we get that

$$\begin{aligned}\max(x_0^3) &\leq \max(A) - \min(E_U^2)\\ &= \max(2y_1^1 + \frac{1}{5}y_2^1 + \frac{12}{5}) - \min(E_U^2)\end{aligned}$$

Using a MIP solver to find the minimum of E_U^2 over the variables of $\boldsymbol{y}^1$ reveals that $\min(E_U^2) = \frac{2}{5}$. We substitute this, and get:

$$\max(x_0^3) \leq \max(2y_1^1 + \frac{1}{5}y_2^1 + \frac{12}{5}) - \frac{2}{5}$$

Finally, since we have reached the first layer, we write:

$$\begin{aligned}\max(x_0^3) &\leq \max(2y_1^1 + \frac{1}{5}y_2^1 + \frac{12}{5}) - \frac{2}{5}\\ &= \max(2\sigma(x_1^1) + \frac{1}{5}\sigma(x_2^1) + \frac{12}{5}) - \frac{2}{5}\\ &= \max(2\sigma(x_0^0 - x_1^0) + \frac{1}{5}\sigma(x_2^0) + \frac{12}{5}) - \frac{2}{5}\end{aligned}$$

and then, using our proposed enhancement, we directly solve this maximization over the input layer instead of back-substituting it any further. The MIP solver replies that:

$$\max(2\sigma(x_0^0 - x_1^0) + \frac{1}{5}\sigma(x_2^0) + \frac{12}{5}) = 6\frac{2}{5}$$

and we then substitute this value to obtain:

$$\max(x_0^3) \leq 6\frac{2}{5} - \frac{2}{5} = 6$$

As we can see, minimizing the errors by using MIP (which is very fast in practice) allows us to back-substitute bounds with optimal bias, which yields tighter bounds for the output variable.

MiniMIP. While DeepMIP produces very strong bounds, for each neuron it must solve multiple MIP instances during back-substitution — many of them for bounds that may already be bias-optimal. This large number of instances to solve can result in a large overhead, and makes it worthwhile to explore heuristics for only solving *some* of these instances.

To illustrate this, we propose a particular, aggressive heuristic that we call *MiniMIP*. Instead of minimizing all error terms during back-substitution, MiniMIP only solves the final query in this series — that is, the query in which the bounds of the current layer are expressed as sums of ReLUs of input neurons. This approach significantly reduces overhead: exactly one MIP instance is solved in each iteration, regardless of the depth of the layer currently being processed. As we later see in our evaluation, even this is already enough to achieve state-of-the-art performance and very tight bounds; and the resulting queries can be solved very efficiently [49].

V. Evaluation

Implementation. For evaluation purposes, we created a proof-of-concept implementation of our approach in Python. The implementation code, alongside all the benchmarks described in this section, is publicly available online [55]. Our implementation uses the PyTorch library [40] for computing the optimal value of α for each ReLU's triangle relaxation, as is done in other tools [54]. We use Gurobi [26] as the MIP solver for the minimization of errors and direct concretization of bounds. We ran all experiments on a compute cluster consisting of Xeon E5-2637 CPUs, and a 2-hour timeout per experiment. We note that our implementation currently runs on CPUs only, and extending it to support GPUs is left for future work.

Abstraction refinement cascade. For each verification query, prior to applying our iterative error minimization scheme, we configured our implementation to first run a light-weight, "ordinary" symbolic-bound propagation pass. Specifically, we ran a single pass of the DeepPoly mechanism [45]. A similar technique is applied by other tools [37].

Benchmarks. We evaluated our approach on fully-connected, ReLU networks trained over the MNIST dataset, taken from the ERAN repository [19]. The topologies of the networks we used appear in Table I.

TABLE I: The DNNs used in our evaluation.

Dataset	Model	Type	Neurons	Hidden Layers	Activation
MNIST	6×100	FC	510	5	ReLU
	9×100		810	8	
	6×200		1010	5	
	9×200		1610	8	

For verification queries, we followed standard practice [31], [37], [54], and attempted to prove the *adversarial robustness* of the first 1000 images of the MNIST test set: that is, we used verification to try and prove that ϵ-perturbations to correctly classified inputs in the dataset cannot change the classification assigned by the DNN.

We compared the DeepMIP approach (specifically, MiniMIP) to two state-of-the-art verification approaches [9]: the PRIMA solver [37], and our implementation of the α-CROWN method [54], which represents the state of the art in symbolic-bound tightening with back-substitution. Indeed, many other verification tools integrate back-substitution with additional techniques, such as search-based techniques [32] or abstraction-refinement [7], making it more difficult to measure the effectiveness of the back-substitution component alone. However, since the α-CROWN implementation in our evaluation also served as the baseline back-substitution method to which we added our methods, any difference between the two is solely due to the addition of our suggested technique. The results of our experiments are summarized in Table II. Recall that symbolic-bound propagation techniques are incomplete, and may fail to prove a given query; the *Solved* columns indicate the number of instances (out of 1000) that each method was able to prove to be robust to adversarial perturbations. The *Time* columns indicate the run time of each method (including timeouts), averaged over the 1000 benchmarks solved.

Our results clearly indicate the superiority of the bounds discovered by DeepMIP: indeed, in all categories, our approach was able to solve the largest number of instances, solving a total of 2378 instances, compared to 2183 instances solved by PRIMA (198 extra instances solved) and 1087 instances solved by α-CROWN (1291 extra instances solved). These improvements come with an overhead, due to the additional MIP queries that need to be solved: our approach is approximately 5.6 times slower than α-CROWN, and 2.5 times slower than PRIMA. Furthermore, DeepMIP timed out on 2 out of the 3829 total benchmarks tested ($\approx 0.05\%$), while PRIMA and α-CROWN did not have any timeouts.

The main conclusions that we draw from these experiments are that (i) the DeepMIP approach has a significant potential for solving queries that other approaches cannot; and (ii) additional work, in the form of improved heuristics, engineering improvements, and support for GPUs is still required to make our approach faster. Our results also indicate that a portfolio-based approach, which starts from light-weight techniques and then progresses towards DeepMIP for difficult queries, could enjoy the benefits of both worlds.

VI. Related Work

The topic of DNN verification has been receiving significant attention from the formal methods community, and various tools and methods and have been proposed for addressing it. These include techniques that leverage SMT solvers (e.g., [27], [32], [39], [53]), LP and MILP solvers (e.g., [13], [15], [36], [49]), reachability analysis [47], abstraction-refinement techniques [7], [16], [17], and many others. The techniques most related to DeepMIP are those that rely on the propagation of symbolic bounds using abstract interpretation (e.g., [21], [50]–[52]). Recent work has also extended beyond answering binary questions about DNNs, instead targeting tasks such as automated DNN repair [23], [34], DNN simplification [22], [35], ensemble selection [3], and quantitative verification and optimization [10], [46]; and also the verification of recurrent neural networks [28], [41], [57] and reinforcement-learning based systems [4], [18], [29]. Our proposed techniques could be integrated into any number of these approaches.

Bound propagation has been playing a significant part in DNN verification efforts for the past few years. Starting with interval-arithmetic-based propagation [31] and optimization queries for individual neurons [15], [49], these approaches have progressed to use various relaxations and over-approximations for individual neurons [21], [45], [51] and sets thereof [37], [38], [44], culminating in highly sophisticated approaches [37], [54]. We consider our work as another step in this very promising research direction.

VII. Conclusion and Future Work

We presented an enhancement to the popular back-substitution procedure, which includes a formulation of the over-approximation errors introduced during back-substitution. These errors can then be minimized, in order to greatly tighten the resulting bounds. Our approach achieves tighter bounds than state-of-the-art approaches, but at the cost of longer running times; and we are currently exploring methods for expediting it. Specifically, moving forward, we intend to focus on adding support for GPUs; on better refinement heuristics; on better MIP encoding [6]; and also on improving the core algorithm to utilize previously calculated bounds and errors. Furthermore, we intend to generalize our methods to other abstract domains, and also to integrate them with search-based techniques.

Acknowledgements

The project was partially supported by the Israel Science Foundation (grant number 683/18) and by the Binational Science Foundation (grant number 2020250).

Appendix A
Relaxation Matrices

The matrices R_U^t and R_L^t are how we apply the triangle relaxation during back-substitution over layer t. for example if:

$$x_j^{i+1} = \sigma(x_0^i) - 2\sigma(x_1^i)$$

TABLE II: Comparing DeepMIP to α-CROWN and PRIMA.

Model	ϵ	α-CROWN		PRIMA		DeepMIP (MiniMIP)	
		Solved	Time (seconds)	Solved	Time (seconds)	Solved	Time (seconds)
6×100	0.026	207	38	504	123	581	302
9×100	0.026	223	88	427	252	463	452
6×200	0.015	349	93	652	222	709	801
9×200	0.015	308	257	600	462	625	1121
Total		1087	476	2183	1059	2378	2676

then in order to find a linear upper bound for x_j^{i+1}, we need to replace $\sigma(x_0^i)$ with its triangle-relaxation upper bound (since it has a positive coefficient), and $\sigma(x_1^i)$ with its triangle-relaxation lower bound. This gives rise to:

$$x_j^{i+1} \leq \frac{u_0^i}{u_0^i - l_0^i}(x_0^i - l_0^i) - 2\alpha x_1^i$$

which can be written as (for some constant term d):

$$x_j^{i+1} \leq \frac{u_0^i}{u_0^i - l_0^i} x_0^i - 2\alpha x_1^i + d$$

Written as a vector product:

$$x_j^{i+1} = \begin{bmatrix}1 & -2\end{bmatrix} \cdot \begin{bmatrix}\sigma(x_0^i) \\ \sigma(x_1^i)\end{bmatrix} \leq \begin{bmatrix}1 & -2\end{bmatrix} \cdot \begin{bmatrix}\frac{u_0^i}{u_0^i - l_0^i} & 0 \\ 0 & \alpha\end{bmatrix} \cdot \begin{bmatrix}x_0^i \\ x_1^i\end{bmatrix} + d$$

We use R_U^i to denote the matrix that was used to relax $\sigma(x^i)$, and observe that it depends on the weights/coefficients of each non-linearity about to be relaxed, and also on the existence of $[l^i, u^i]$ in order to compute the corresponding relaxations. Formally we define the matrix $R_U^t(\omega^t, l^t, u^t)$ as:

$$R_U^t(\omega^t, l^t, u^t)[i, j] = 0 \qquad i \neq j$$

$$R_U^t(\omega^t, l^t, u^t)[i, i] \equiv \begin{cases} 1 & \text{if } l_i^t \geq 0 \\ 0 & \text{if } u_i^t \leq 0 \\ \frac{u_i^t}{u_i^t - l_i^t} & \text{if } \omega_i^t \geq 0 \text{ and } l_i^t \leq 0 \leq u_i^t \\ \alpha & \text{if } \omega_i^t \leq 0 \text{ and } l_i^t \leq 0 \leq u_i^t \end{cases}$$

where ω^t is a row vector such that ω_i^t contains the coefficient of $\sigma(x_i^t)$, and l^t, u^t are vectors such that $l_i^t \leq x_i^t \leq u_i^t$. Similarly, we define $R_L^t(\omega^t, l^t, u^t)$ as:

$$R_L^t(\omega^t, l^t, u^t)[i, j] = 0 \qquad i \neq j$$

$$R_L^t(\omega^t, l^t, u^t)[i, i] \equiv \begin{cases} 1 & \text{if } l_i^t \geq 0 \\ 0 & \text{if } u_i^t \leq 0 \\ \frac{u_i^t}{u_i^t - l_i^t} & \text{if } \omega_i^t \leq 0 \text{ and } l_i^t \leq 0 \leq u_i^t \\ \alpha & \text{if } \omega_i^t \geq 0 \text{ and } l_i^t \leq 0 \leq u_i^t \end{cases}$$

We note that there exists similar matrices for updating the constant term during back-substitution; we omit them to reduce clutter. Furthermore, when it is clear from context, we write R_L^t, R_U^t instead of $R_L^t(\omega^t, l^t, u^t), R_U^t(\omega^t, l^t, u^t)$.

REFERENCES

[1] M. Akintunde, A. Kevorchian, A. Lomuscio, and E. Pirovano. Verification of RNN-Based Neural Agent-Environment Systems. In *Proc. 33rd AAAI Conf. on Artificial Intelligence (AAAI)*, pages 197–210, 2019.

[2] M. AlQuraishi. AlphaFold at CASP13. *Bioinformatics*, 35(22):4862–4865, 2019.

[3] G. Amir, G. Katz, and M. Schapira. Verification-Aided Deep Ensemble Selection. In *Proc. 22nd Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, 2022.

[4] G. Amir, M. Schapira, and G. Katz. Towards Scalable Verification of Deep Reinforcement Learning. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 193–203, 2021.

[5] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané. Concrete Problems in AI Safety, 2016. Technical Report. https://arxiv.org/abs/1606.06565.

[6] R. Anderson, J. Huchette, C. Tjandraatmadja, and J. Vielma. Strong Mixed-Integer Programming Formulations for Trained Neural Networks, 2018. Technical Report. http://arxiv.org/abs/1811.08359.

[7] P. Ashok, V. Hashemi, J. Kretinsky, and S. Mohr. DeepAbstract: Neural Network Abstraction for Accelerating Verification. In *Proc. 18th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 92–107, 2020.

[8] G. Avni, R. Bloem, K. Chatterjee, T. Henzinger, B. Konighofer, and S. Pranger. Run-Time Optimization for Learned Controllers through Quantitative Games. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 630–649, 2019.

[9] S. Bak, C. Liu, and T. Johnson. The Second International Verification of Neural Networks Competition (VNN-COMP 2021): Summary and Results, 2021. Technical Report. http://arxiv.org/abs/2109.00498.

[10] T. Baluta, S. Shen, S. Shinde, K. Meel, and P. Saxena. Quantitative Verification of Neural Networks And its Security Applications. In *Proc. 26th ACM Conf. on Computer and Communication Security (CCS)*, 2019.

[11] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi. Measuring Neural Net Robustness with Constraints. In *Proc. 30th Conf. on Neural Information Processing Systems (NIPS)*, 2016.

[12] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars, 2016. Technical Report. http://arxiv.org/abs/1604.07316.

[13] R. Bunel, I. Turkaslan, P. Torr, P. Kohli, and P. Mudigonda. A Unified View of Piecewise Linear Neural Network Verification. In *Proc. 32nd Conf. on Neural Information Processing Systems (NeurIPS)*, pages 4795–4804, 2018.

[14] T. Dreossi, D. Fremont, S. Ghosh, E. Kim, H. Ravanbakhsh, M. Vazquez-Chanlatte, and S. Seshia. VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 432–442, 2019.

[15] R. Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. In *Proc. 15th Int. Symp. on Automated Technology for Verification and Analysis (ATVA)*, pages 269–286, 2017.

[16] Y. Elboher, E. Cohen, and G. Katz. Neural Network Verification using Residual Reasoning. In *Proc. 20th Int. Conf. on Software Engineering and Formal Methods (SEFM)*, 2022.

[17] Y. Elboher, J. Gottschlich, and G. Katz. An Abstraction-Based Framework for Neural Network Verification. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 43–65, 2020.

[18] T. Eliyahu, Y. Kazak, G. Katz, and M. Schapira. Verifying Learning-Augmented Systems. In *Proc. Conf. of the ACM Special Interest Group*

on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM), pages 305–318, 2021.

[19] ERAN. The ERAN Repository, 2022. https://github.com/eth-sri/eran.

[20] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song. Robust Physical-World Attacks on Deep Learning Visual Classification. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 1625–1634, 2018.

[21] T. Gehr, M. Mirman, D. Drachsler-Cohen, E. Tsankov, S. Chaudhuri, and M. Vechev. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proc. 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.

[22] S. Gokulanathan, A. Feldsher, A. Malca, C. Barrett, and G. Katz. Simplifying Neural Networks using Formal Verification. In *Proc. 12th NASA Formal Methods Symposium (NFM)*, pages 85–93, 2020.

[23] B. Goldberger, Y. Adi, J. Keshet, and G. Katz. Minimal Modifications of Deep Neural Networks using Verification. In *Proc. 23rd Int. Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 260–278, 2020.

[24] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[25] D. Gunning. Explainable Artificial Intelligence (XAI), 2017. Defense Advanced Research Projects Agency (DARPA) Project.

[26] Gurobi. The Gurobi MILP Solver, 2021. https://www.gurobi.com/.

[27] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu. Safety Verification of Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*, pages 3–29, 2017.

[28] Y. Jacoby, C. Barrett, and G. Katz. Verifying Recurrent Neural Networks using Invariant Inference. In *Proc. 18th Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, pages 57–74, 2020.

[29] P. Jin, J. Tian, D. Zhi, X. Wen, and M. Zhang. Trainify: A CEGAR-Driven Training and Verification Framework for Safe Deep Reinforcement Learning. In *Proc. 34th Int. Conf. on Computer Aided Verification (CAV)*, pages 193–218, 2022.

[30] K. Julian, J. Lopez, J. Brush, M. Owen, and M. Kochenderfer. Policy Compression for Aircraft Collision Avoidance Systems. In *Proc. 35th Digital Avionics Systems Conf. (DASC)*, pages 1–10, 2016.

[31] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: a Calculus for Reasoning about Deep Neural Networks. *Formal Methods in System Design (FMSD)*, 2021.

[32] G. Katz, D. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, D. Dill, M. Kochenderfer, and C. Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In *Proc. 31st Int. Conf. on Computer Aided Verification (CAV)*, pages 443–452, 2019.

[33] W. Kokke, E. Komendantskaya, D. Kienitz, R. Atkey, and D. Aspinall. Neural Networks, Secure by Construction: An Exploration of Refinement Types. In *Proc. 18th Asian Symposium on Programming Languages and Systems (APLAS)*, pages 67–85, 2020.

[34] B. Könighofer, F. Lorber, N. Jansen, and R. Bloem. Shield Synthesis for Reinforcement Learning. In *Proc. Int. Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 290–306, 2020.

[35] O. Lahav and G. Katz. Pruning and Slicing Neural Networks using Formal Verification. In *Proc. 21st Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD)*, pages 183–192, 2021.

[36] A. Lomuscio and L. Maganti. An Approach to Reachability Analysis for Feed-Forward ReLU Neural Networks, 2017. Technical Report. http://arxiv.org/abs/1706.07351.

[37] M. Müller, G. Makarchuk, G. Singh, M. Puschel, and M. Vechev. PRIMA: General and Precise Neural Network Certification via Scalable Convex Hull Approximations. In *Proc. 49th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2022.

[38] M. Ostrovsky, C. Barrett, and G. Katz. An Abstraction-Refinement Approach to Verifying Convolutional Neural Networks. In *Proc. 20th. Int. Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2022.

[39] L. Pulina and A. Tacchella. An Abstraction-Refinement Approach to Verification of Artificial Neural Networks. In *Proc. 22nd Int. Conf. on Computer Aided Verification (CAV)*, pages 243–257, 2010.

[40] PyTorch. The PyTorch Library, 2022. https://pytorch.org/.

[41] W. Ryou, J. Chen, M. Balunovic, G. Singh, A. Dan, and M. Vechev. Scalable Polyhedral Verification of Recurrent Neural Networks. In *Proc. 33rdd Int. Conf. on Computer Aided Verification (CAV)*, pages 225–248, 2021.

[42] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, and S. Dieleman. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.

[43] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, 2014. Technical Report. http://arxiv.org/abs/1409.1556.

[44] G. Singh, R. Ganvir, M. Puschel, and M. Vechev. Beyond the Single Neuron Convex Barrier for Neural Network Certification. In *Proc. 33rd Conf. on Neural Information Processing Systems (NeurIPS)*, 2019.

[45] G. Singh, T. Gehr, M. Puschel, and M. Vechev. An Abstract Domain for Certifying Neural Networks. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2019.

[46] C. Strong, H. Wu, A. Zeljić, K. Julian, G. Katz, C. Barrett, and M. Kochenderfer. Global Optimization of Objective Functions Represented by ReLU Networks. *Journal of Machine Learning*, pages 1–28, 2021.

[47] X. Sun, K. H., and Y. Shoukry. Formal Verification of Neural Network Controlled Autonomous Systems. In *Proc. 22nd ACM Int. Conf. on Hybrid Systems: Computation and Control (HSCC)*, 2019.

[48] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing Properties of Neural Networks, 2013. Technical Report. http://arxiv.org/abs/1312.6199.

[49] V. Tjeng, K. Xiao, and R. Tedrake. Evaluating Robustness of Neural Networks with Mixed Integer Programming, 2017. Technical Report. http://arxiv.org/abs/1711.07356.

[50] H. Tran, S. Bak, and T. Johnson. Verification of Deep Convolutional Neural Networks Using ImageStars. In *Proc. 32nd Int. Conf. on Computer Aided Verification (CAV)*, pages 18–42, 2020.

[51] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal Security Analysis of Neural Networks using Symbolic Intervals. In *Proc. 27th USENIX Security Symposium*, 2018.

[52] T.-W. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, D. Boning, I. Dhillon, and L. Daniel. Towards Fast Computation of Certified Robustness for ReLU Networks, 2018. Technical Report. http://arxiv.org/abs/1804.09699.

[53] H. Wu, A. Zeljić, G. Katz, and C. Barrett. Efficient Neural Network Analysis with Sum-of-Infeasibilities. In *Proc. 28th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 143–163, 2022.

[54] K. Xu, H. Zhang, S. Wang, Y. Wang, S. Jana, X. Lin, and C.-J. Hsieh. Fast and Complete: Enabling Complete Neural Network Verification with Rapid and Massively Parallel Incomplete Verifiers, 2020. Technical Report. http://arxiv.org/abs/2011.13824.

[55] T. Zelazny, H. Wu, C. Barrett, and G. Katz. DeepMIP Code, 2022. https://doi.org/10.5281/zenodo.6982973.

[56] T. Zelazny, H. Wu, C. Barrett, and G. Katz. On Optimizing Back-Substitution Methods for Neural Network Verification (Full Version), 2022. Technical Report. https://arxiv.org/abs/2208.07669.

[57] H. Zhang, M. Shinn, A. Gupta, A. Gurfinkel, N. Le, and N. Narodytska. Verification of Recurrent Neural Networks for Cognitive Tasks via Reachability Analysis. In *Proc. 24th European Conf. on Artificial Intelligence (ECAI)*, pages 1690–1697, 2020.