Neural Network Verification with Branch-and-Bound for General Nonlinearities

Zhouxing Shi*[1], Qirui Jin*[2], Zico Kolter[3], Suman Jana[4],
Cho-Jui Hsieh[1], and Huan Zhang[5]
*Equal contribution

[1] University of California, Los Angeles
[2] University of Michigan
[3] Carnegie Mellon University
[4] Columbia University
[5] University of Illinois Urbana-Champaign
z.shi@ucla.edu, qiruijin@umich.edu, huan@huan-zhang.com

Abstract. Branch-and-bound (BaB) is among the most effective techniques for neural network (NN) verification. However, existing works on BaB for NN verification have mostly focused on NNs with piecewise linear activations, especially ReLU networks. In this paper, we develop a general framework, named GenBaB, to conduct BaB on general nonlinearities to verify NNs with general architectures, based on linear bound propagation for NN verification. To decide which neuron to branch, we design a new branching heuristic which leverages linear bounds as shortcuts to efficiently estimate the potential improvement after branching. To decide nontrivial branching points for general nonlinear functions, we propose to pre-optimize branching points, which can be efficiently leveraged during verification with a lookup table. We demonstrate the effectiveness of our GenBaB on verifying a wide range of NNs, including NNs with activation functions such as Sigmoid, Tanh, Sine and GeLU, as well as NNs involving multi-dimensional nonlinear operations such as multiplications in LSTMs and Vision Transformers. Our framework also allows the verification of general nonlinear computation graphs and enables verification applications beyond simple NNs, particularly for AC Optimal Power Flow (ACOPF). GenBaB is part of the latest α,β-CROWN[6], the winner of the 4th and the 5th International Verification of Neural Networks Competition (VNN-COMP 2023 and 2024). Code for reproducing the experiments is available at https://github.com/shizhouxing/GenBaB.

Keywords: Neural network verification · Branch-and-bound · Linear relaxation.

1 Introduction

Neural network (NN) verification aims to formally verify whether a neural network satisfies certain properties, such as safety or robustness properties, prior to its

[6] https://github.com/Verified-Intelligence/alpha-beta-CROWN

deployment in safety-critical applications. Existing NN verifiers typically compute certified bounds for the output given a pre-defined input region and check the desired properties on the output bounds. As computing exact bounds is NP-complete [20], it becomes crucial to relax the bound computation to improve the efficiency. Bound propagation methods [10, 16, 36, 42, 45, 50] have been commonly used, which relax nonlinearities in NNs into linear lower and upper bounds which can be efficiently propagated to finally bound the output of an entire NN.

To obtain tighter verified bounds, Branch-and-Bound (BaB) has been widely utilized [4, 5, 7, 11, 25, 43, 48] in state-of-the-art NN verifiers, where BaB iteratively *branches* the bounds of intermediate neurons, such that subproblems of verification are created and tighter *bounds* can be computed for each subproblem. However, previous works mostly focused on ReLU networks due to the simplicity of ReLU from its piecewise linear nature. Branching a ReLU neuron only requires branching at 0, and it immediately becomes linear in either branch around 0. Conversely, handling NNs with nonlinearities beyond ReLU introduces additional complexity as the convenience of piecewise linearity diminishes. It is important for verifying many models with non-ReLU nonlinearities, including: NNs with non-ReLU activation functions; more complex NNs such as LSTMs [18] and Transformers [40] which have nonlinearities including multiplication and division beyond activation functions; applications such as AC Optimal Power Flow (ACOPF) [14] where the verification problem is defined on a computational graph consisting of a NN and also several nonlinear operators encoding the nonlinear constraints to be verified. Although some previous works have considered BaB for NNs beyond ReLU networks, e.g., [16, 46] considered BaB on networks with S-shaped activations such as Sigmoid, these works still often specialize in specific and relatively simple types of nonlinearities. A more principled framework for handling general nonlinearities is lacking, leaving ample room for further advancements in verifying non-ReLU NNs.

In this paper, we propose **GenBaB**, a principled neural network verification framework with BaB for general nonlinearities. To enable BaB for general nonlinearities beyond ReLU, we first formulate a general BaB framework, and we introduce general branching points, where we may branch at points other than 0 for nonlinear functions, which is needed when the nonlinearity is not piecewise linear around 0. We then propose a new branching heuristic named "Bound Propagation with Shortcuts (BBPS)" for branching general nonlinearities, which carefully leverages the linear bounds from bound propagation as shortcuts to efficiently and effectively estimate the bound improvement from branching a neuron. Moreover, we propose to decide nontrivial branching points by pre-optimizing branching points, according to the tightness of the resulted linear relaxation, and we save the optimized branching points into a lookup table to be efficiently used when verifying an entire NN with different data instances.

We demonstrate the effectiveness of our GenBaB on a variety of networks, including feedforward networks with Sigmoid, Tanh, Sine, or GeLU activations, as well as LSTMs and Vision Transformers (ViTs). These models involve various nonlinearities including S-shaped activations, periodic trigonometric functions, and also multiplication and division which are multi-dimensional nonlinear operations beyond activation functions. We also enable verification on models for the AC Optimal Power Flow (ACOPF) application [14]. GenBaB is generally effective and outperforms existing baselines. The improvement from GenBaB is particularly significant for models involving functions with stronger nonlinearity. For example, on a 4 × 100 network with the Sine activation, GenBaB improves the verification from 4% to 60% instances verified (NNs with the Sine activation have been proposed for neural representations and neural rendering in Sitzmann et al. [37]).

2 Background

The NN verification problem. Let $f : \mathbb{R}^d \rightarrow \mathbb{R}^K$ be a neural network taking input $\mathbf{x} \in \mathbb{R}^d$ and outputting $f(\mathbf{x}) \in \mathbb{R}^K$. Suppose $\mathcal{C}$ is the input region to be verified, and $s : \mathbb{R}^K \rightarrow \mathbb{R}$ is an output specification function, $h : \mathbb{R}^d \mapsto \mathbb{R}$ is the function that combines the NN and the output specification as $h(\mathbf{x}) = s(f(\mathbf{x}))$. NN verification can typically be formulated as verifying if $h(\mathbf{x}) > 0, \forall \mathbf{x} \in \mathcal{C}$ provably holds. A commonly adopted special case is robustness verification given a small input region, where $f(\mathbf{x})$ is a K-way classifier and $h(\mathbf{x}) \coloneqq \min_{i \neq c}\{f_c(\mathbf{x}) - f_i(\mathbf{x})\}$ checks the worst-case margin between the ground-truth class c and any other class i. The input region is often taken as a small ℓ_∞-ball with radius ϵ around a data point $\mathbf{x}_0$, i.e., $\mathcal{C} \coloneqq \{\mathbf{x} \mid \|\mathbf{x} - \mathbf{x}_0\|_\infty \leq \epsilon\}$. This is a succinct and useful problem for provably verifying the robustness properties of a model and also for benchmarking NN verifiers, although there are other NN verification problems beyond robustness [3]. We mainly focus on this setting for its simplicity following prior works.

Linear bound propagation. We develop our GenBaB based on linear bound propagation [47, 50] for computing the verified bounds of each subproblem during the BaB. Linear bound propagation can lower bound $h(\mathbf{x})$ by propagating linear bounds w.r.t. the output of one or more intermediate layers as $h(\mathbf{x}) \geq \sum_i \mathbf{A}_i \hat{\mathbf{x}}_i + \mathbf{c}$ $(\forall \mathbf{x} \in \mathcal{C})$, where $\hat{\mathbf{x}}_i$ $(i \leq n)$ is the output of intermediate layer i in the network $f(\mathbf{x})$ with n layers, $\mathbf{A}_i$ are the coefficients w.r.t. layer i, and $\mathbf{c}$ is a bias term. In the beginning, the linear bound is simply $h(\mathbf{x}) \geq \mathbf{I} \cdot h(\mathbf{x}) + \mathbf{0}$ which is actually an equality. In the bound propagation, $\mathbf{A}_i \hat{\mathbf{x}}_i$ is recursively substituted by the linear bound of $\hat{\mathbf{x}}_i$ w.r.t its input. For simplicity, suppose layer $i-1$ is the input to layer i and $\hat{\mathbf{x}}_i = h_i(\hat{\mathbf{x}}_{i-1})$, where $h_i(\cdot)$ is the computation for layer i. And suppose we have the linear bounds of $\hat{\mathbf{x}}_i$ w.r.t its input $\hat{\mathbf{x}}_{i-1}$ as:

$$\underline{\mathbf{a}}_i \hat{\mathbf{x}}_{i-1} + \underline{\mathbf{b}}_i \leq \hat{\mathbf{x}}_i = h_i(\hat{\mathbf{x}}_{i-1}) \leq \overline{\mathbf{a}}_i \hat{\mathbf{x}}_{i-1} + \overline{\mathbf{b}}_i, \tag{1}$$

with parameters $\underline{\mathbf{a}}_i, \underline{\mathbf{b}}_i, \overline{\mathbf{a}}_i, \overline{\mathbf{b}}_i$ for the linear bounds, and "$\leq$" holds elementwise. Then $\mathbf{A}_i \hat{\mathbf{x}}_i$ can be substituted and lower bounded by $\mathbf{A}_i \hat{\mathbf{x}}_i \geq \mathbf{A}_{i-1} \hat{\mathbf{x}}_{i-1} + \big(\mathbf{A}_{i,+} \underline{\mathbf{b}}_i + \mathbf{A}_{i,-} \overline{\mathbf{b}}_i\big)$, where $\mathbf{A}_{i-1} = \mathbf{A}_{i,+} \underline{\mathbf{a}}_i + \mathbf{A}_{i,-} \overline{\mathbf{a}}_i$, ("+" and "-" in the subscripts denote taking positive and negative elements, respectively). In this way, the linear bounds are propagated from layer i to layer $i-1$. Ultimately, the linear bounds can be propagated to the input of the network $\mathbf{x}$ as $h(\mathbf{x}) \geq \mathbf{A}_0\mathbf{x} + \mathbf{c}$ ($\mathbf{A}_0 \in \mathbb{R}^{1\times d}$), where the input can be viewed as the 0-th layer. Depending on $\mathcal{C}$, this linear bound can be concretized into a lower bound without $\mathbf{x}$. We focus on settings where $\mathcal{C}$ is an ℓ_∞-ball as mentioned above, and thereby we have:

$$\forall \|\mathbf{x} - \mathbf{x}_0\|_\infty \leq \epsilon, \quad \mathbf{A}_0\mathbf{x} + \mathbf{c} \geq \mathbf{A}_0\mathbf{x}_0 - \epsilon\|\mathbf{A}_0\|_1 + \mathbf{c}. \tag{2}$$

To obtain Eq. (1), if h_i is a linear operator, we simply have $\underline{\mathbf{a}}_i\hat{\mathbf{x}}_{i-1} + \underline{\mathbf{b}}_i = \overline{\mathbf{a}}_i\hat{\mathbf{x}}_{i-1} + \overline{\mathbf{b}}_i = h_i(\hat{\mathbf{x}}_{i-1})$ which is h_i itself. Otherwise, linear relaxation is used, which relaxes a nonlinearity and bound the nonlinearity by linear functions. An intermediate bound on $\hat{\mathbf{x}}_{i-1}$ as $\mathbf{l}_{i-1} \leq \hat{\mathbf{x}}_{i-1} \leq \mathbf{u}_{i-1}$ is usually required for the relaxation, which can be obtained by treating the intermediate layer as the output of a network and running additional bound propagation.

3 Method

3.1 Overall Framework

Notations. Although in Section 2, we considered a feedforward NN for simplicity, linear bound propagation has been generalized to NNs with general architectures and general computational graphs [47]. In our work, we also consider a general computational graph $h(\mathbf{x})$ for input region $\mathbf{x} \in \mathcal{C}$. Instead of a feedforward network with *n layers* in Section 2, we consider a computational graph with *n nodes*, where each node i computes some function $h_i(\cdot)$ which may either correspond to a linear layer in the NN or a nonlinearity. We use $\hat{\mathbf{x}}_i$ to denote the output of node i, which may contain many neurons, and we use $\hat{\mathbf{x}}_{i,j}$ to denote the output of the j-th neuron in node i. Intermediate bounds of node i may be needed to relax and bound $h_i(\cdot)$, and we use $\mathbf{l}_{i,j}, \mathbf{u}_{i,j}$ to denote the intermediate lower and upper bound respectively. We use $\mathbf{l}$ and $\mathbf{u}$ to denote all the intermediate lower bounds and upper bounds, respectively, for the entire computational graph.

Overview of GenBaB. Figure 4 illustrates our GenBaB framework. Our GenBaB is a general branch-and-bound framework to handle NNs with general nonlinearities, for NN verification with linear bound propagation. Note that our contributions focus on the *branching* part for general nonlinearities, while *bounding* for individual subdomains during BaB follows existing linear bound propagation which has supported general models [47].

We conduct an initial verification using linear bound propagation before entering BaB. We proceed to BaB only if the initial verification is not sufficient for a successful verification, and we aim to use BaB to enhance the verification for such hard cases. In our BaB, we branch the intermediate bounds of neurons connected to general nonlinearities. We maintain a dynamic pool of intermediate bound domains, $\mathcal{D} = \{(\mathbf{l}^{(i)}, \mathbf{u}^{(i)})\}_{i=1}^m$, where each domain $(\mathbf{l}^{(i)}, \mathbf{u}^{(i)})$ $(1 \leq i \leq m)$ denotes the intermediate bounds of a subproblem in the BaB, $m = |\mathcal{D}|$ is the number of current domains, and initially we have $\mathcal{D} = \{(\mathbf{l}, \mathbf{u})\}$ with the intermediate bounds from the initial verification. Then in each iteration of BaB, we pop a domain from $\mathcal{D}$, and we select a neuron to branch and a branching point between the intermediate bounds of the selected neuron. To support general nonlinearities, we formulate a new and general branching framework in Section 3.2, where we introduce general branching points, in contrast to branching ReLU at 0 only, and we also support more complicated networks architectures where a nonlinearity can involve multiple input nodes or output nodes. To decide nontrivial branching points, in Section 3.3, we propose to pre-optimize the branching points, which aims to produce the tightest linear relaxation after taking the optimized branching point. And in order to decide which neuron we choose to branch, we propose a new branching heuristic in Section 3.4 to estimate the potential improvement for each choice of a branched neuron, where we carefully leverage linear bounds as an efficient shortcut for a more precise estimation.

Each branching step generates new subdomains. For the new subdomains, we update $\mathbf{l}, \mathbf{u}$ for the branched neurons according to the branching points, and the branching decision is also encoded into the bound propagation as additional constraints by Lagrange multipliers following Wang et al. [43]. For each new subdomain, given updated $\mathbf{l}, \mathbf{u}$, we use $V(h, \mathcal{C}, \mathbf{l}, \mathbf{u})$ to denote a new verified bound computed with new intermediate bounds $\mathbf{l}, \mathbf{u}$. Subdomains with $V(h, \mathcal{C}, \mathbf{l}, \mathbf{u}) > 0$ are verified and discarded, otherwise they are added to $\mathcal{D}$ for further branching. We repeat the process until no domain is left in $\mathcal{D}$ and the verification succeeds, or when the timeout is reached and the verification fails. In the implementation, our BaB is batched where many domains are handled in parallel on a GPU with the batch size dynamically tuned to fit the GPU memory.

3.2 Branching for General Nonlinearities

As illustrated in Figure 1, branching for general nonlinearities on general computational graphs is more complicated, in contrast to BaB for ReLU networks. For general nonlinearities, we need to consider branching at points other than 0. In addition, unlike typical activation functions, some nonlinearities may take more than one inputs and thereby have multiple input nodes that can be branched, such as multiplication in LSTM ("$f_{t+1} \odot c_t$" in Figure 1) or Transformers [18, 40]. On general computational graphs, a node can also be followed by multiple nonlinearities, as appeared in LSTMs (such as "c_t" in Figure 1), and then branching the intermediate bounds of this node can affect multiple nonlinearities.

To resolve these challenges, we propose a more general formulation for branching on general computational graphs with general nonlinearities. Each time, we consider branching the intermediate bounds of a neuron j in a node i, namely $[\mathbf{l}_{i,j}, \mathbf{u}_{i,j}]$, if node i is the input of some nonlinearity. We consider branching the concerned neuron into 2 branches with a nontrivial branching point $\mathbf{p}_{i,j}$, as $[\mathbf{l}_{i,j}, \mathbf{u}_{i,j}] \rightarrow [\mathbf{l}_{i,j}, \mathbf{p}_{i,j}],\ [\mathbf{p}_{i,j}, \mathbf{u}_{i,j}]$. Here we consider branching from the perspective of each node i which is the input to at least one nonlinearity and decide if we branch the intermediate bounds $[\mathbf{l}_i, \mathbf{u}_i]$ of this node. This consideration allows us to conveniently support nonlinearities with multiple input nodes or multiple nonlinearities sharing an input node. On the contrary, if we consider branching from the perspective of each nonlinearity, the considered nonlinearity may share

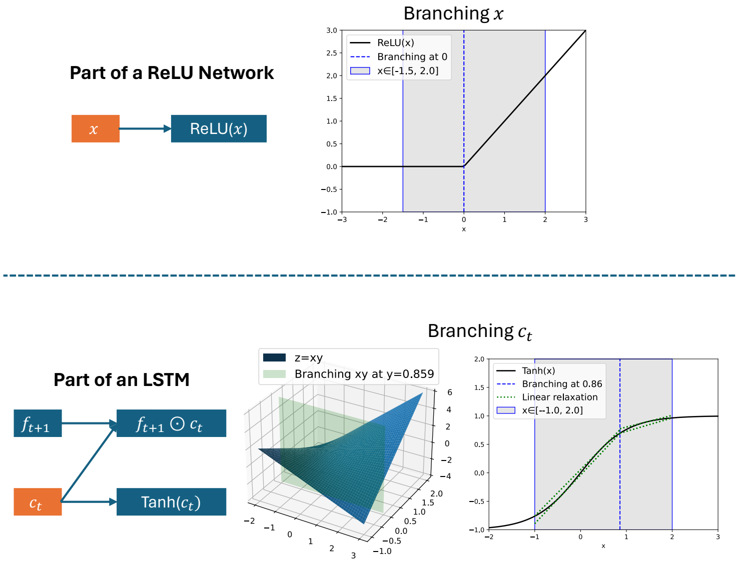
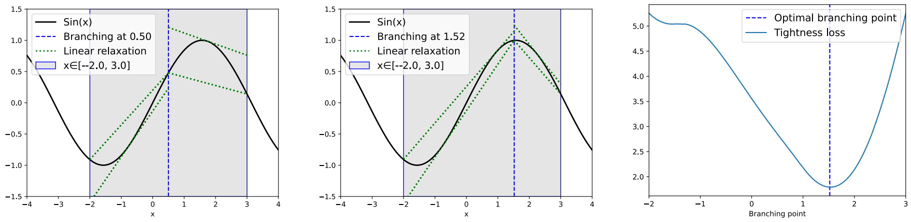


Fig. 1: Illustration of the more complicated nature of branching for general nonlinearities (branching a ReLU activation v.s. branching for nonlinearities in an LSTM). Notations for part of an LSTM follows PyTorch's documentation (`https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html`). Nodes in orange are being branched. For general nonlinearities, branching points can be non-zero (0.86 in the LSTM example here), a nonlinearity can take multiple input nodes ($f_{t+1} \odot c_t$ here), and a node can also be followed by multiple nonlinearities (c_t is followed by a multiplication and also Tanh, and branching c_t affects both two nonlinearities).

some input node with another nonlinearity and thus other nonlinearities can also be affected.

3.3 Where to Branch? New Considerations for General Nonlinear Functions

The more complex nature of general nonlinear functions also brings flexibility on choosing branching points, compared to the ReLU activation where only branching at 0 is reasonable. A straightforward way is to branch in the middle between the intermediate lower and upper bounds, as shown in Figure 2a. However, this can be suboptimal for many nonlinear functions. Intuitively, as tighter linear relaxation can often lead to tighter verified bounds [26, 48], we aim to choose a branching point such that the linear relaxation for both sides after the branching can be as tight as possible. Therefore, we propose to *pre-optimize branching points* for each case of nonlinearity in the model, before actually running BaB on different data instances. We enumerate all pairs of possible intermediate bounds within a certain range with a step size, where we set a small step size which defines the



(a) Branching a Sine activation in the middle.

(b) Branching a Sine at our pre-optimized branching point.

(c) The tightness loss defined in Eq. (3) for different branching points.

Fig. 2: Illustration of branching the intermediate bounds of a neuron connected to the Sine activation [37].

gap between the adjacent enumerated intermediate bounds. And we save the optimized branching points into a lookup table. During verification, for each pair of intermediate bounds we actually encounter, we efficiently query the lookup table and take the branching point for the closest intermediate bound pair in the lookup table (if no valid branching point is obtained from the lookup table, we try branching in the middle instead as a backup). An example of pre-optimized branching points is shown in Figure 2b. We only need to pre-optimize branching points once for each new model, and the produced lookup table can be used on an arbitrary number of data instances, and thus the time cost of the pre-optimization is negligible for the overall verification.

We now formulate the objective of the pre-optimization. For simplicity here, we mainly assume that we have a unary nonlinear function $q(x)$, although our method supports functions with any number of inputs in practice. Suppose the input intermediate bounds for $q(x)$ is $l \le x \le u$, we aim to find a branching point $p = P(l, u)$ such that the overall tightness of the linear relaxation for input range $[l, p]$ and $[p, u]$, respectively, is the best. Suppose the linear relaxation for input range $[l, p]$ is $\underline{a}_1 x + \underline{b}_1 \le q(x) \le \overline{a}_1 x + \overline{b}_1$, and similarly $\underline{a}_2 x + \underline{b}_2 \le q(x) \le \overline{a}_2 x + \overline{b}_2$ for input range $[p, u]$. Following previous works such as Shi et al. [33], we use the integral of the gap between the lower linear relaxation and the upper linear relaxation to measure the tightness (the linear relaxation is considered as tighter when the gap is smaller). We define it as a *tightness loss* $P(q(x), l, u, p)$ for nonlinearity $q(x)$ with input range $[l, u]$ and branching point p:

$$P(q(x), l, u, p) = \int_l^p \Big((\overline{a}_1 x + \overline{b}_1) - (\underline{a}_1 x + \underline{b}_1) \Big) \mathrm{d}x + \int_p^u \Big((\overline{a}_2 x + \overline{b}_2) - (\underline{a}_2 x + \underline{b}_2) \Big) \mathrm{d}x \,, \tag{3}$$

where the parameters for the linear relaxation $(\underline{a}_1, \overline{a}_1, \underline{b}_1, \overline{b}_1, \underline{a}_2, \overline{a}_2, \underline{b}_2, \overline{b}_2)$ all depend on p. We take the best branching point p $(l < p < u)$ which minimizes $P(q(x), l, u, p)$. Figure 2c plots the tightness loss for the Sine activation. This problem can be solved by gradient descent, or an enumeration over a number of potential branching points if the nonlinear function only has one or two inputs.

Moreover, we also support a generalized version of Eq. (3) for nonlinear functions with multiple inputs (such as multiplication involving two inputs), where we use a multiple integral to measure the tightness for multi-dimensional nonlinearities. And when a branched node has multiple nonlinear output nodes, we take the sum for multiple nonlinearities as $\sum_{q\in\mathcal{Q}} P(q(x), l, u, p)$, where $\mathcal{Q}$ is the set of output nonlinearities. As such, our pre-optimized branching points support general computational graphs.

3.4 Which Neuron to Branch? A New Branching Heuristic

Since a NN usually contains many neurons where branching can potentially occur, typically a branching heuristic is used to efficiently decide a neuron to branch, so that the time cost of each BaB iteration is moderate to allow more BaB iterations within the time budget. The branching heuristic is essentially a scoring function for estimating the new verified bound after branching at each neuron, in order to choose a good neuron which potentially leads to a good improvement after the branching. We propose a new branching heuristic to support general nonlinearities.

Specifically, we design a function $\tilde{V}(\mathbf{l}, \mathbf{u}, i, j, k, \mathbf{p}_{i,j})$ which estimates the new bound of the k-th $(1 \leq k \leq 2)$ branch, after branching neuron j in node i using branching points $\mathbf{p}_{i,j}$. We use $B(\mathbf{l}, \mathbf{u}, i, j, k, \mathbf{p}_{i,j})$ to denote the updated intermediate bounds after this branching, and essentially we aim to use $\tilde{V}(\mathbf{l}, \mathbf{u}, i, j, k, \mathbf{p}_{i,j})$ to efficiently estimate $V(h, \mathcal{C}, B(\mathbf{l}, \mathbf{u}, i, j, k, \mathbf{p}_{i,j}))$ which is the actual verified bound after the branching, but it is too costly to directly compute an actual verified bound for each branching option.

Suppose we consider branching a neuron j in node i and we aim to estimate $V(\cdot)$ for each branch k. In the linear bound propagation, when the bounds are propagated to node i, we have:

$$h(\mathbf{x}) \geq \mathbf{A}_{i,j}^{(k)} \hat{\mathbf{x}}_{i,j} + \mathbf{c}^{(k)} \geq V(h, \mathcal{C}, B(\mathbf{l}, i, j, k, \mathbf{p}_{i,j})), \tag{4}$$

where we use $\mathbf{A}_{i,j}^{(k)}$ and $\mathbf{c}^{(k)}$ to denote the parameters in the linear bounds for the k-th branch, and here $\mathbf{c}^{(k)}$ a bias term accumulated on all the neurons. Since we do not update the intermediate bounds except for the branched neurons during BaB for efficiency following Wang et al. [43], branching a neuron in node i only affects the linear relaxation of nonlinear nodes immediately after node i (i.e., output nodes of i). Therefore, $\mathbf{A}_{i,j}^{(k)}$ and $\mathbf{c}^{(k)}$ can be computed by only propagating the linear bounds from the output nodes of i, using previously stored linear bounds, rather than from the final output of $h(\mathbf{x})$.

For a more efficient estimation, instead of propagating the linear bounds towards the input of the network step by step, we propose a new branching heuristic named *Bound Propagation with Shortcuts (BBPS)*, where we use a shortcut to directly propagate the bounds to the input. Specifically, we save the linear bounds of all the potentially branched intermediate nodes during the initial verification. For every neuron j in intermediate node i, we record:

$$\forall \mathbf{x} \in \mathcal{C}, \quad \underline{\hat{\mathbf{A}}}_{ij}\mathbf{x} + \underline{\hat{\mathbf{c}}}_{ij} \leq \hat{\mathbf{x}}_{ij} \leq \overline{\hat{\mathbf{A}}}_{ij}\mathbf{x} + \overline{\hat{\mathbf{c}}}_{ij}, \tag{5}$$

where $\underline{\hat{\mathbf{A}}}_{ij}, \underline{\hat{\mathbf{c}}}_{ij}, \overline{\hat{\mathbf{A}}}_{ij}, \overline{\hat{\mathbf{c}}}_{ij}$ are parameters for the linear bounds. These are obtained when linear bound propagation is used for computing the intermediate bounds $[\mathbf{l}_{i,j}, \mathbf{u}_{i,j}]$ and the linear bounds are propagated to the input $\mathbf{x}$. We then use Eq. (5) to compute a lower bound for $\mathbf{A}_{i,j}^{(k)}\hat{\mathbf{x}}_{i,j} + \mathbf{c}^{(k)}$:

$$\mathbf{A}_{i,j}^{(k)}\hat{\mathbf{x}}_{i,j} + \mathbf{c}^{(k)} \geq (\mathbf{A}_{i,j,+}^{(k)}\underline{\hat{\mathbf{A}}}_{ij} + \mathbf{A}_{i,j,-}^{(k)}\overline{\hat{\mathbf{A}}}_{ij})\mathbf{x} + \mathbf{A}_{i,j,+}^{(k)}\underline{\hat{\mathbf{c}}}_{ij} + \mathbf{A}_{i,j,-}^{(k)}\overline{\hat{\mathbf{c}}}_{ij} + \mathbf{c}^{(k)}. \quad (6)$$

The right-hand-side can be concretized by Eq. (2) to serve as an approximation for $V(\cdot)$ after the branching. In this way, the linear bounds are directly propagated from node i to input $\mathbf{x}$ and concretized using a shortcut. We thereby take the concretized bound as $\tilde{V}(\mathbf{l}, \mathbf{u}, i, j, k, \mathbf{p}_{i,j})$ for our BBPS heuristic score.

This computation is efficient, and it does not affect the time complexity of BaB as the time complexity is mainly dominated by the bound computation after each branching. Our branching heuristic is also generally formulated. We leverage updates on the linear relaxation of any nonlinearity, and general branching points and general number of inputs nodes are supported when we update the linear relaxation. Node i can also have multiple nonlinear output nodes, as we accumulate the linear bounds propagated from all the output nodes to produce Eq. (4).

Comparison to branching heuristics in previous works. Existing branching heuristics from previous works [4, 5, 7, 25] are more restrictive, as they mostly focused on branching ReLU neurons with a fixed branching point (0 for ReLU) and their heuristic is specifically formulated for ReLU, unlike our general formulation above. Even if we directly generalize their branching heuristic to support general nonlinearities, we also empirically find they are often not precise enough for general nonlinearities due to their more aggressive approximation. In the existing BaBSR heuristic originally for ReLU networks [4], they essentially propagate the bounds only to the node before the branched one with an early stop, and they then ignore the coefficients ($\mathbf{A}_{i-1,j}^{(k)}$ for a feedforward NN) without propagating further. In contrast, in our BBPS heuristic, we carefully utilize a shortcut to propagate the bounds to the input as Eq. (6) rather than discard linear terms early. Therefore, we expect our BBPS heuristic to be more precise and effective.

4 Experiments

4.1 Settings

Implementation and additional experimental details are provided in Appendix D.

Models and Data. We focus on verifying NNs with nonlinearities beyond ReLU, and we experiment on models with various nonlinearities as shown in Table 1. We mainly consider the commonly used ℓ_∞ robustness verification specification on image classification. We use the term *instance* to refer to a data example along with the corresponding verification specification. We adopt some MNIST [24] models with Sigmoid and Tanh activation functions from previous works [29, 35, 36], along with their data instances. Besides, to test our method on more models with various nonlinearities using a consistent training setting for all the models, we train many new models with various nonlinearities on CIFAR-10 [23] by PGD adversarial training [27], using an ℓ_∞ perturbation with $\epsilon = 1/255$ in both training and verification. The models we train on CIFAR-10 include models with Sigmoid, Tanh, Sine, and GeLU activation functions, respectively, as well as LSTM [18] and ViT [8]. We adopt PGD adversarial training, because NNs trained without robust training are known to be highly vulnerable to tiny adversarial perturbations [13, 38] and formal verification is not possible unless ϵ is much smaller. For these CIFAR-10 models, we first run vanilla CROWN [47, 49] (linear bound propagation without optimized linear relaxation [26, 48] or BaB [43, 48]), to remove instances which are too easy where vanilla CROWN already succeeds. We also remove instances where PGD attack succeeds, as such instances are impossible to verify. We only retain the first 100 instances if there are more instances left. We set a timeout of 300 seconds for our BaB in all these experiments. In addition, we adopt an NN verification benchmark for verifying properties in the Machine Learning for AC Optimal Power Flow (ML4ACOPF) problem [14][7] which is beyond robustness verification. In the Appendix, we have results on additional models: a ResNet model [15] in Appendix C.3; models with larger $\epsilon = 2/255$ and $\epsilon = 8/255$ in Appendix C.4; and a ReLU model in Appendix C.5, included for completeness.

Baselines. We compare our GenBaB with the previous α,β-CROWN which did not support BaB on non-ReLU nonlinearities. We also compare with several other baselines, including DeepPoly [36], PRIMA [29], VeriNet [16], PROVER [31], DeepT [1], Wu et al. [46], Wei et al. [44], on the models they support, respectively. Among these baselines, only VeriNet and Wu et al. [46] support BaB on Sigmoid or Tanh models, and none of the baseline supports BaB on general nonlinearities. While the original BaBSR heuristic in Bunel et al. [4] only supported ReLU networks, we also implemented a generalized version of BaBSR for nonlinearities beyond ReLU for an empirical comparison in Table 3, based on the difference in treating the linear term discussed in Section 3.4.

Table 1: List of models with various nonlinearities in our experiments.

Model	Nonlinearities in the model
Feedforward	sigmoid, tanh, sin, GeLU
LSTM	sigmoid, tanh, xy
ViT with ReLU	ReLU, xy, x/y, x^2, $\sqrt{x}$, $\exp(x)$
ML4ACOPF	ReLU, sigmoid, sin, xy, x^2

4.2 Main Results

Experiments on Sigmoid and Tanh networks for MNIST. We first experiment on Sigmoid networks and Tanh networks for MNIST and show the results in Table 2.

[7] Benchmark: `https://github.com/AI4OPT/ml4acopf_benchmark`.

Table 2: Number of verified instances out of the first 100 test examples on MNIST for several Sigmoid networks and Tanh networks along with their ϵ. The settings are the same as those in PRIMA [29]. "$L \times W$" in the network names denote a fully-connected NN with L layers and W hidden neurons in each layer. The upper bounds in the last row are computed by PGD attack [27], as a sound verification should not verify instances where PGD can successfully find counterexamples.

	Sigmoid Networks				Tanh Networks			
Method	6×100	6×200	9×100	ConvSmall	6×100	6×200	9×100	ConvSmall
	$\epsilon=0.015$	$\epsilon=0.012$	$\epsilon=0.015$	$\epsilon=0.014$	$\epsilon=0.006$	$\epsilon=0.002$	$\epsilon=0.006$	$\epsilon=0.005$
DeepPoly[ab]	30	43	38	30	38	39	18	16
PRIMA[a]	53	73	56	51	61	68	52	30
VeriNet[c]	65	81	56	-	31	30	16	-
Marabou [46][?]	65	75	96[?]	63	-	-	-	-
Vanilla CROWN[b]	53	63	49	65	18	24	44	55
α,β-CROWN (w/o BaB)	62	81	**62**	84	**65**	72	58	69
GenBaB (ours)	**71**	**83**	**62**	**92**	**65**	**78**	**59**	**75**
Upper bound	93	99	92	97	94	97	96	98

[a] Results for DeepPoly and PRIMA are directly from Müller et al. [29].

[b] While DeepPoly and CROWN are thought to be equivalent on ReLU networks [29], these two works adopt different relaxation for Sigmoid and Tanh, which results in different results here.

[c] Results for VeriNet are obtained by running the tool (`https://github.com/vas-group-imperial/VeriNet`) by ourselves. VeriNet depends on the FICO Xpress commercial solver which requires a license for models that are relatively large. FICO Xpress declined the request we submitted for an academic license due to the lack of a course tutor. Thus, results on ConvSmall models are not available.

[?] We found that the result Wu et al. [46] reported on the Sigmoid 9×100 model exceeds the upper bound by PGD attack ($96 > 92$), and thus the result tends to be not fully valid (also reported in Zhou et al. [53]).

On 6 out of the 8 models, our GenBaB is able to verify more instances over α,β-CROWN without BaB and also outperforms all the non-CROWN baselines. We find that improving on Sigmoid 9×100 and Tanh 6×100 networks by BaB is harder, as the initial bounds are typically too loose on the unverifiable instances before BaB, possibly due to these models being trained without robustness consideration.

Experiments on feedforward NNs with various activation functions for CIFAR-10. In Table 3, we show results for models on CIFAR-10. On all the models, GenBaB verifies much more instances compared to α,β-CROWN without BaB. We also conduct ablation studies to investigate the effect of our BBPS heuristic and branching points, with results shown in the last three rows of Table 3. Comparing "Base BaB" and " + BBPS", on most of the models, we find that our BBPS heuristic significantly improves over directly generalizing the BaBSR heuristic [4] used in "Base BaB". Comparing "+ BBPS" and "+ BBPS, + pre-optimized", we find that our pre-optimized branching points achieve a noticeable improvement on many models over always branching in the middle. The results demonstrate the

Table 3: Number of verified instances out of 100 filtered instances on CIFAR-10 with $\epsilon = 1/255$ for feedforward NNs with various activation functions. The last three rows contain results for the ablation study, where "Base BaB" does not use our BBPS heuristic or pre-optimized branching points, but it uses a generalized BaBSR heuristic [4] and always branches intermediate bounds in the middle.

Method	Sigmoid Networks				Tanh Networks		Sine Networks			GeLU Networks		
	4×100	4×500	6×100	6×200	4×100	6×100	4×100	4×200	4×500	4×100	4×200	4×500
PRIMA[a]	0	0	0	0	0	0	-	-	-	-	-	-
Vanilla CROWN[b]	0	0	0	0	0	0	0	0	0	0	0	0
α,β-CROWN w/o BaB[c]	28	16	43	39	25	6	4	2	4	44	33	27
GenBaB (ours)	**58**	**24**	**64**	**50**	**49**	**10**	**60**	**35**	**22**	**82**	**65**	**39**
Ablation Studies												
Base BaB	34	19	44	41	34	8	9	8	7	64	54	39
+ BBPS	57	24	63	49	48	10	56	34	21	74	59	36
+ BBPS, + pre-optimized	58	24	64	50	49	10	60	35	22	82	65	39

[a] Results for PRIMA are obtained by running ERAN (`https://github.com/eth-sri/eran`) which contains PRIMA. PRIMA does not support Sine or GeLU activations.

[b] We have extended its support to GeLU, as discussed in Appendix B.3.

[c] We have extended optimizable linear relaxation in α,β-CROWN to Sine and GeLU, as discussed in Appendix B.

effectiveness of our GenBaB with our BBPS heuristic and pre-optimized branching points. GenBaB also exhibits much better scalability, where we compare the model size each method can handle w.r.t. a threshold on the number of verified instances. For example, if our threshold is 20 verified instances, GenBaB can at least scale to 4×500 (22 instances verified) while α,β-CROWN w/o BaB cannot even scale to 4×100 (likely even much smaller, as only 4 instances are verified for 4×100).

For PRIMA and vanilla CROWN, as we only use relatively hard instances for verification here, these two methods are unable to verify any instance in this experiment. For VeriNet, all the models here are too large without a license for the FICO Xpress solver (an academic license was not available to us as mentioned in Table 2); we have not obtained the code to run Wu et al. [46] on these models. Thus, we do not include the results for VeriNet or Wu et al. [46].

Experiments on LSTMs. Next, we experiment on LSTMs containing more complex nonlinearities, including both Sigmoid and Tanh activations, as well as multiplication as $\text{sigmoid}(x)\tanh(y)$ and $\text{sigmoid}(x)y$. We compare with PROVER [31] which is a specialized verifier for RNNs and it outperforms earlier works [21]. While there are other works on verifying RNN and LSTM, such as [9, 28, 30], we have not obtained their code, and we also make orthogonal contributions compared to them on improving the relaxation for RNN verification which can also be combined with our BaB. We take the hardest model, an LSTM for MNIST, from the main experiments of PROVER (other models can be verified by PROVER on more than 90% instances and are thus omitted), where each 28×28 image is sliced into 7 frames for LSTM. We also have two LSTMs trained by

Table 4: Number of verified instances out of 100 instances on LSTMs and ViTs. The MNIST model is from PROVER [31] with $\epsilon = 0.01$, and the CIFAR-10 models are trained by ourselves with $\epsilon = 1/255$. "LSTM-7-32" indicates an LSTM with 7 input frames and 32 hidden neurons, similar for the other two models. "ViT-L-H" stands for L layers and H heads. Some models have fewer than 100 instances, after filtering out easy or impossible instances, as shown in "upper bounds". Results for PROVER are obtained by running the tool (`https://github.com/eth-sri/prover`). Results for DeepT are obtained by running the tool (`https://github.com/eth-sri/DeepT`). PROVER and DeepT specialize in RNNs and ViTs, respectively.

Method	MNIST Model	CIFAR-10 Models					
	LSTM-7-32	LSTM-4-32	LSTM-4-64	ViT-1-3	ViT-1-6	ViT-2-3	ViT-2-6
PROVER	63	8	3	-	-	-	-
DeepT	-	-	-	0	1	0	1
α,β-CROWN w/o BaB	82	16	9	1	3	11	7
GenBaB (ours)	**84**	**20**	**14**	**49**	**72**	**65**	**56**
Upper bound	98	100	100	67	92	72	69

ourselves on CIFAR-10, where we linearly map each 32×32 image into 4 patches as the input tokens, similar to ViTs with patches [8]. Table 4 shows the results. α,β-CROWN without BaB can already outperform PROVER with specialized relaxation for RNN and LSTM. Our GenBaB outperforms both PROVER and α,β-CROWN without BaB.

Experiments on ViTs. We also experiment on ViTs which contain more other nonlinearities, as shown in Table 1. For ViTs, we compare with DeepT [1] which is specialized for verifying Transformers without BaB. We show the results in Table 4, where our methods outperform DeepT, and our GenBaB effectively improves the verification. Moreover, in Appendix C.2, we compare with Wei et al. [44] which supports verifying attention networks but not the entire ViT, and we experiment on models from Wei et al. [44] and find that our GenBaB also outperforms Wei et al. [44].

Experiments on ML4ACOPF. Finally, we experiment on models for the Machine Learning for AC Optimal Power Flow (ML4ACOPF) problem [14], and we adopt the ML4ACOPF neural network verification benchmark, a standardized benchmark in the 2023 International Verification of Neural Networks Competition (VNN-COMP'23). The benchmark consists of a NN with power demands as inputs, and the output of the NN gives an operation plan of electric power plants. Then, the benchmark aims to check for a few nonlinear constraint violations of this plan, such as power generation and balance constraints. These constraints, as part of the computational graph to verify, involve many nonlinearities including Sine, Sigmoid, multiplication, and square function. Our work is the first to support this verification problem. Among the 23 benchmark instances, PGD attack finds a counterexample on one instance, and our GenBaB verifies all the remaining 22

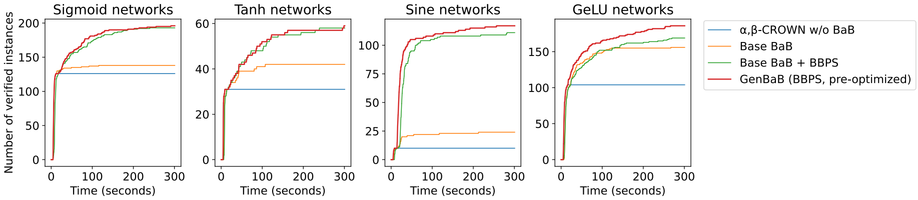


Fig. 3: Total number of verified instances against running time threshold on feedforward networks for CIFAR-10 with various activation functions. "Base BaB" means that in the most basic BaB setting, we use a generalized BaBSR heuristic and always branch in the middle point of intermediate bounds. "Base + BBPS" uses our BBPS heuristic. Our full GenBaB uses both BBPS and pre-optimized branching points.

instances. Only 16 instances can be verified if BaB is disabled. This experiment shows a more practical application of our work and further demonstrates the effectiveness of our framework.

4.3 Time Cost

Table 5: Time cost of pre-optimizing the branching points for models with different nonlinearities. We only need to run pre-optimization once for each model. The cost is thus negligible as we have many data instances to verify.

Model	Sigmoid	Tanh	Sin	GeLU	LSTM	ViT
Time cost (seconds)	49	55	112	82	761	746

In this section, we analyze the time cost of our method. Our GenBaB aims to verify additional instances which cannot be verified without BaB, for models with general nonlinearities. Average time is not a suitable metric here [43], because different methods verify different numbers of instances, and a stronger verifier which can verify more hard instances requiring more time cost will naturally have a larger average time compared to a weak verifier which can only verify the easiest instances quickly. Instead, we plot the number of verified instances against different time thresholds in Figure 3. Such plots, a.k.a. "cactus plots", are commonly adopted in previous works [3, 43]. The plots show that our GenBaB enables the verification of more instances as more time budget is allowed for BaB. While the baseline without BaB can verify some relatively easy instances within a short running time (GenBaB can also verify these easy instances during the initial verification with the same time cost if BaB is not needed), the baseline cannot utilize the remaining time budget to verify more instances. Time cost for LSTM and ViT models are shown in Appendix C.1. In Table 5, we also show the time cost of pre-optimizing the branching points. Overall, the pre-optimization can be done quickly. As explained in Section 3.3, this time cost is negligible for the overall verification, as we only need to run the pre-optimization once for each model and the produced lookup table of branching points can be used to verify an arbitrary number of instances.

4.4 Comparison with BaB on ReLU for Models Containing ReLU

Table 6: Number of verified instances by GenBaB compared to BaB on ReLU only, for certain models containing ReLU. For BaB on ReLU only, we show results for two different branching heuristic (FSB [7] and our BBPS).

Method	ViT-1-3	ViT-1-6	ViT-2-3	ViT-2-6	ML4ACOPF
BaB on ReLU only (FSB)	47	70	63	55	18
BaB on ReLU only (BBPS)	47	70	63	55	21
GenBaB	**49**	**72**	**65**	**56**	**22**
Upper bound	67	92	72	69	22

Although our focus is on BaB on non-ReLU nonlinearities, some of the relatively complicated models involved in our experiments still contain ReLU, and thus we compare our GenBaB with BaB on ReLU only for these models. Specifically, only ViT and ML4ACOPF models in our experiments contain ReLU, although they also contain many other nonlinearities. We show results in Table 6. The results demonstrate that our GenBaB which branches on general nonlinearities outperforms BaB on ReLU only for the models containing ReLU. And many other models with other nonlinearities do not even contain ReLU. Threfore, our GenBaB is important for the BaB on models with general nonlinearities. We also observe that when we only conduct BaB on ReLU for ML4ACOPF, our BBPS heuristic also outperforms the FSB heuristic [7] which is the default branching heuristic adopted by α,β-CROWN for ReLU (FSB is improved from BaBSR [4] and enhanced with a filtering mechanism to compute actual verified bounds for a shortlist of neurons), and our GenBaB which considers all the nonlinearities can verify more instances (all the 22 possible instances are verified) compared to BaB on ReLU only.

5 Related Work

Due to the NP-complete nature of the NN verification [20], linear bound propagation [36, 45, 50] has been proposed to relax nonlinearities in a NN network using linear lower and upper bounds and then propagate the linear relationship between different layers, so that tractable output bounds can be efficiently computed for much larger NNs with various architectures [2, 21, 33, 47]. A limitation of using linear bound propagation only is that the linear relaxation, which depends on the output bounds of intermediate layers, can often have a limited tightness as the intermediate bounds gradually become looser in later layers. Therefore, branch-and-bound (BaB) has been an essential technique in state-of-the-art verifiers [5, 7, 17, 22, 25, 32, 41, 43, 46, 48] leveraging linear relaxation, which iteratively branches the intermediate bounds of selected neurons to enable tight linear relaxation and compute tighter output bounds. However, most of the existing works on the BaB for NN verification have focused on ReLU networks with the piecewise-linear ReLU activation function, and they are not directly applicable to NNs with nonlinearities beyond ReLU. Nevertheless, there are several previous works on the BaB for verifying NNs with nonlinearities other than ReLU. Henriksen and Lomuscio [16] conducted BaB on Sigmoid and Tanh networks, but their framework depends on a commercial LP solver which has been argued as less effective than recent NN verification methods using linear bound propagation [43]. Besides, Wu et al. [46] studied verifying Sigmoid networks with counter-example-guided abstraction refinement. These works have focused on S-shaped activations such as Sigmoid and Tanh, and there still lacks a general framework supporting general nonlinearities beyond a particular type of activation functions, which we address in this paper.

Orthogonal to our contributions on BaB for general nonlinearities, many works studied the verification of NNs with various nonlinearities without considering BaB, by improving the linear relaxation or extending the support of verification to various architectures or specifications: Sigmoid and Tanh networks [2, 6, 50], RNNs and LSTMs [9, 21, 28, 31, 39, 51], Transformers [1, 34, 44, 52], general computational graphs [47], and specifications on activation patterns instead of input [12]. Contributions along these lines may be combined with our work, as our BaB is independent from the underlying linear relaxation adopted. Moreover, some works improved the branching heuristic for verifying ReLU networks: Lu and Mudigonda [25] proposed to use a Graph Neural Network for the branching heuristic; De Palma et al. [7] proposed Filtered Smart Branching (FSB) which filters initial candidates by a heuristic score and then uses a more accurate bound computation to select an optimal neuron from a shortlist; Ferrari et al. [11] considered the effect of a tighter multi-neuron relaxation in the branching heuristic. These insights originally for ReLU networks may inspire future improvement of the BaB for general nonlinearities.

6 Conclusion

To conclude, we propose a general BaB framework for NN verification involving general nonlinearities in general computational graphs. We also propose a new branching heuristic for deciding branched neurons and a pre-optimization procedure for deciding branching points. Experiments on verifying NNs with various nonlinearities demonstrate the effectiveness of our method.

Acknowledgments

This project is supported in part by NSF 2048280, 2331966, 2331967 and ONR N00014-23-1-2300:P00001. Huan Zhang is supported in part by the AI2050 program at Schmidt Sciences (Grant #G-23-65921).

Bibliography

[1] Bonaert, G., Dimitrov, D.I., Baader, M., Vechev, M.: Fast and precise certification of transformers. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 466–481 (2021)

[2] Boopathy, A., Weng, T., Chen, P., Liu, S., Daniel, L.: Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks. In: The Thirty-Third AAAI Conference on Artificial Intelligence, pp. 3240–3247 (2019), `https://doi.org/10.1609/aaai.v33i01.33013240`

[3] Brix, C., Bak, S., Liu, C., Johnson, T.T.: The fourth international verification of neural networks competition (vnn-comp 2023): Summary and results. arXiv preprint arXiv:2312.16760 (2023)

[4] Bunel, R., Mudigonda, P., Turkaslan, I., Torr, P., Lu, J., Kohli, P.: Branch and bound for piecewise linear neural network verification. Journal of Machine Learning Research **21**(2020) (2020)

[5] Bunel, R., Turkaslan, I., Torr, P.H.S., Kohli, P., Mudigonda, P.K.: A unified view of piecewise linear neural network verification. In: Advances in Neural Information Processing Systems, pp. 4795–4804 (2018)

[6] Choi, S.W., Ivashchenko, M., Nguyen, L.V., Tran, H.D.: Reachability analysis of sigmoidal neural networks. ACM Transactions on Embedded Computing Systems (2023)

[7] De Palma, A., Bunel, R., Desmaison, A., Dvijotham, K., Kohli, P., Torr, P.H., Kumar, M.P.: Improved branch and bound for neural network verification via lagrangian decomposition. arXiv preprint arXiv:2104.06718 (2021)

[8] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., Houlsby, N.: An image is worth 16x16 words: Transformers for image recognition at scale. In: International Conference on Learning Representations (2021)

[9] Du, T., Ji, S., Shen, L., Zhang, Y., Li, J., Shi, J., Fang, C., Yin, J., Beyah, R., Wang, T.: Cert-rnn: Towards certifying the robustness of recurrent neural networks. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, p. 516–534, CCS '21 (2021), ISBN 9781450384544, `https://doi.org/10.1145/3460120.3484538`

[10] Dvijotham, K., Stanforth, R., Gowal, S., Mann, T.A., Kohli, P.: A dual approach to scalable verification of deep networks. In: Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence, UAI 2018, Monterey, California, USA, August 6-10, 2018, pp. 550–559 (2018)

[11] Ferrari, C., Mueller, M.N., Jovanović, N., Vechev, M.: Complete verification via multi-neuron relaxation guided branch-and-bound. In: International Conference on Learning Representations (2021)

[12] Geng, C., Le, N., Xu, X., Wang, Z., Gurfinkel, A., Si, X.: Towards reliable neural specifications. In: International Conference on Machine Learning, pp. 11196–11212, PMLR (2023)

[13] Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. In: International Conference on Learning Representations (2015)

[14] Guha, N., Wang, Z., Wytock, M., Majumdar, A.: Machine learning for ac optimal power flow. arXiv preprint arXiv:1910.08842 (2019)

[15] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016), `https://doi.org/10.1109/CVPR.2016.90`

[16] Henriksen, P., Lomuscio, A.: Efficient neural network verification via adaptive refinement and adversarial search. In: ECAI 2020, pp. 2513–2520, IOS Press (2020)

[17] Henriksen, P., Lomuscio, A.: Deepsplit: An efficient splitting method for neural network verification via indirect effect analysis. In: IJCAI, pp. 2549–2555 (2021)

[18] Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8), 1735–1780 (1997)

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

[20] Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient smt solver for verifying deep neural networks. In: International Conference on Computer Aided Verification, pp. 97–117 (2017)

[21] Ko, C., Lyu, Z., Weng, L., Daniel, L., Wong, N., Lin, D.: POPQORN: quantifying robustness of recurrent neural networks. In: International Conference on Machine Learning, Proceedings of Machine Learning Research, vol. 97, pp. 3468–3477 (2019)

[22] Kouvaros, P., Lomuscio, A.: Towards scalable complete verification of relu neural networks via dependency-based branching. In: IJCAI, pp. 2643–2650 (2021)

[23] Krizhevsky, A., Hinton, G., et al.: Learning multiple layers of features from tiny images. Technical Report TR-2009 (2009)

[24] LeCun, Y., Cortes, C., Burges, C.: Mnist handwritten digit database. ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist **2** (2010)

[25] Lu, J., Mudigonda, P.: Neural network branching for neural network verification. In: Proceedings of the International Conference on Learning Representations (ICLR 2020), Open Review (2020)

[26] Lyu, Z., Ko, C., Kong, Z., Wong, N., Lin, D., Daniel, L.: Fastened CROWN: tightened neural network robustness certificates. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, pp. 5037–5044 (2020)

[27] Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A.: Towards deep learning models resistant to adversarial attacks. In: International Conference on Learning Representations (2018)

[28] Mohammadinejad, S., Paulsen, B., Deshmukh, J.V., Wang, C.: Diffrnn: Differential verification of recurrent neural networks. In: Formal Modeling and Analysis of Timed Systems: 19th International Conference, FORMATS 2021, Paris, France, August 24–26, 2021, Proceedings 19, pp. 117–134, Springer (2021)

[29] Müller, M.N., Makarchuk, G., Singh, G., Püschel, M., Vechev, M.: Prima: general and precise neural network certification via scalable convex hull approximations. Proceedings of the ACM on Programming Languages **6**(POPL), 1–33 (2022)

[30] Paulsen, B., Wang, C.: Linsyn: Synthesizing tight linear bounds for arbitrary neural network activation functions. In: Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part I, pp. 357–376, Springer (2022)

[31] Ryou, W., Chen, J., Balunovic, M., Singh, G., Dan, A., Vechev, M.: Scalable polyhedral verification of recurrent neural networks. In: International Conference on Computer Aided Verification, pp. 225–248 (2021)

[32] Shi, Z., Wang, Y., Zhang, H., Kolter, J.Z., Hsieh, C.J.: Efficiently computing local lipschitz constants of neural networks via bound propagation. Advances in Neural Information Processing Systems **35**, 2350–2364 (2022)

[33] Shi, Z., Zhang, H., Chang, K., Huang, M., Hsieh, C.: Robustness verification for transformers. In: International Conference on Learning Representations (2020)

[34] Shi, Z., Zhang, H., Chang, K.W., Huang, M., Hsieh, C.J.: Robustness verification for transformers. In: International Conference on Learning Representations (2019)

[35] Singh, G., Ganvir, R., Püschel, M., Vechev, M.T.: Beyond the single neuron convex barrier for neural network certification. In: Advances in Neural Information Processing Systems, pp. 15072–15083 (2019)

[36] Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. Proceedings of the ACM on Programming Languages **3**(POPL), 41 (2019)

[37] Sitzmann, V., Martel, J., Bergman, A., Lindell, D., Wetzstein, G.: Implicit neural representations with periodic activation functions. Advances in Neural Information Processing Systems **33**, 7462–7473 (2020)

[38] Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I.J., Fergus, R.: Intriguing properties of neural networks. In: International Conference on Learning Representations (2014)

[39] Tran, H.D., Choi, S.W., Yang, X., Yamaguchi, T., Hoxha, B., Prokhorov, D.: Verification of recurrent neural networks with star reachability. In: Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control, pp. 1–13 (2023)

[40] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Advances in Neural Information Processing Systems 30: Annual Conference on Neural

Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA, pp. 5998–6008 (2017)

[41] Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: Advances in Neural Information Processing Systems, pp. 6369–6379 (2018)

[42] Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: 27th {USENIX} Security Symposium ({USENIX} Security 18), pp. 1599–1614 (2018)

[43] Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C.J., Kolter, J.Z.: Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. Advances in Neural Information Processing Systems **34**, 29909–29921 (2021)

[44] Wei, D., Wu, H., Wu, M., Chen, P.Y., Barrett, C., Farchi, E.: Convex bounds on the softmax function with applications to robustness verification. In: International Conference on Artificial Intelligence and Statistics, pp. 6853–6878, PMLR (2023)

[45] Wong, E., Kolter, J.Z.: Provable defenses against adversarial examples via the convex outer adversarial polytope. In: International Conference on Machine Learning, Proceedings of Machine Learning Research, vol. 80, pp. 5283–5292 (2018)

[46] Wu, H., Tagomori, T., Robey, A., Yang, F., Matni, N., Pappas, G., Hassani, H., Pasareanu, C., Barrett, C.: Toward certified robustness against real-world distribution shifts. arXiv preprint arXiv:2206.03669 (2022)

[47] Xu, K., Shi, Z., Zhang, H., Wang, Y., Chang, K., Huang, M., Kailkhura, B., Lin, X., Hsieh, C.: Automatic perturbation analysis for scalable certified robustness and beyond. In: Advances in Neural Information Processing Systems (2020)

[48] Xu, K., Zhang, H., Wang, S., Wang, Y., Jana, S., Lin, X., Hsieh, C.: Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers. In: International Conference on Learning Representations (2021)

[49] Zhang, H., Chen, H., Xiao, C., Gowal, S., Stanforth, R., Li, B., Boning, D.S., Hsieh, C.: Towards stable and efficient training of verifiably robust neural networks. In: International Conference on Learning Representations (2020)

[50] Zhang, H., Weng, T., Chen, P., Hsieh, C., Daniel, L.: Efficient neural network robustness certification with general activation functions. In: Advances in Neural Information Processing Systems, pp. 4944–4953 (2018)

[51] Zhang, Y., Du, T., Ji, S., Tang, P., Guo, S.: Rnn-guard: Certified robustness against multi-frame attacks for recurrent neural networks. arXiv preprint arXiv:2304.07980 (2023)

[52] Zhang, Y., Shen, L., Guo, S., Ji, S.: Galileo: General linear relaxation framework for tightening robustness certification of transformers. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 38, pp. 21797–21805 (2024)

[53] Zhou, X., Xu, H., Xu, A., Shi, Z., Hsieh, C.J., Zhang, H.: Testing neural network verifiers: A soundness benchmark with hidden counterexamples. arXiv preprint arXiv:2412.03154 (2024)

A Additional Illustration

Figure 4 illustrates our proposed framework.

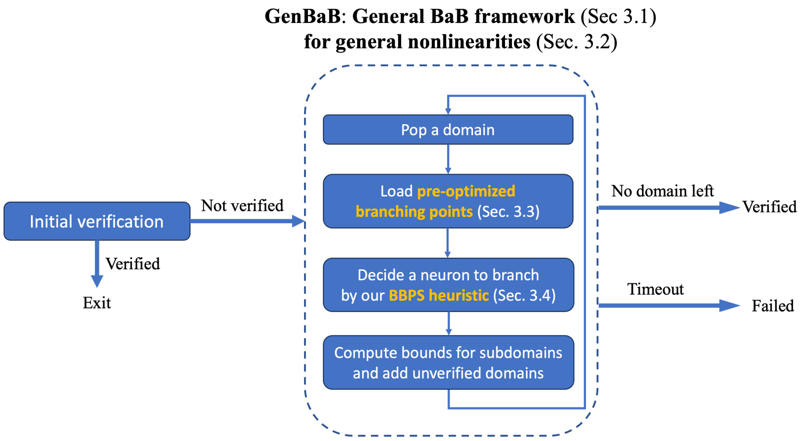

GenBaB: General BaB framework (Sec 3.1)
for general nonlinearities (Sec. 3.2)
Initial verification
Not verified
Verified
Exit
Pop a domain
Load pre-optimized branching points (Sec. 3.3)
Decide a neuron to branch by our BBPS heuristic (Sec. 3.4)
Compute bounds for subdomains and add unverified domains
No domain left
Verified
Timeout
Failed

Fig. 4: Illustration of our new GenBaB framework summarized in Section 3.1.

B Additional Optimizable Linear Relaxation

In this section, we derive new optimizable linear relaxation for nonlinearities including multiplication, sine, and GeLU, which are not originally supported in α,β-CROWN for optimizable linear relaxation.

B.1 Optimizable Linear Relaxation for Multiplication

For each elementary multiplication xy where $x \in [\underline{x}, \overline{x}]$, $y \in [\underline{y}, \overline{y}]$ are the intermediate bounds for x and y, we aim to relax and bound xy as:

$$\forall x \in [\underline{x}, \overline{x}], y \in [\underline{y}, \overline{y}], \quad \underline{a}x + \underline{b}y + \underline{c} \leq xy \leq \overline{a}x + \overline{b}y + \overline{c}, \tag{7}$$

where $\underline{a}, \underline{b}, \underline{c}, \overline{a}, \overline{b}, \overline{c}$ are parameters in the linear bounds. Shi et al. [34] derived optimal parameters that minimize the gap between the relaxed upper bound and the relaxed lower bound:

$$\begin{aligned}&\mathop{\arg\min}_{\underline{a},\underline{b},\underline{c},\overline{a},\overline{b},\overline{c}} \int_{x\in[\underline{x},\overline{x}]} \int_{y\in[\underline{y},\overline{y}]} (\overline{a}x + \overline{b}y + \overline{c}) - (\underline{a}x + \underline{b}y + \underline{c}) \\ &\text{s.t. Eq. (7) holds.}\end{aligned} \tag{8}$$

However, the optimal parameters they found only guarantee that the linear relaxation is optimal for this node, but not the final bounds after conducting a bound propagation on the entire NN. Therefore, we aim to make these parameters optimizable to tighten the final bounds as previous works did for ReLU networks or S-shaped activations [26, 48].

We notice that Shi et al. [34] mentioned that there are two solutions for $\underline{a}, \underline{b}, \underline{c}$ and $\overline{a}, \overline{b}, \overline{c}$ respectively that solves Eq. (8):

$$\begin{cases} \underline{a}_1 = \underline{y} \\ \underline{b}_1 = \underline{x} \\ \underline{c}_1 = -\underline{x}\underline{y} \end{cases}, \quad \begin{cases} \overline{a}_1 = \overline{y} \\ \overline{b}_1 = \underline{x} \\ \overline{c}_1 = -\underline{x}\overline{y} \end{cases}, \tag{9}$$

$$\begin{cases} \underline{a}_2 = \overline{y} \\ \underline{b}_2 = \overline{x} \\ \underline{c}_2 = -\overline{x}\overline{y} \end{cases}, \quad \begin{cases} \overline{a}_2 = \underline{y} \\ \overline{b}_2 = \overline{x} \\ \overline{c}_2 = -\overline{x}\underline{y} \end{cases}. \tag{10}$$

Therefore, to make the parameters optimizable, we introduce parameters $\underline{\alpha}$ and $\overline{\alpha}$, and we interpolate between Eq. (9) and Eq. (10) as:

$$\begin{cases} \underline{a} = \underline{\alpha}\underline{y} + (1 - \underline{\alpha})\overline{y} \\ \underline{b} = \underline{\alpha}\underline{x} + (1 - \underline{\alpha})\overline{x} \\ \underline{c} = -\underline{\alpha}\underline{x}\underline{y} - (1 - \underline{\alpha})\overline{x}\overline{y} \end{cases} \quad \text{s.t. } 0 \leq \underline{\alpha} \leq 1, \tag{11}$$

$$\begin{cases} \overline{a} = \overline{\alpha}\overline{y} + (1 - \overline{\alpha})\underline{y} \\ \overline{b} = \overline{\alpha}\underline{x} + (1 - \overline{\alpha})\overline{x} \\ \overline{c} = -\overline{\alpha}\underline{x}\overline{y} - (1 - \overline{\alpha})\overline{x}\underline{y} \end{cases} \quad \text{s.t. } 0 \leq \overline{\alpha} \leq 1. \tag{12}$$

It is easy to verify that interpolating between two sound linear relaxations satisfying Eq. (7) still yields a sound linear relaxation. And $\underline{\alpha}$ and $\overline{\alpha}$ are part of all the optimizable linear relaxation parameters $\boldsymbol{\alpha}$ mentioned in Section 2.

B.2 Optimizable Linear Relaxation for Sine

We also derive new optimized linear relaxation for periodic functions, in particular $\sin(x)$. For $sin(x)$ where $x \in [\underline{x}, \overline{x}]$, we aim to relax and bound $sin(x)$ as:

$$\forall x \in [\underline{x}, \overline{x}], \quad \underline{a}x + \underline{b} \leq \sin(x) \leq \overline{a}x + \overline{b}, \tag{13}$$

where $\underline{a}, \underline{b}, \overline{a}, \overline{b}$ are parameters in the linear bounds. A non-optimizable linear relaxation for sin already exists in α,β-CROWN and we adopt it as an initialization and focus on making it optimizable. At initialization, we first check the line connecting $(\underline{x}, \sin(\underline{x}))$ and $(\overline{x}, \sin(\overline{x}))$, and this line is adopted as the lower bound or the upper bound without further optimization, if it is a sound bounding line.

Otherwise, a tangent line is used as the bounding line with the tangent point being optimized. Within $[\underline{x}, \overline{x}]$, if $\sin(x)$ happens to be monotonic with at most

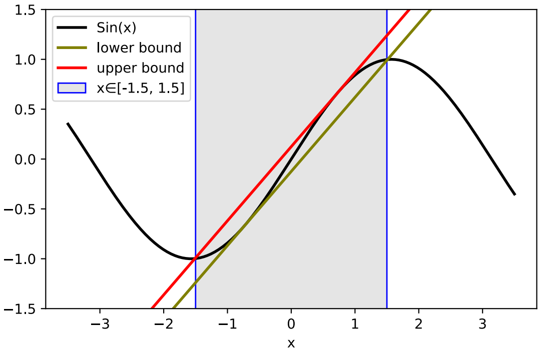


Fig. 5: Linear relaxation for a Sin activation in an input range $[-1.5, 1.5]$ where the function is S-shaped.

only one inflection point, the tangent point can be optimized in a way similar to bounding an S-shaped activation [26], as illustrated in Figure 5.

Otherwise, there are multiple extreme points within the input range. Initially, we aim to find a tangent line that passes $(\underline{x}, \sin(\underline{x}))$ as the bounding line. Since $\underline{x}$ may be at different cycles of the sin function, we project into the cycle with range $[-0.5\pi, 1.5\pi]$, by taking $\tilde{\underline{x}}_l = \underline{x} - 2\underline{k}_l\pi$, where $\underline{k}_l = \lfloor \frac{\underline{x}+0.5\pi}{2\pi} \rfloor$. With a binary search, we find a tangent point $\underline{\alpha}_l$ on the projected cycle that satisfies

$$\sin'(\underline{\alpha}_l)(\underline{\alpha}_l - \tilde{\underline{x}}_l) + \sin(\tilde{\underline{x}}_l) = \sin(\underline{\alpha}_l), \tag{14}$$

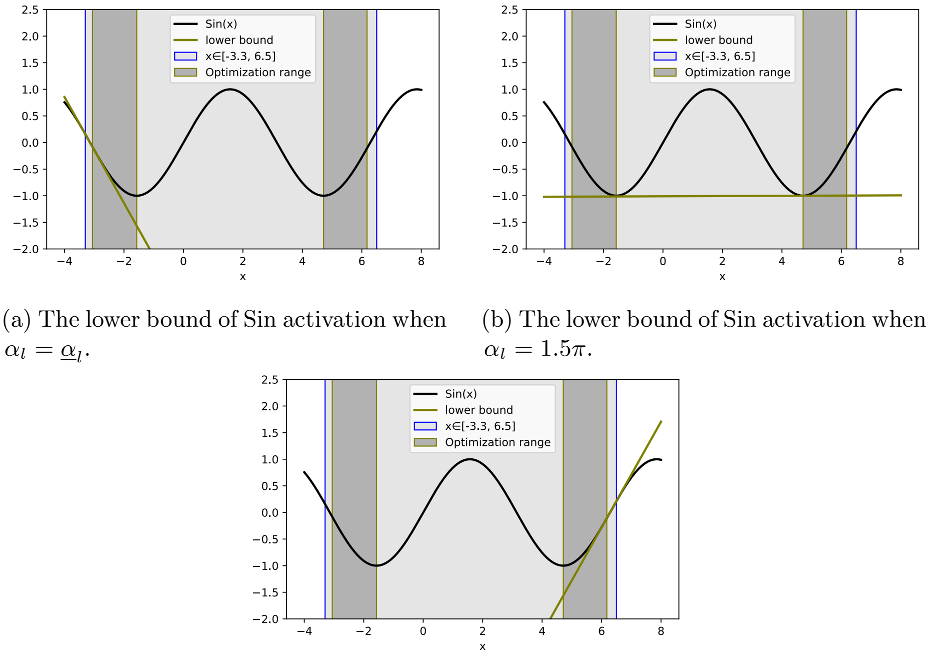
which corresponds to a tangent point $\underline{t}_l = \underline{\alpha}_l + 2k_l\pi$ at the original cycle of $\underline{x}$, and for any tangent point within the range of $[\underline{\alpha}_l + 2\underline{k}_l\pi, 1.5\pi + 2\underline{k}_l\pi]$, the tangent line is a valid lower bound. Similarly, we also consider the tangent line passing $(\overline{x}, \sin(\overline{x}))$, and we take $\tilde{\overline{x}}_l = \overline{x} - 2\overline{k}_l\pi$, where $\overline{k}_l = \lfloor \frac{\overline{x}-1.5\pi}{2\pi} \rfloor$, so that $\tilde{\overline{x}}_l$ is within range $[1.5\pi, 3.5\pi]$. We also conduct a binary search to find the tangent point $\overline{\alpha}_l$, which corresponds to to $\overline{\alpha}_l + 2\overline{k}_l\pi$ in the original cycle of $\overline{x}$, and for any tangent point within the range $[1.5\pi + 2\overline{k}_l\pi, \overline{\alpha}_l + 2\overline{k}_l\pi]$, the tangent line is also a valid lower bound. We make the tangent point optimizable with a parameter α_l $(\underline{\alpha_l} \leq \alpha_l \leq \overline{\alpha}_l)$, which corresponds to a tangent line at tangent point t_l as the lower bound in Eq. (13) and Figure 6:

$$\begin{cases} \underline{a} = \sin'(t_l) \\ \underline{b} = \sin(t_l) - \underline{a}t_l \end{cases}, \tag{15}$$

$$\text{where} \begin{cases} t_l = \alpha_l + 2\underline{k}_l\pi \ \text{ if } \ \underline{\alpha}_l \leq \alpha_l \leq 1.5\pi \\ t_l = \alpha_l + 2\overline{k}_l\pi \ \text{ if } \ 1.5\pi < \alpha_l \leq \overline{\alpha}_l \end{cases}. \tag{16}$$

In particular, when $\alpha_l = 1.5\pi$, both $\alpha_l + 2\underline{k}_l\pi$ and $\alpha_l + 2\overline{k}_l\pi$ are tangent points for the same tangent line.

The derivation for the upper bound is similar. We take $\tilde{\underline{x}}_u = \underline{x} - 2\underline{k}_u\pi$, where $\underline{k}_u = \lfloor \frac{\underline{x}-0.5\pi}{2\pi} \rfloor$, so that $\tilde{\underline{x}}_u$ is in range $[0.5\pi, 2.5\pi]$. And we take $\tilde{\overline{x}}_u = \overline{x} - 2\overline{k}_u\pi$,



(a) The lower bound of Sin activation when $\alpha_l = \underline{\alpha}_l$.

(b) The lower bound of Sin activation when $\alpha_l = 1.5\pi$.

(c) The lower bound of Sin activation when $\alpha_l = \overline{\alpha}_l$.

Fig. 6: Optimizing the lower bound of a Sin activation, where "Optimization range" shows all the valid tangent points for the lower bound during the optimization.

where $\overline{k}_u = \lfloor \frac{\overline{x}-2.5\pi}{2\pi} \rfloor$, so that $\tilde{\overline{x}}_u$ is in range $[2.5\pi, 4.5\pi]$. Let $\underline{\alpha_u}$ be the tangent point where the tangent line crosses $\underline{\tilde{x}}_u$, and $\overline{\alpha}_u$ be the tangent point where the tangent line crosses $\tilde{\overline{x}}_u$, as found by a binary search. We define an optimizable parameter α_u ($\underline{\alpha_u} \leq \alpha_u \leq \overline{\alpha}_u$) which corresponds to a tangent line as the upper bound:

$$\begin{cases} \overline{a} = \sin'(t_u) \\ \overline{b} = \sin(t_u) - \overline{a}t_u \end{cases}, \tag{17}$$

$$\text{where} \quad \begin{cases} t_u = \alpha_u + 2\underline{k}_u\pi \ \text{ if } \ \underline{\alpha}_u \leq \alpha_u \leq 2.5\pi \\ t_u = \alpha_u + 2\overline{k}_u\pi \ \text{ if } \ 2.5\pi < \alpha_u \leq \overline{\alpha}_u \end{cases}. \tag{18}$$

B.3 Optimizable Linear Relaxation for GeLU

For GeLU function where $x \in [\underline{x}, \overline{x}]$ are the intermediate bounds for x, we aim to relax and bound $\text{GeLU}(x)$ as:

$$\forall x \in [\underline{x}, \overline{x}], \quad \underline{a}x + \underline{b} \leq \text{GeLU}(x) \leq \overline{a}x + \overline{b}, \tag{19}$$

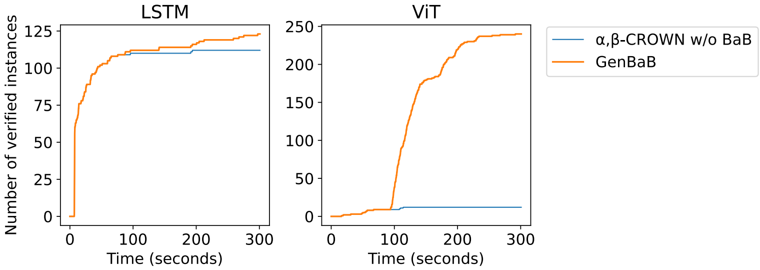


Fig. 7: Total number of verified instances against running time threshold on LSTM and ViT.

where $\underline{a}, \underline{b}, \overline{a}, \overline{b}$ are parameters in the linear bounds.

Given input range $[\underline{x}, \overline{x}]$, if $\overline{x} \leq 0$ or $\underline{x} \geq 0$, the range contains only one inflection point, the tangent point can be optimized in a way similar to bounding an S-shaped activation [26]. In other cases, $\underline{x} < 0$ and $\overline{x} > 0$ holds. For the upper bound, we use the line passing $(\underline{x}, \text{GeLU}(\underline{x}))$ and $(\overline{x}, \text{GeLU}(\overline{x}))$. For the lower bound, we derive two sets of tangent lines that crosses $(\underline{x}, \text{GeLU}(\underline{x}))$ and $(\overline{x}, \text{GeLU}(\overline{x}))$ with tangent points denoted as $\underline{\alpha}$ and $\overline{\alpha}$ respectively. We determine $\underline{\alpha}, \overline{\alpha}$ using a binary search that solves:

$$\begin{cases} \text{GeLU}'(\underline{\alpha})(\underline{\alpha} - \underline{x}) + \text{GeLU}(\underline{x}) = \text{GeLU}(\underline{\alpha}) \\ \text{GeLU}'(\overline{\alpha})(\overline{\alpha} - \overline{x}) + \text{GeLU}(\overline{x}) = \text{GeLU}(\overline{\alpha}) \end{cases}. \tag{20}$$

Any tangent line with a tangent point α ($\underline{\alpha} \leq \alpha \leq \overline{\alpha}$) is a valid lower bound, which corresponds to the lower bound in Eq. (19) with:

$$\begin{cases} \underline{a} = \text{GeLU}'(\alpha) \\ \underline{b} = \text{GeLU}(\alpha) - \alpha\,\text{GeLU}'(\alpha) \end{cases} \quad \text{s.t. } \underline{\alpha} \leq \alpha \leq \overline{\alpha}. \tag{21}$$

C Additional Results

C.1 Time Cost on LSTM and ViT

In Figure 7, we show the time cost on LSTM and ViT models, as discussed in Section 4.3.

C.2 Experiments on Self-Attention Networks from [44]

To compare with Wei et al. [44] which only supports verifying single-layer self-attention networks but not the entire ViT, we adopt pre-trained models from Wei et al. [44] and run our verification methods under their settings, with 500 test images in MNIST using $\epsilon = 0.02$. We show the results in Table 7, where our methods also outperform Wei et al. [44] on all the models.

Table 7: Number of verified instances out of 500 instances in MNIST with $\epsilon = 0.02$. A-small, A-medium and A-big are three self-attention networks with different parameter sizes from Wei et al. [44].

Method	A-small	A-medium	A-big
Wei et al. [44]	406	358	206
α,β-CROWN w/o BaB	444	388	176
GenBaB (ours)	**450**	**455**	**232**
Upper bound	463	479	482

C.3 Experiments on a ResNet Model

In this section, we demonstrate our method on a ResNet model [15]. The model has the same size as the one used in Wang et al. [43], which has 2 residual blocks with 6 convolutional layers and fully-connected layers in total. Since our focus is on general nonlinearities, we use GeLU activation instead of ReLU. We train the model on CIFAR-10 with PGD adversarial training using $\epsilon = 2/255$. As the results shown in Table 8, our GenBaB significantly improves the verification on the ResNet model compared to α,β-CROWN without BaB.

Table 8: Number of verified instances on a ResNet model with the same architecture as the ResNet in Wang et al. [43], but the activation function is GeLU instead of ReLU.

Method	ResNet
α,β-CROWN w/o BaB	24
GenBaB (ours)	**74**

C.4 Experiments on Larger ϵ

In this section, we demonstrate GenBaB on larger ϵ. We consider $\epsilon = 2/255$ and $\epsilon = 8/255$ for 4 × 100 feedforward networks with various activation functions on CIFAR-10. As the results shown in Table 9, our GenBaB effectively improves the verification on all these models.

C.5 Experiments on a ReLU Network

In this section, we study the effect of our BBPS heuristic on ReLU models. We adopt settings in Müller et al. [29], Singh et al. [35, 36] and experiment on a "ConvSmall" model with ReLU activation. The verification is evaluated on 1000 instances on CIFAR-10, following prior works. We show the results in Table 10,

Table 9: Number of verified instances on 4 × 100 feedforward networks with various activation functions on CIFAR-10 when a larger $\epsilon = 2/255$ or $\epsilon = 8/255$ is used.

Method	$\epsilon = 2/255$				$\epsilon = 8/255$			
	Sigmoid	Tanh	Sin	GeLU	Sigmoid	Tanh	Sin	GeLU
α,β-CROWN w/o BaB	33	15	11	39	16	11	2	34
GenBaB (ours)	**56**	**26**	**65**	**65**	**37**	**19**	**22**	**35**

Table 10: Results on a "ConvSmall" model with ReLU activation [29, 35, 36] on 1000 instances from CIFAR-10. Percentage of instances verified by various methods are reported. For methods other than PRIMA, we use α,β-CROWN as the underlying verifier but vary the branching heuristic. See explanation about the backup score in Appendix C.5.

Method	Verified
PRIMA	44.6%
BaBSR w/o backup score	45.6%
BaBSR w/ backup score	46.2%
Backup score only	45.0%
BBPS w/o backup score	46.0%
BBPS w/ backup score	46.2%

We find that on this ReLU network, our BBPS also works better than the BaBSR heuristic, when there is no *backup score* (46.0% verified by BBPS v.s. 45.6% verified by BaBSR). However, we find that recent works typically add a *backup score* for BaBSR, which is another heuristic score that serves as a backup for neurons with extremely small BaBSR scores. The backup score did not exist in the original BaBSR heuristic [4] but it appeared in De Palma et al. [7] and has also been adopted by works such as Wang et al. [43] when using BaBSR for ReLU networks. This backup score basically uses the intercept of the linear relaxation for the upper bound of a ReLU neuron that needs branching. Unlike BaBSR or BBPS, the backup score does not aim to directly estimate the change on the bounds computed by bound propagation, but aims to use the intercept to reflect the reduction of the linear relaxation after the branching. When the backup score is combined with BaBSR or BBPS for ReLU networks, the backup score seems to dominate the performance, where both BaBSR and BBPS have similar performance with the backup score added (46.2% verified), which hides the underlying improvement of BBPS over BaBSR by providing a more precise estimation. However, the backup score is specifically for ReLU, assuming that the intercept of the linear relaxation can reflect the reduction of the linear relaxation, which is not the case for general nonlinearities. We leave it for future work to study the possibility of designing a backup score for general nonlinearities.

D Experimental Details

Implementation details. We implement our GenBaB algorithm based on α,β-CROWN[8] which originally did not support BaB for nonlinearities other than ReLU. To pre-optimize the branching points, we enumerate the branching points (p in Eq. (3)) with a step size instead of performing gradient descent, considering that we only have up to two parameters for the branching points in our experiments. For nonlinearities with a single input, we use a step size of 0.01, and for nonlinearities with two inputs, we use a step size of 0.1. We pre-optimize the branching points for intermediate bounds within the range of $[-5, 5]$. For all the experiments, each experiment is run using a single NVIDIA GTX 1080Ti GPU.

Details on training the models. To train our models on CIFAR-10, we use PGD adversarial training [27]. We use 7 PGD steps during the training and the step size is set to $\epsilon/4$. For training the Sigmoid networks in Table 3, we use the SGD optimizer with a learning rate of 5×10^{-2} for 100 epochs; and for training the Tanh networks, we use the SGD optimizer with a learning rate of 1×10^{-2} for 100 epochs. For training the LSTMs in Table 4, we use the Adam optimizer with a learning of 10^{-3} for 30 epochs. And for training the ViTs, we use the Adam optimizer with a learning of 5×10^{-3} for 100 epochs. For Sine networks, we use the SGD optimizer with a learning rate of 1×10^{-3} for 100 epochs

Memory cost. Memory cost of our framework is highly manageable. To store bounds for branched domains, note that the pool of domains with branched intermediate bounds is stored on CPU memory (not GPU), and in each iteration of BaB, only a batch of domains is loaded to GPU and handled in parallel on GPU, the batch size can be configurable to fit the GPU memory (mentioned in Section 3.1). For bound computation during BaB, as mentioned in Section 3.4, since intermediate bounds are not re-computed during BaB, the space complexity for each subproblem is the same as a regular NN forward pass.

For the lookup table for pre-optimized branching points, the memory cost is also small: 4MB for 1D nonlinearities (tanh, sigmoid, sin, etc.) and 800MB for 2D nonlinearities (multiplication). Although the memory cost of a lookup table can become larger for higher-dimensional nonlinearities, the granularity of lookup tables is configurable to reduce the memory requirement at the cost of slightly less optimal branching points. For further scalability, future works may compress the lookup table by an NN (such as Julian et al. [19]) for high dimensional nonlinearities, and the validity of branching points predicted by NN is easy to guarantee by clipping the prediction.

[8] `https://github.com/Verified-Intelligence/alpha-beta-CROWN`