# An Efficient Framework for Computing Tight Lipschitz Constants of Neural Networks

**Anonymous Author(s)**
Affiliation
Address
email

## Abstract

Lipschitz constants are connected to many properties of neural networks, such as robustness, fairness, and generalization. Existing methods for computing Lipschitz constants either are computationally inefficient or produce loose upper bounds. In this paper, we develop an efficient framework for computing the $\ell_\infty$ local Lipschitz constant of a neural network by tightly upper bounding the norm of Clarke Jacobian. We view the computation for Clarke Jacobian by chain rule as a higher-order backward computational graph. On this graph, we adopt bound-propagation with linear relaxation to obtain provable bounds, and we derive tight linear relaxations for specific nonlinearities in Clarke Jacobian that previous works could not tightly bound. We further tighten our bounds by Branch-and-Bound (BaB) when time budget allows. Experiments show that on tiny models, our method produces comparable bounds compared to exact methods that cannot scale to slightly larger models; on larger models, our method efficiently produces tighter results than existing relaxed or naive methods, and our method scales to much larger practical models that previous works could not handle. We also demonstrate a potential application of our method for provable monotonicity analysis.

## 1 Introduction

Lipschitz constants are important for characterizing many properties of neural networks, including robustness [36, 16, 38, 48, 18], fairness [10, 21], generalization [2] and explanation [12]. In order to more precisely characterize the local behavior of a network, we consider local Lipschitz constants which are only required to hold for a small local region. Intuitively they characterize how fast the output of the network changes between any two input within the region.

It is challenging to exactly and efficiently compute local Lipschitz constants. Naive approaches such as computing the product of the induced norm for all layers cannot capture local information and typically produce vacuous bound that are not useful in practice. On the other hand, Jordan & Dimakis [20] used mixed-integer programming (MIP) to compute exact local Lipschitz constants for small ReLU networks (e.g., up to a few hundred neurons), but solving a MIP is often too inefficient and this approach cannot scale to slightly larger networks. Another line of works focus on computing a relaxed upper bound for better scalability compared to MIP. Weng et al. [43], Zhang et al. [48] used bound propagation methods to bound the Jacobian and compute local Lipschitz bounds. In neural network verification, linear bound propagation [44, 47, 34] methods compute guaranteed output bounds for neural networks by propagating the linear relationship between layers, where nonlinearities in networks are relaxed into linear bounds, e.g., activation functions can be bounded by two lines. However, for nontrivial cases when bounding the gradient of ReLU activation in Jacobian computation, Weng et al. [43], Zhang et al. [48] essentially used loose interval bounds [17, 29, 15] that only take horizontal bounding lines with zero slopes, instead of tighter linear relaxation bounds.

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

Moreover, most of these works only used small toy models and cannot feasibly handle larger practical networks.

In this paper, we aim to efficiently compute relatively tight $\ell_\infty$ local Lipschitz constants. We formulate the problem as upper bounding the norm of the Clarke Jacobian by the chain rule; and we view the computation for this norm as a *higher-order backward computational graph* augmented to the original forward graph of the network, which allows us to develop a scalable and flexible framework for practical neural networks.

Our contributions can be summarized as follows: First, we generalize existing linear bound propagation methods, originally used for neural network verification, to a higher-order backward computational graph for bounding Clarke Jacobian. Second, on the backward graph, we propose tight linear relaxations for Clarke gradients with an optimality guarantee, which is significantly different from relaxing activation functions in neural network verification, while previous works used loose interval bounds in nontrivial cases for local Lipschitz contants. When time budget allows, we further tighten our bounds via Branch-and-Bound, for a trade-off between tightness and time cost. Lastly, our experiments show that our method efficiently produces tightest $\ell_\infty$ local Lipschitz constants compared to other relaxed methods, and is much more efficient than the exact MIP method. Moreover, while previous works commonly used small toy models, our method can scale to much larger models including practical convolutional neural networks (CNN) that previous works could not handle. We also demonstrate an application of our method for provably analyzing the monotonicity of neural networks.

## 2 Related Work

### 2.1 Computing Lipschitz Constants

For neural networks, a loose global Lipschitz constant (upper bound) can be computed by the product of layer-wise induced norms [36]. Based on this product, Virmaux & Scaman [39] considered the effect of activation and approximated upper bounds which, however, are not guaranteed; Gouk et al. [14] used a power method for convolutional layers. LipSDP [11] used semidefinite programming (SDP) to compute tighter bounds. Global Lipschitz constants can be much looser than local Lipschitz constants as they have to hold even for distinct input points, and they cannot characterize the local behavior of neural networks. Thus we focus on local Lipschitz constants in this paper.

On local Lipschitz constants, LipMIP [20] used mixed integer programming (MIP) to compute exact results, which is costly and cannot scale to slightly larger models. LipOpt [13] used polynomial optimization but is still inefficient and limited to smooth activations not including the widely used ReLU. Avant & Morgansen [1] derived layer-wise analytical bounds but only on a simplified definition for Lipschitz constants, which cannot characterize Lipschitzness in the entire local region.

Several other works used methods essentially based on interval bound propagation [17, 29, 15] which computes an interval of lower and upper bounds for each element in the network or the chain rule for Jacobian. Among them, RecurJac [48], improved from FastLip [43], has a recursive procedure to tighten bounds in some special cases but still uses interval bounds in nontrivial cases; and a concurrent work [24] handled non-smooth input perturbations. However, interval bounds are known to be very loose [49]. Bhowmick et al. [3] combined interval bounds with branch-and-bound to compute exact local Lipschitz constants, which is inefficient due to loose interval bounds. In contrast, we propose tight linear relaxation for computing local Lipschitz constants with bound propagation.

### 2.2 Certified Bounds for Neural Networks

Computing sound local Lipschitz constants is also closely related to neural network verification which computes certified output bounds. Some works computed exact bounds by satisfiability modulo theory [22] or mixed integer linear programming [37]. For more efficient verification, bound propagation with convex relaxations has been proposed [44, 41, 47, 43, 34, 31]. In addition, Branch-and-Bound can be used to compute tighter bounds [22, 5, 6, 40, 41, 46, 42] by branching activations or input and bounding each smaller subdomain respectively. Although bound propagation has been generalized to general computational graphs [45], they have not considered high-order computational graphs and did not provide necessary linear relaxation for nonlinearities in Clarke Jacobian.

# 3 Background

## 3.1 ReLU Network

Suppose $f(\mathbf{x})$ is a neural network given input $\mathbf{x} \in \mathbb{R}^d$. For the simplicity of presentation, we mainly assume the neural network is feedforward with ReLU activation throughout the paper, but our method can be applied to general network architectures and activations as will be discussed in Section 4. Suppose the network has $n$ layers, it takes input $h_0(\mathbf{x}) = \mathbf{x}$ and then computes

$$\forall i \in [n],\ z_i(\mathbf{x}) = \mathbf{W}_i h_{i-1}(\mathbf{x}) + \mathbf{b}_i, \quad \forall i \in [n-1],\ h_i(\mathbf{x}) = \sigma(z_i(\mathbf{x})),\ h_n(\mathbf{x}) = z_n(\mathbf{x}),$$

where $i \in [n]$ means $1 \leq i \leq n$, $z_i(\mathbf{x})$ is the pre-activation output of the $i$-th linear layer with weight $\mathbf{W}_i$ and bias $\mathbf{b}_i$, and $h_i(\mathbf{x})$ is the output after activation $\sigma(\cdot)$. This formulation also supports CNNs since convolutional layers are also linear.

## 3.2 Lipschitz Constant

The $(\alpha, \beta)$-Lipschitz constant of a network $f(\mathbf{x})$ over an open set $\mathcal{X} \in \mathbb{R}^d$ is defined as:

$$L^{(\alpha,\beta)}(f, \mathcal{X}) = \sup_{\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{X},\ \mathbf{x}_1 \neq \mathbf{x}_2} \frac{\|f(\mathbf{x}_1) - f(\mathbf{x}_2)\|_\beta}{\|\mathbf{x}_1 - \mathbf{x}_2\|_\alpha}. \tag{1}$$

If $f$ is smooth and $(\alpha, \beta)$-Lipschitz continuous over $\mathcal{X}$, the Lipschitz constant can be computed by upper bounding the norm of Jacobian, i.e., $L^{(\alpha,\beta)}(f, \mathcal{X}) = \sup_{\mathbf{x} \in \mathcal{X}} \|\nabla f(\mathbf{x})\|_{\alpha,\beta}$. Since neural networks with non-smooth ReLU activation are non-smooth functions, we consider Clarke Jacobian (generalized Jacobian) [7] instead, which is defined as the convex hull of $\lim_{i\to\infty} \nabla f(\mathbf{x}_i)$ for any sequence $\{\mathbf{x}_i\}_{i=1}^\infty$ such that every $f(\mathbf{x}_i)$ is differentiable at $\mathbf{x}_i$. We denote the Clarke Jacobian at $\mathbf{x}$ as $\partial f(\mathbf{x})$, and then $L^{(\alpha,\beta)}(f, \mathcal{X}) = \sup_{\mathbf{x} \in \mathcal{X},\ \mathbf{J}(\mathbf{x}) \in \partial f(\mathbf{x})} \|\mathbf{J}(\mathbf{x})\|_{\alpha,\beta}$.

Global Lipschitz constant, which considers the supremum over $\mathcal{X} = \mathbb{R}^d$, needs to guarantee Eq. (1) even for distant $\mathbf{x}_1, \mathbf{x}_2$, and thus it can be loose and cannot capture the local behavior of the network. We focus on local Lipschitz constants, where $\mathcal{X}$ is a small local region. Specifically, we consider $\mathcal{X} = B_p(\mathbf{x}_0, \epsilon) := \{\mathbf{x} : \|\mathbf{x} - \mathbf{x}_0\|_p \leq \epsilon\}$ which is a small $\ell_p$-ball around $\mathbf{x}_0$, and then the local Lipschitz constant only needs to hold for input near a clean data point $\mathbf{x}_0$. We focus on $\ell_\infty$ local Lipschitz constants in this paper, where $\alpha = \beta = \infty$ for the aforementioned definition.

## 3.3 Backward Bound Propagation

To bound the Clarke Jacobian, we will use bound propagation which is originally used to for certifiably bounding the output of neural networks. We adopt a backward bound propagation [47, 44, 34, 45] which typically propagates the linear relationship between the output layer to be bounded and all the previous layers it depends on. Suppose we want to compute certified bounds for output layer $h_n(\mathbf{x})$ w.r.t. all $\mathbf{x}$ from a small domain $B_p(\mathbf{x}_0, \epsilon)$. Starting from the output layer, we take $\mathbf{A}_n = \mathbf{I}$, and then $h_n(\mathbf{x}) = \mathbf{A}_n h_n(\mathbf{x})$. For every $i \in [n]$, suppose $h_i(\mathbf{x})$ can be bounded by linear functions w.r.t. $h_{i-1}(\mathbf{x})$ parameterized by $\underline{\mathbf{P}}_i, \overline{\mathbf{P}}_i, \underline{\mathbf{q}}_i, \overline{\mathbf{q}}_i$ (the bounds elementwise hold):

$$\underline{\mathbf{P}}_i h_{i-1}(\mathbf{x}) + \underline{\mathbf{q}}_i \leq h_i(\mathbf{x}) \leq \overline{\mathbf{P}}_i h_{i-1}(\mathbf{x}) + \overline{\mathbf{q}}_i. \tag{2}$$

Then for every $i = n, n-1, \cdots, 1$ in order, $\mathbf{A}_i h_i(\mathbf{x})$ can be recursively bounded by substituting $h_i(\mathbf{x})$ with Eq. (2), and then $\mathbf{A}_i h_i(\mathbf{x}) \leq \mathbf{A}_{i-1} h_{i-1}(\mathbf{x}) + \mathbf{c}_i$, with:

$$\mathbf{A}_{i-1} = [\mathbf{A}_i]_+ \overline{\mathbf{P}}_i + [\mathbf{A}_i]_- \underline{\mathbf{P}}_i, \quad \mathbf{c}_i = [\mathbf{A}_i]_+ \overline{\mathbf{q}}_i + [\mathbf{A}_i]_- \underline{\mathbf{q}}_i, \tag{3}$$

where $[\cdot]_+$ stands for taking positive elements from the matrix or vector, and vice versa for $[\cdot]_-$. $\mathbf{A}_{i-1}$ stands for the coefficients of the linear relationship propagated from layer $i$ to layer $i-1$, and $\mathbf{c}_i$ is a bias term produced at this layer. By this recursive procedure in a backward manner and accumulating $\mathbf{c}_i (i \in [n])$, eventually bounds are propagated to the input layer as $h_n(\mathbf{x}) \leq \mathbf{A}_0 \mathbf{x} + \sum_{i=1}^n \mathbf{c}_i$, which directly represents the linear relationship between the output layer and $\mathbf{x}$. We can eliminate $\mathbf{x}$ from $\mathbf{A}_0 \mathbf{x}$ to obtain final bounds by considering the domain of $\mathbf{x}$. In particular, when $\mathbf{x} \in B_p(\mathbf{x}_0, \epsilon)$ $(p \geq 1)$, for every $j$-th dimension of $h_n(\mathbf{x})$, we can use the fact that $[\mathbf{A}_0]_{j,:}\mathbf{x} \leq [\mathbf{A}_0]_{j,:}\mathbf{x}_0 + \epsilon \|[\mathbf{A}_0]_{j,:}\|_q$, where $[\mathbf{A}_0]_{j,:}$ stands for the $j$-th row in $\mathbf{A}_0$, $q = 1/(1-1/p)$, and $q$-norm is the dual-norm of $p$-norm.

All the bounds here provably hold for any $\mathbf{x} \in B_p(\mathbf{x}_0, \epsilon)$, and lower bounds can also be obtained by a similar bound propagation.

Parameters in Eq. (2) can be obtained by considering the relation between $h_i(\mathbf{x})$ and $h_{i-1}(\mathbf{x})$. We can first obtain bounds $\mathbf{l}_i \leq z_i(\mathbf{x}) \leq \mathbf{u}_i$ using bound propagation by viewing $z_i(\mathbf{x})$ as the output layer. Then the linear relaxation bounds $h_i(\mathbf{x}) = \sigma(z_i(\mathbf{x}))$ with linear functions of $z_i(\mathbf{x})$:

$$\forall \mathbf{l}_i \leq z_i(\mathbf{x}) \leq \mathbf{u}_i, \;\; \underline{\mathbf{s}}_i z_i(\mathbf{x}) + \underline{\mathbf{t}}_i \leq \sigma(z_i(\mathbf{x})) \leq \bar{\mathbf{s}}_i z_i(\mathbf{x}) + \bar{\mathbf{t}}_i. \tag{4}$$

For every neuron $j$, if $[\mathbf{l}_i]_j \geq 0$ or $[\mathbf{u}_i]_j \leq 0$, $\sigma(z_i(\mathbf{x})) = z_i(\mathbf{x})$ or $\sigma(z_i(\mathbf{x})) = 0$ respectively is already linear, and then we can simply take $[\underline{\mathbf{s}}_i]_j [z_i(\mathbf{x})]_j + [\underline{\mathbf{t}}_i]_j = [\bar{\mathbf{s}}_i]_j [z_i(\mathbf{x})]_j + [\bar{\mathbf{t}}_i]_j = \sigma(z_i(\mathbf{x}))$. Otherwise when $[\mathbf{l}_i]_j < 0 < [\mathbf{u}_i]_j$, as illustrated in Figure 1, the upper bound can be the line passing ReLU activation at $[z_i(\mathbf{x})]_j = [\mathbf{l}_i]_j$ and $[z_i(\mathbf{x})]_j = [\mathbf{u}_i]_j$ respectively, i.e., $[\bar{\mathbf{s}}_i]_j [z_i(\mathbf{x})]_j + [\bar{\mathbf{t}}_i]_j = \left(\frac{\sigma([\mathbf{u}_i]_j) - \sigma([\mathbf{l}_i]_j)}{[\mathbf{u}_i]_j - [\mathbf{l}_i]_j}\right)(z_i(\mathbf{x}) - [\mathbf{l}_i]_j) + \sigma([\mathbf{l}_i]_j)$; the lower bound can be any line with a slope between 0 and 1 and a bias of 0, i.e., $0 \leq [\underline{\mathbf{s}}_i]_j \leq 1$, $[\underline{\mathbf{t}}_i]_j = 0$, where $[\underline{\mathbf{s}}_i]_j$ can be viewed as a parameter optimized with an objective of tightening output bounds [28, 46, 30]. To satisfy Eq. (2), we can take:

$$\underline{\mathbf{P}}_i = \text{diag}(\underline{\mathbf{s}}_i)\mathbf{W}_i, \;\; \overline{\mathbf{P}}_i = \text{diag}(\bar{\mathbf{s}}_i)\mathbf{W}_i, \;\; \underline{\mathbf{q}}_i = \text{diag}(\underline{\mathbf{s}}_i)\mathbf{b}_i + \underline{\mathbf{t}}_i, \;\; \bar{\mathbf{q}}_i = \text{diag}(\bar{\mathbf{s}}_i)\mathbf{b}_i + \bar{\mathbf{t}}_i. \tag{5}$$

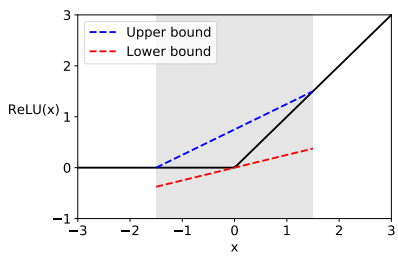


Figure 1: Two lines can bound the ReLU activation when the sign of its input is unstable ($-1.5 \leq x \leq 1.5$ in this example) and are used for bound propagation with linear relaxation.

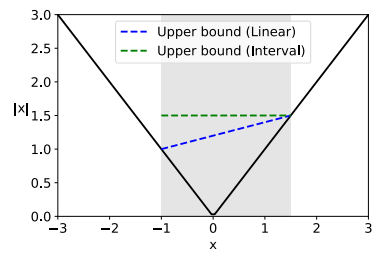


Figure 2: Given the bound on the input to an absolute value function ($-1 \leq x \leq 1.5$ in this example), a line can upper bound the function. We take the blue one with slope $\frac{1}{3}$, while Zhang et al. [48] would take the green one with slope 0, which is a looser upper bound.

# 4 Methodology

We present our method in this section. We first present a novel view to formulate the local Lipschitz constant computation as a higher-order backward computational graph, and use bound propagation to obtain the bounds of the Clarke Jacobian. We discuss our methods for tightly bounding non-linearities in Clarke Jacobian. Finally, we use Branch-and-Bound to further tighten the bounds.

## 4.1 Clarke Jacobian for ReLU Networks

In ReLU networks, the ReLU activation for a single neuron is defined as $\sigma(x) = \max\{x, 0\}$. It is non-smooth at $x = 0$. When computing the Lipschitz constant, we consider Clarke gradient [7] denoted as $\partial\sigma(x)$: when $x < 0$, $\partial\sigma(x) = \{0\}$; when $x > 0$, $\partial\sigma(x) = \{1\}$; and when $x = 0$, $\partial\sigma(x) = [0, 1]$. For the $i$-th layer ($1 \leq i < n$), we use a diagonal matrix $\Delta_i(\mathbf{x})$ to denote a Clarke gradient of ReLU activation for all the neurons in the layer, where $\Delta_i(\mathbf{x}) \in \partial\sigma(z_i(\mathbf{x}))$, $[\Delta_i(\mathbf{x})]_{jj} \in \partial\sigma([z_i(\mathbf{x})]_j)$ , and $\partial\sigma([z_i(\mathbf{x})]_j)$ is the Clarke gradient of the $j$-th neuron. Then a chain rule can be used to compute the Clarke Jacobian for the whole network [19, 20]. Denote $\mathbf{J}_i(\mathbf{x}) \in \frac{\partial f(\mathbf{x})}{\partial h_{i-1}(\mathbf{x})}$ as a Clarke Jacobian w.r.t. the $(i-1)$-th layer in the chain rule. We have $\mathbf{J}_n(\mathbf{x}) = \mathbf{W}_n$, and for all $i \in [n-1]$, we have

$$\mathbf{J}_i(\mathbf{x}) \in \frac{\partial f(\mathbf{x})}{\partial h_{i-1}(\mathbf{x})} = \{\mathbf{J}_{i+1}(\mathbf{x})\Delta_i(\mathbf{x})\mathbf{W}_i : \mathbf{J}_{i+1}(\mathbf{x}) \in \frac{\partial f(\mathbf{x})}{\partial h_i(\mathbf{x})}, \Delta_i(\mathbf{x}) \in \partial\sigma(z_i(\mathbf{x}))\}. \tag{6}$$

Finally $\mathbf{J}_1(\mathbf{x}) \in \partial f(\mathbf{x})$ is a Clarke Jacobian of the entire network [20]. As we aim to compute the $\ell_\infty$ local Lipschitz constant, we upper bound $\|\mathbf{J}_1(\mathbf{x})\|_\infty$ ($\mathbf{J}_1(\mathbf{x}) \in \partial f(x)$) w.r.t. all $\mathbf{x} \in \mathcal{X}$.

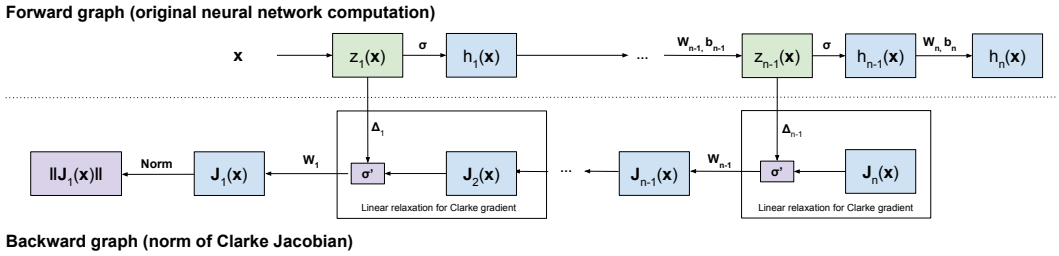


Figure 3: Illustration of our proposed framework where we formulate the computation for local Lipschitz constants by augmenting the original computation graph of the neural network (forward graph) with an additional subgraph for computing the norm of Clarke Jacobian (backward graph). The backward graph relies on the pre-activation values $(\mathbf{z}_1(\mathbf{x}), \cdots, \mathbf{z}_{n-1}(\mathbf{x}))$ in the forward graph to compute the Clarke gradient of activations. Linear relaxation for two categories of nonlinearities in the backward graph (nodes in purple), norm ($\|\mathbf{J}_1(\mathbf{x})\|$) and Clarke gradient ($\sigma'$), are detailed in Figure 2, Figure 4 and Figure 5. We also perform branch-and-bound (BaB) to tighten the bounds by branching pre-activation nodes in the forward graph (nodes in green).

Given the forward pass computation of the original network, we can formulate a computational graph for $\|\mathbf{J}_1(\mathbf{x})\|$ in a backward pass accordingly as shown in Figure 3. We augment the original forward graph with the backward graph, and the backward graph has a dependency on the output of the forward graph due to $\Delta_1, \Delta_2, \cdots, \Delta_{n-1}$. We build the augmented graph by traversing the forward graph, similar to automatic differentiation for neural network training. The computational graph formulation for Clarke Jacobian allows our method to be easily generalized to different network architectures, with a bound propagation framework for general neural networks [45], as long as the linear relaxation for all the involved nonlinearities is well derived.

We use the bound propagation framework described in Section 3.3 to upper bound $\|\mathbf{J}_1(\mathbf{x})\|_\infty$. On the backward graph, when bounds are propagated to $\mathbf{J}_i(\mathbf{x})$, we use $\widetilde{\mathbf{A}}_i$ to denote the coefficient matrix, and we use $\widetilde{\mathbf{c}}_i$ to denote the bias term (in contrast to $\mathbf{A}_i$ and $\mathbf{c}_i$ in Eq. (3)), and then we have: $\forall i \in [n],\ \|\mathbf{J}_1(\mathbf{x})\|_\infty \leq \widetilde{\mathbf{A}}_i \mathbf{J}_i(\mathbf{x}) + \sum_{j=0}^{i-1} \widetilde{\mathbf{c}}_j$. As intermediate bounds for $z_i(\mathbf{x})$ are needed for bounding the forward graph, we also need intermediate bounds $\mathbf{L}_i \leq \mathbf{J}_i(\mathbf{x}) \leq \mathbf{U}_i$ $(\forall \mathbf{x} \in \mathcal{X})$ for the backward graph. And there are particularly two categories of nonlinearities to be tackled on the backward graph: 1) Norm in $\|\mathbf{J}_1(\mathbf{x})\|_\infty$; 2) $\mathbf{J}_{i+1}(\mathbf{x})\Delta_i$ $(\forall i \in [n-1])$ in Eq. (6), which applies the Clarke gradient of ReLU on the later layer's Clarke Jacobian. We handle them in Section 4.2 and Section 4.3 respectively. Then bounds can be propagated starting at $\|\mathbf{J}_1(\mathbf{x})\|_\infty$, to $\mathbf{J}_1(\mathbf{x}), \mathbf{J}_2(\mathbf{x}), \cdots, \mathbf{J}_n(\mathbf{x})$ in order. Bounds are eventually propagated to $\mathbf{J}_n(\mathbf{x})$ as $\|\mathbf{J}_1(\mathbf{x})\|_\infty \leq \widetilde{\mathbf{A}}_n \mathbf{J}_n(\mathbf{x}) + \sum_{j=0}^{i-1} \widetilde{\mathbf{c}}_j$, where $\mathbf{J}_n(\mathbf{x})$ can be substituted by $\mathbf{J}_n(\mathbf{x}) = \mathbf{W}_n$ (Eq. (6)), which yields final bounds on $\|\mathbf{J}_1(\mathbf{x})\|_\infty$.

### 4.2 Norm of Clarke Jacobian

The $\ell_\infty$ norm of Clarke Jacobian is the first nonlinearity we would encounter during backward bound propagation, as it is the last node in the backward graph (see Figure 3). When $f(\mathbf{x})$ is a $K$-way classifier and $f(\mathbf{x}) \in \mathbb{R}^k$, we have $\mathbf{J}_1(\mathbf{x}) \in \mathbb{R}^{K \times d}$, and its norm is computed as $\|\mathbf{J}_1(\mathbf{x})\|_\infty = \max_{1 \leq k \leq K} \sum_{j=1}^{d} \left|[\mathbf{J}_1(\mathbf{x})]_{kj}\right|$, where $K$ rows in $\mathbf{J}_1(\mathbf{x})$ can be bounded separately, and after that we can aggregate results on $K$ rows by taking the $\max$. For the simplicity of the following analysis, we assume $K = 1$ as we can handle one row in $\mathbf{J}_1(\mathbf{x})$ at each time, but they can still be batched in implementation. Then Clarke Jacobian $\mathbf{J}_i(\mathbf{x})$ $(i \in [n])$ and its bounds $\mathbf{L}_i$, $\mathbf{U}_i$ can be viewed as row vectors, and we will use $[\cdot]_j$ to denote the $j$-th element. By a tight linear relaxation for $|\cdot|$ as shown in Figure 2 (only the upper bound is needed), we have the following bound for $\|\mathbf{J}_1(\mathbf{x})\|_\infty$:

**Proposition 1.** *Suppose* $\forall j \in [d]$, $[\mathbf{L}_1]_j \leq [\mathbf{J}_1(\mathbf{x})]_j \leq [\mathbf{U}_1]_j$ $(\forall \mathbf{x} \in \mathcal{X})$, *the following bound holds:*

$$\|\mathbf{J}_1(\mathbf{x})\|_\infty \leq \widetilde{\mathbf{A}}_1 \mathbf{J}_1(\mathbf{x}) + \widetilde{\mathbf{c}}_0, \quad \textit{where } [\widetilde{\mathbf{A}}_1]_j = \frac{\left|[\mathbf{U}_1]_j\right| - \left|[\mathbf{L}_1]_j\right|}{[\mathbf{U}_1]_j - [\mathbf{L}_1]_j}, \quad [\widetilde{\mathbf{c}}_0]_j = -[\widetilde{\mathbf{A}}_1]_j [\mathbf{L}_1]_j + \left|[\mathbf{L}_1]_j\right|.$$

We prove it in Appendix C.1, and this propagates bounds from the norm of Clarke Jacobian to $\mathbf{J}_1(\mathbf{x})$.

### 4.3 Clarke Gradient of the Activation

For each layer $i \in [n-1]$ in the chain rule for computing the Clarke Jacobian as Eq. (6), $\mathbf{J}_{i+1}(\mathbf{x})\Delta_i(\mathbf{x})$ is involved in addition to fixed weight matrix $\mathbf{W}_i$. $\Delta_i(\mathbf{x})$ is determined by pre-activation value $z_i(\mathbf{x})$ from the forward graph. For the $j$-th neuron in the layer, $[\mathbf{J}_{i+1}(\mathbf{x})\Delta_i(\mathbf{x})]_j = [\mathbf{J}_{i+1}(\mathbf{x})]_j[\Delta_i(\mathbf{x})]_{jj}$, when $[\mathbf{u}_i]_j < 0$ or $[\mathbf{l}_i]_j > 0$, we have $[\Delta_i(\mathbf{x})]_{jj} = 0$ and $[\Delta_i(\mathbf{x})]_{jj} = 1$ respectively fixed, and then for these special cases $[\mathbf{J}_{i+1}(\mathbf{x})\Delta_i(\mathbf{x})]_j$ is linear w.r.t. $[\mathbf{J}_{i+1}(\mathbf{x})]_j$. Otherwise, $[\mathbf{J}_{i+1}(\mathbf{x})\Delta_i(\mathbf{x})]_j$ is nonlinear since both $[\mathbf{J}_{i+1}(\mathbf{x})]_j$ and $[\Delta(\mathbf{x})]_{jj}$ may vary given different $\mathbf{x} \in \mathcal{X}$. We then focus on relaxing the nonlinearity $[\mathbf{J}_{i+1}(\mathbf{x})]_j[\Delta_i(\mathbf{x})]_{jj}$ given $[\Delta_i(\mathbf{x})]_{jj} = [0, 1]$.

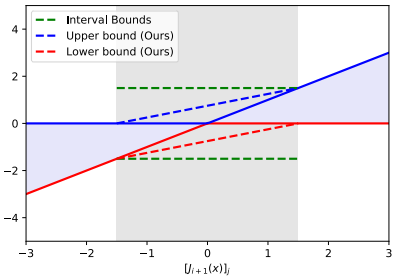


Figure 4: For a neuron $j$ in layer $i$, when $[\Delta_i(\mathbf{x})]_{jj} = [0, 1]$, we plot $[\mathbf{J}_{i+1}(\mathbf{x})\Delta_i(\mathbf{x})]_j$ as a group of functions w.r.t. $[\mathbf{J}_{i+1}(\mathbf{x})]_j$ (violet area). In this example, $-1.5 \leq [\mathbf{J}_{i+1}(\mathbf{x})]_j \leq 1.5$. Our linear relaxation (dashed blue and green lines) is much tighter than interval bounds (dashed green lines) by Zhang et al. [48].

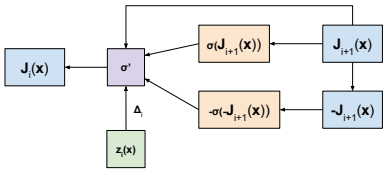


Figure 5: Computation graph for bounding each Clarke gradient. It is the internal computation for boxes in Figure 3. Two ReLU nodes are inserted as per Proposition 2. In bound propagation, the $\sigma'$ node takes intermediate bounds from $z_i(\mathbf{x})$ to decide whether bounds are directly propagated from $\mathbf{J}_{i+1}(\mathbf{x})$ (no relaxation needed), or indirectly via $\sigma(\mathbf{J}_{i+1}(\mathbf{x}))$ and $-\sigma(-\mathbf{J}_{i+1}(\mathbf{x}))$ (with relaxation), for each neuron respectively.

As illustrated in Figure 4, the solid blue line and solid red line are the exact upper and lower bound respectively for $[\mathbf{J}_{i+1}(\mathbf{x})\Delta_i(\mathbf{x})]_j$, which are piecewise linear w.r.t. $[\mathbf{J}_{i+1}(\mathbf{x})]_j$. The most nontrivial case appears when $[\mathbf{L}_{i+1}(\mathbf{x})]_j < 0 < [\mathbf{U}_{i+1}(\mathbf{x})]_j$, where exact bounds for $[\mathbf{J}_{i+1}(\mathbf{x})\Delta_i(\mathbf{x})]_j$ are nonlinear given the input region. We relax the exact bounds into linear bounds that can be handled by bound propagation with linear relaxation. Previous works such as RecurJac [48] have not tightly bounded this linearity in the nontrivial case and commonly degenerate to interval bounds (dashed green lines with zero slopes in the figure, which are loose bounds).

In contrast, we propose a tight linear relaxation (dashed blue and red lines in Figure 4). We observe that the exact upper bound is essentially a ReLU function of $\mathbf{J}_{i+1}(\mathbf{x})$ (solid blue line in Figure 4), and the exact lower bound can be viewed as a ReLU function flipped both horizontally and vertically (solid red line in Figure 4). Formally, we have:

**Proposition 2.** *For the $j$-th neuron in layer $i$ ($i \in [n-1]$), if $[\Delta_i(\mathbf{x})]_{jj} = [0, 1]$, we have:*

$$[\mathbf{J}_{i+1}(\mathbf{x})\Delta_i(\mathbf{x})]_j \geq \min\{[\mathbf{J}_{i+1}(\mathbf{x})]_j, 0\}, \quad [\mathbf{J}_{i+1}(\mathbf{x})\Delta_i(\mathbf{x})]_j \leq \max\{[\mathbf{J}_{i+1}(\mathbf{x})]_j, 0\}, \tag{7}$$

*which can be rewritten using ReLU activation:*

$$\min\{[\mathbf{J}_{i+1}(\mathbf{x})]_j, 0\} = -\sigma(-[\mathbf{J}_{i+1}(\mathbf{x})]_j), \quad \max\{[\mathbf{J}_{i+1}(\mathbf{x})]_j, 0\} = \sigma([\mathbf{J}_{i+1}(\mathbf{x})]_j). \tag{8}$$

Its correctness can be easily verified by considering the sign of $[\mathbf{J}_{i+1}(\mathbf{x})]_j$ as shown in Appendix C.2. We then decompose $[\mathbf{J}_{i+1}(\mathbf{x})\Delta_i(\mathbf{x})]_j$ into two ReLU activations (Eq. (8)), as illustrated in Figure 5.

ReLU can be further relaxed by Eq. (4), as derived in Appendix C.3, which yields:

**Proposition 3.** *For the $j$-th neuron in the $i$-th layer ($i \in [n-1]$), given $[\Delta_i(\mathbf{x})]_{jj} = [0, 1]$ and $[\mathbf{L}_{i+1}]_j \leq [\mathbf{J}_{i+1}(\mathbf{x})]_j \leq [\mathbf{U}_{i+1}]_j$, we have the following relaxation:*

$$[\mathbf{J}_{i+1}(\mathbf{x})\Delta_i(\mathbf{x})]_j \geq [\widetilde{\mathbf{s}}_{i+1}]_j[\mathbf{J}_{i+1}(\mathbf{x})]_j + [\widetilde{\mathbf{t}}_{i+1}]_j, \quad [\mathbf{J}_{i+1}(\mathbf{x})\Delta_i(\mathbf{x})]_j \leq [\widetilde{\overline{\mathbf{s}}}_{i+1}]_j[\mathbf{J}_{i+1}(\mathbf{x})]_j + [\widetilde{\overline{\mathbf{t}}}_{i+1}]_j,$$

$$\textit{where} \quad [\widetilde{\mathbf{s}}_{i+1}]_j = \frac{-\sigma(-[\mathbf{U}_{i+1}]_j) + \sigma(-[\mathbf{L}_{i+1}]_j)}{[\mathbf{U}_{i+1}]_j - [\mathbf{L}_{i+1}]_j}, \quad [\widetilde{\mathbf{t}}_{i+1}]_j = -[\widetilde{\mathbf{s}}_{i+1}]_j[\mathbf{L}_{i+1}]_j - \sigma(-[\mathbf{L}_{i+1}]_j),$$

$$[\widetilde{\overline{\mathbf{s}}}_{i+1}]_j = \frac{\sigma([\mathbf{U}_{i+1}]_j) - \sigma([\mathbf{L}_{i+1}]_j)}{[\mathbf{U}_{i+1}]_j - [\mathbf{L}_{i+1}]_j}, \quad [\widetilde{\overline{\mathbf{t}}}_{i+1}]_j = -[\widetilde{\overline{\mathbf{s}}}_{i+1}]_j[\mathbf{L}_{i+1}]_j + \sigma([\mathbf{L}_{i+1}]_j).$$

In the following theorem, we show that relaxation in Proposition 3 is the tightest linear relaxation:

**Theorem 4.** *For the $j$-th neuron in the $i$-th layer ($i \in [n-1]$), and any $\underline{s}, \underline{t}, \overline{s}, \overline{t}$ satisfying*

$$\forall J \in \Big[[\mathbf{L}_{i+1}]_j, [\mathbf{U}_{i+1}]_j\Big], \forall [\Delta_i(\mathbf{x})]_{jj} \in [0,1], \;\; \underline{s}J + \underline{t} \leq J \cdot [\Delta_i(\mathbf{x})]_{jj} \leq \overline{s}J + \overline{t}, \tag{9}$$

*we have*

$$(\underline{s}, \underline{t}) \neq ([\widetilde{\underline{\mathbf{s}}}_{i+1}]_j, [\widetilde{\underline{\mathbf{t}}}_{i+1}]_j) \implies \forall [\mathbf{L}_{i+1}]_j < J < [\mathbf{U}_{i+1}]_j, \;\; \underline{s}J + \underline{t} < [\widetilde{\underline{\mathbf{s}}}_{i+1}]_j J + [\widetilde{\underline{\mathbf{t}}}_{i+1}]_j,$$
$$(\overline{s}, \overline{t}) \neq ([\widetilde{\overline{\mathbf{s}}}_{i+1}]_j, [\widetilde{\overline{\mathbf{t}}}_{i+1}]_j) \implies \forall [\mathbf{L}_{i+1}]_j < J < [\mathbf{U}_{i+1}]_j, \;\; \overline{s}J + \overline{t} > [\widetilde{\overline{\mathbf{s}}}_{i+1}]_j J + [\widetilde{\overline{\mathbf{t}}}_{i+1}]_j,$$

*where $[\widetilde{\underline{\mathbf{s}}}_{i+1}]_j, [\widetilde{\underline{\mathbf{t}}}_{i+1}]_j, [\widetilde{\overline{\mathbf{s}}}_{i+1}]_j, [\widetilde{\overline{\mathbf{t}}}_{i+1}]_j$ are by Proposition 3.*

This theorem states that our proposed linear relaxation is provably tighter than any other valid linear relaxation. We prove this theorem in Appendix C.4.

### 4.4 Branch and Bound

Formulating the backward computational graph also enables the use of Branch-and-Bound (BaB) to further tighten the bounds if time budget allows. We denote $\mathcal{C}_0 = \{(\mathbf{l}_1, \mathbf{u}_1), (\mathbf{l}_2, \mathbf{u}_2), \cdots (\mathbf{l}_{n-1}, \mathbf{u}_{n-1})\}$ as the domain of all pre-activation bounds of ReLU neurons before BaB, not including ReLU from Eq. (8). We use $\mathcal{C}$ to denote a pool of domains that we currently have, and initially $\mathcal{C} = \{\mathcal{C}_0\}$. BaB aims to recursively split domains in $\mathcal{C}$ into smaller subdomains $\mathcal{C}_1, \mathcal{C}_2, \cdots$, where $\mathcal{C}_0 = \mathcal{C}_1 \bigcup \mathcal{C}_2 \bigcup \cdots$, and bounds for each domain can be computed respectively using bound propagation, since linear relaxation can generally be tighter with smaller domains.

At each iteration, we take $B$ domains in $\mathcal{C}$ with loosest bounds, where $B$ is the batch size for BaB. For each domains, we choose one layer $\hat{i}$ and one ReLU neuron $\hat{j}$ in layer $\hat{i}$ on the forward graph, such that the neuron has uncertain Clarke gradient ($[\mathbf{l}_{\hat{i}}]_{\hat{j}} \leq 0 \leq [\mathbf{u}_{\hat{i}}]_{\hat{j}}$). We branch this domain into two subdomains with intermediate bounds $([\mathbf{l}_{\hat{i}}]_{\hat{j}}, -\tilde{\epsilon})$ and $(\tilde{\epsilon}, [\mathbf{l}_{\hat{i}}]_{\hat{j}})$ respectively, where $\tilde{\epsilon}$ is a sufficiently small value (e.g., $10^{-9}$). Thereby, the Clarke gradient for ReLU becomes certain in each subdomain (0 and 1 respectively) with other neurons unchanged. We compute the bounds for the new subdomains, and we repeat this process until reaching a time limit or there is no domain left to branch.

We use a heuristic score to decide which neuron to branch for a given domain, by estimating the potential improvement to the bounds after the branching [5]. Here for branching a ReLU neuron on the forward graph, we estimate the potential improvement on the bounds of the backward graph. In particular, suppose neuron $\hat{j}$ in layer $\hat{i}$ is branched, the gap between the upper and lower bound in the liner relaxation is eliminated, which can tighten the final bound. We estimate the area of the gap before branching (the gap between the blue dashed line and the red dashed line in Figure 4) as $\frac{1}{2}([\mathbf{U}_{\hat{i}+1}]_{\hat{j}} - [\mathbf{L}_{\hat{i}+1}]_{\hat{j}})^2$, multiplied by the coefficient $[\widetilde{\mathbf{A}}_{\hat{i}}]_{\hat{j}}$ from the bound propagation as the heuristic score. We thereby branch neurons with highest scores.

## 5 Experiments

In our experiments, we focus on evaluating the tightness of $\ell_\infty$ local Lipschitz constant bounds and the efficiency of computation. There are three parts: 1) We compare the tightness and efficiency on relatively small models and synthetic data, to accommodate slow baselines; 2) We also use practical image datasets with larger models. 3) Finally, we demonstrate an application of our method on analyzing the monotonicity of neural networks. Additional experimental details are provided in Appendix B.

**Baselines** We compare our methods to the following baselines: **NaiveUB** upper bounds Lipschitz constants simply by multiplying the induced norm of each layer, which can scale to arbitrarily large models but the bound is often vacuous; **LipMIP** [20] computes exact local Lipschitz constants using MIP, which compute the exact (tightest) bound but can only work for very small models. **LipSDP** [11] computes upper bounds of $\ell_2$ local Lipschitz constants using SDP, and we convert their $\ell_2$ results into upper bounds for the $\ell_\infty$-case by a $\sqrt{d}$ factor. Since LipSDP by itself does not support the $\ell_\infty$ case which may not be the intended use of LipSDP, the results converted from the $\ell_2$ case are for reference

only. **LipBaB** [3] uses loose interval bounds with branch-and-bound. **RecurJac** [48] uses bound propagation with interval bounds in nontrivial cases but has tightened bounds for some special cases. All the baselines do not support CNN in their implementation, and we convert the CNN models to equivalent MLP models for baselines.

Table 1: Local Lipschitz constant values and runtime (seconds) on MLP and CNN models with growing width for a 16-dimensional synthetic input data point. Smaller values are tighter results. Width for MLP stands for number of neurons in each hidden layer, and width for CNN stands for number of filters in each convolutional layer. "C" and "F" in the model names for CNNs denote the number of convolutional layers and fully-connected layers respectively. We set a timeout of 1000s for LipMIP and LipSDP, and 60s for BaB. "*" denotes that we report the upper bound LipMIP returns at timeout, and "-" means LipSDP cannot return an upper bound at timeout.

| Method | 4-Layer MLP | | | | | | CNN-2C1F | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Width=32 | | Width=64 | | Width=128 | | Width=4 | | Width=8 | | Width=16 | |
| | Value | Runtime | Value | Runtime | Value | Runtime | Value | Runtime | Value | Runtime | Value | Runtime |
| NaiveUB | 24.31 | 0.05 | 33.02 | 0.01 | 49.39 | 0.00 | 24.38 | 0.00 | 39.99 | 0.01 | 51.68 | 0.03 |
| LipMIP | 12.13 | 16.25 | 102.64 | 1,000.05* | 456.89 | 1,000.12* | 9.54 | 1,000.08* | 57.88 | 1,000.17* | 628.91 | 1,000.40* |
| LipSDP | 21.49 | 11.01 | 27.27 | 103.04 | - | - | 11.03 | 729.92 | - | - | - | - |
| LipBaB | 12.13 | 2.92 | 30.59 | 63.16 | 73.71 | 60.98 | 5.39 | 61.62 | 15.34 | 61.43 | 36.40 | 81.82 |
| RecurJac | 12.38 | 17.40 | 20.25 | 17.07 | 47.23 | 16.66 | 5.02 | 16.21 | 9.02 | 16.73 | 38.19 | 16.38 |
| Ours (w/o BaB) | 12.28 | 6.95 | 17.45 | 6.42 | 35.66 | 6.67 | 4.69 | 7.29 | 8.41 | 7.40 | 30.28 | 7.29 |
| Ours (w/ BaB) | 12.13 | 8.13 | 16.30 | 13.57 | 28.69 | 60.10 | 4.69 | 7.59 | 8.19 | 52.29 | 28.6 | 60.1 |

### 5.1 Comparison on a Synthetic Dataset

We follow Jordan & Dimakis [20] and train several small models on a synthetic dataset. We compare different methods on models with varying width as shown in Table 1, and we also show results with varying depth and $\epsilon$ in Appendix A.1. Since some of these models are already large for LipMIP, we set a time limit of 1000 seconds, and we report the upper bound it returns at timeout. For our BaB, we set a timeout of 60 seconds and stop further branching at timeout.

Without BaB, our method is very efficient while it already outperforms baselines on tightness except for LipMIP on the smallest MLP. Compared to LipMIP, for the smallest MLP on which LipMIP does not timeout, our result without BaB is very close to their exact result; and for other relatively larger models, LipMIP cannot produce exact results within the timeout and our result is much tighter than the upper bounds by LipMIP. And LipSDP fails to yield results within the timeout on many models. Although LipBaB has BaB to tighten bounds, their results are still loose within the time budget, since their BaB is based on loose interval bounds and is thus inefficient. RecurJac is a relatively efficient baseline but still returns looser bounds compared to our method. Our BaB further improves the tightness of our bounds, and our method with BaB consistently produces tightest results within a reasonable time budget, compared to all the baselines.

### 5.2 Comparison on Image Datasets

Table 2: Average local Lipschitz constant values and runtime (seconds) on MNIST. Clean accuracy is evaluated on the entire test set, and Lipschitz constants are evaluated on the first 100 examples in the test set. We set a timeout value of 120s for LipMIP and 60s for BaB. "-" means LipMIP and LipBaB cannot return any valid upper bound at timeout on the CNN model.

| Model | Clean acc. | NaiveUB | | LipMIP | | LipBaB | | RecurJac | | Ours (w/o BaB) | | Ours (w/ BaB) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Value | Runtime | Value | Runtime | Value | Runtime | Value | Runtime | Value | Runtime | Value | Runtime |
| 3-layer MLP | 94.90% | 3,257.16 | 0.00 | 14,218.99* | 120.51 | 947.69 | 62.77 | 1,091.31 | 0.22 | 667.35 | 4.95 | 397.25 | 52.23 |
| CNN-2C2F | 98.83% | 80,239.62 | 0.00 | - | - | - | - | 12,514.55 | 115.43 | 5,473.03 | 8.21 | 5,458.84 | 60.04 |

To further evaluate the scalability of different methods to relatively large models, we then conduct experiments on image datasets including MNIST [26], CIFAR-10 [23], and TinyImagenet [25]. We use a 3-layer MLP and a 4-layer CNN on MNIST, and we use 4-layer and 6-layer CNN models on CIFAR-10 and TinyImageNet. All the models here are too large for LipSDP. LipMIP and LipBaB can only handle the MLP model on MNIST, and they cannot return any valid upper bound for the CNN models within a reasonable time budget. RecurJac can handle the CNN model on MNIST, but other CNN models on CIFAR-10 and TinyImageNet are still too large for RecurJac after converted to equivalent MLP models. In contrast, our formulation for the backward computational graph allows us to utilize the efficient bound propagation for convolutional layers from Xu et al. [45].

Table 3: Average local Lipschitz constant values and runtime (seconds) on CIFAR-10 and TinyImageNet. Clean accuracy is evaluated on the entire test set, and Lipschitz constants are evaluated on the first 100 examples in the test set.

| Dataset | CIFAR-10 | | TinyImageNet | |
|---|---|---|---|---|
| Model | CNN-2C2F | CNN-4C2F | CNN-2C2F | CNN-4C2F |
| Clean accuracy | 76.53% | 78.91% | 29.97% | 29.1% |
| NaiveUB | 1707252.88 | 365293440.00 | 1512185.25 | 87148664.00 |
| RecurJac [a] | 50227.09 | 7735673.50 | 9548.22 | 111570.94 |
| Value (ours) | 18638.14 | 1049447.88 | 4556.61 | 25096.26 |
| Runtime (ours) | 11.17 | 29.79 | 95.42 | 137.04 |

[a] RecurJac's original implementation cannot handle large models here, and its results are obtained by supporting its relaxation in our implementation.

Table 4: Percentage of examples on which the predicted confidence for high income level is monotonically increasing (↑) or decreasing (↓) w.r.t. each feature, by Recurjac and our method respectively.

| Feature | RecurJac | | Ours | |
|---|---|---|---|---|
| | ↑ | ↓ | ↑ | ↓ |
| Age | 32% | 0% | 40% | 0% |
| Education num | 55% | 0% | 58% | 0% |
| Capital gain | 0% | 0% | 5% | 0% |
| Capital loss | 4% | 0% | 7% | 0% |
| Hours-per-week | 95% | 0% | 98% | 0% |

We show results on MNIST in Table 2. Our results are much tighter compared to the baselines even if we do not enable BaB, and BaB can further tighten results. On CIFAR-10 and TinyImageNet, we do not enable BaB on to save time cost on relatively larger models, and we show results in Table 3. Our method can scale to these larger CNNs that previous works could not handle. In addition, we implement RecurJac's looser relaxation in our framework and report its results which are much worse than our results. We present additional results in Appendix.

### 5.3 An Application on Monotonicity Analysis

We further use an application to demonstrate that our proposed method can be used to check the monotonicity of neural networks. Given a network, we aim to check whether the network's output is a monotonic function w.r.t. each input feature. Monotonicity can be preferred properties in some real-world scenarios [33, 8, 27, 35]. In this experiment, we adopt the Adult dataset [4], with a binary classification task about predicting income level. There are only several continuous features, and we aim to check the monotonicity of income level w.r.t. age, education number, capital gain, capital loss, and hours per week. We train an MLP model on this dataset and verify its monotonicity on the first 100 test examples. Signs of Clarke Jacobian can be used to check monotonicity [32], and we achieve this by checking the bounds of Clarke Jacobian. For each continuous feature $j$ we concern, we set an input domain $\mathcal{X}_j$ where only this particular feature can be varied between the minimum and maximum possible values of this feature in the dataset, while we keep other features fixed. And then we obtain bounds on the Clarke Jacobian of the output class for high income level, as $\mathbf{L}_1 \leq \mathbf{J}_1(\mathbf{x}) \leq \mathbf{U}_1$ $(\forall \mathbf{x} \in \mathcal{X}_j)$. Then if $[\mathbf{L}_1]_j > 0$, the predicted confidence on high income level is monotonically increasing w.r.t. feature $j$, and vice versa. We count the percentage of examples that satisfy each type of monotonicity respectively. We show results in Table 4. It suggests that the predicted confidence on high income level tends to be monotonically increasing w.r.t. age, education number on at least part of the examples, and hours per week, which is reasonable. Results also suggest that the models learn little monotonicity on capital gain or loss features. We also compare ours results with those by RecurJac, and our method verifies more monotonicity than RecurJac. We expect the gap to be larger if the models and datasets are larger.

## 6 Conclusion

In this paper, we propose an efficient framework to compute tight $\ell_\infty$ local Lipschitz constants for neural networks by upper bounding the norm of Clarke Jacobian. We model the computation for the Clarke Jacobian as a backward computational graph. On the backward graph, we conduct bound propagation and propose tight linear relaxation for nonlinearities in Clarke Jacobian. We further tighten bounds with branch-and-bound when time budget allows. Experiments show that our method efficiently produces much tighter results compared to existing relaxed methods and can scale to larger convolutional networks on which previous works cannot handle. We also use our method to analyze the monotonicity of neural networks as a potential application. However, it remains challenging to handle even more large-scale models such as those for ImageNet [9], and it will be interesting for future work to further extend the scalability. In addition, we have focused on the $\ell_\infty$ case only, and future work may investigate and solve challenges in extending this work to other norms.

# References


[1] Avant, T. and Morgansen, K. A. Analytical bounds on the local lipschitz constants of relu networks. *arXiv preprint arXiv:2104.14672*, 2021.

[2] Bartlett, P. L., Foster, D. J., and Telgarsky, M. Spectrally-normalized margin bounds for neural networks. 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. 6240–6249, 2017.

[3] Bhowmick, A., D'Souza, M., and Raghavan, G. S. Lipbab: computing exact lipschitz constant of relu networks. In *International Conference on Artificial Neural Networks*, pp. 151–162, 2021.

[4] Blake, C. Uci repository of machine learning databases. *http://www. ics. uci. edu/˜ mlearn/ML-Repository. html*, 1998.

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

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

[7] Clarke, F. H. Generalized gradients and applications. *Transactions of the American Mathematical Society*, 205:247–262, 1975.

[8] Daniels, H. and Velikova, M. Monotone and partially monotone neural networks. *IEEE Transactions on Neural Networks*, 21(6):906–917, 2010.

[9] Deng, J., Dong, W., Socher, R., Li, L., Li, K., and Li, F. Imagenet: A large-scale hierarchical image database. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.

[10] Dwork, C., Hardt, M., Pitassi, T., Reingold, O., and Zemel, R. Fairness through awareness. In *Proceedings of the 3rd innovations in theoretical computer science conference*, pp. 214–226, 2012.

[11] Fazlyab, M., Robey, A., Hassani, H., Morari, M., and Pappas, G. J. Efficient and accurate estimation of lipschitz constants for deep neural networks. In *Advances in Neural Information Processing Systems*, pp. 11423–11434, 2019.

[12] Fel, T., Vigouroux, D., Cadène, R., and Serre, T. How good is your explanation? algorithmic stability measures to assess the quality of explanations for deep neural networks. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pp. 720–730, 2022.

[13] Gómez, F. L., Rolland, P., and Cevher, V. Lipschitz constant estimation of neural networks via sparse polynomial optimization. In *International Conference on Learning Representations*, 2020.

[14] Gouk, H., Frank, E., Pfahringer, B., and Cree, M. J. Regularisation of neural networks by enforcing lipschitz continuity. *Machine Learning*, 110(2):393–416, 2021.

[15] Gowal, S., Dvijotham, K., Stanforth, R., Bunel, R., Qin, C., Uesato, J., Mann, T., and Kohli, P. On the effectiveness of interval bound propagation for training verifiably robust models. *arXiv preprint arXiv:1810.12715*, 2018.

[16] Hein, M. and Andriushchenko, M. Formal guarantees on the robustness of a classifier against adversarial manipulation. 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. 2266–2276, 2017.

[17] Hickey, T., Ju, Q., and Van Emden, M. H. Interval arithmetic: From principles to implementation. *Journal of the ACM (JACM)*, 48(5):1038–1068, 2001.

[18] Huang, Y., Zhang, H., Shi, Y., Kolter, J. Z., and Anandkumar, A. Training certifiably robust neural networks with efficient local lipschitz bounds. In *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021.

[19] Imbert, C. Support functions of clarke's generalized jacobian and of its plenary hull. *Nonlinear Analysis: Theory, Methods and Applications*, 29(8):1111–1125, 2002.

[20] Jordan, M. and Dimakis, A. G. Exactly computing the local lipschitz constant of relu networks. In *Advances in Neural Information Processing Systems*, 2020.

[21] Kang, J., He, J., Maciejewski, R., and Tong, H. Inform: Individual fairness on graph mining. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 379–389, 2020.

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

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

[24] Laurel, J., Yang, R., Singh, G., and Misailovic, S. A dual number abstraction for static analysis of clarke jacobians. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–30, 2022.

[25] Le, Y. and Yang, X. Tiny imagenet visual recognition challenge. *CS 231N*, 2015.

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

[27] Liu, X., Han, X., Zhang, N., and Liu, Q. Certified monotonic neural networks. In *Advances in Neural Information Processing Systems*, 2020.

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

[29] Mirman, M., Gehr, T., and Vechev, M. T. Differentiable abstract interpretation for provably robust neural networks. In *International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 3575–3583, 2018.

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

[31] Salman, H., Yang, G., Zhang, H., Hsieh, C., and Zhang, P. A convex relaxation barrier to tight robustness verification of neural networks. In *Advances in Neural Information Processing Systems*, pp. 9832–9842, 2019.

[32] Schaible, S. et al. Generalized monotone nonsmooth maps. *Journal of Convex Analysis*, 3: 195–206, 1996.

[33] Sill, J. Monotonic networks. *Advances in neural information processing systems*, 10, 1997.

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

[35] Sivaraman, A., Farnadi, G., Millstein, T. D., and den Broeck, G. V. Counterexample-guided learning of monotonic neural networks. In *Advances in Neural Information Processing Systems*, 2020.

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

[37] Tjeng, V., Xiao, K. Y., and Tedrake, R. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*, 2019.

[38] Tsuzuku, Y., Sato, I., and Sugiyama, M. Lipschitz-margin training: Scalable certification of perturbation invariance for deep neural networks. In *Advances in Neural Information Processing Systems*, pp. 6542–6551, 2018.

[39] Virmaux, A. and Scaman, K. Lipschitz regularity of deep neural networks: analysis and efficient estimation. In *Advances in Neural Information Processing Systems*, pp. 3839–3848, 2018.

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

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

[42] Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C.-J., and Kolter, J. Z. Beta-crown: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification. *arXiv preprint arXiv:2103.06624*, 2021.

[43] Weng, T., Zhang, H., Chen, H., Song, Z., Hsieh, C., Daniel, L., Boning, D. S., and Dhillon, I. S. Towards fast computation of certified robustness for relu networks. In *International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 5273–5282, 2018.

[44] Wong, E. and Kolter, J. Z. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 5283–5292, 2018.

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

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

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

[48] Zhang, H., Zhang, P., and Hsieh, C. Recurjac: An efficient recursive algorithm for bounding jacobian matrix of neural networks and its applications. In *The Thirty-Third AAAI Conference on Artificial Intelligence*, pp. 5757–5764, 2019. doi: 10.1609/aaai.v33i01.33015757.

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

# Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] Our work will potentially enable users to better verify and understand the properties of neural networks, for more reliable and trustworthy deployment in the future. We believe there is no negative social impact.

(d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]

2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [N/A] There is no randomness in our computation for local Lipschitz constants.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [Yes]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [Yes]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]