ExLipBaB: Exact Lipschitz Constant Computation for Piecewise Linear Neural Networks

Tom Splittgerber
Competence Center for Clinical Trials Bremen, Faculty 3, University of Bremen, Germany
tsplittg@uni-bremen.de

Abstract

It has been shown that a neural network's Lipschitz constant can be leveraged to derive robustness guarantees, to improve generalizability via regularization or even to construct invertible networks. Therefore, a number of methods varying in the tightness of their bounds and their computational cost have been developed to approximate the Lipschitz constant for different classes of networks. However, comparatively little research exists on methods for exact computation, which has been shown to be NP-hard. Nonetheless, there are applications where one might readily accept the computational cost of an exact method. These applications could include the benchmarking of new methods or the computation of robustness guarantees for small models on sensitive data. Unfortunately, existing exact algorithms restrict themselves to only ReLU-activated networks, which are known to come with severe downsides in the context of Lipschitz-constrained networks. We therefore propose a generalization of the LipBaB algorithm to compute exact Lipschitz constants for arbitrary continuous piecewise linear neural networks and p-norms. With our method, networks may contain traditional activations like ReLU or LeakyReLU, activations like GroupSort or the related MaxMin and FullSort, which have been of increasing interest in the context of Lipschitz-constrained networks, or other piecewise linear functions like MaxPool.

1 Introduction

In recent years, various methods have been developed that require the computation or approximation of a neural network's (NN's) Lipschitz constant. As the Lipschitz constant quantifies a NN's maximum change in output to small changes in input, a core application is the quantification of a NN's robustness to adversarial examples [Szegedy *et al.*, 2014; Cisse *et al.*, 2017; Tsuzuku *et al.*, 2018; Weng *et al.*, 2018a]. Furthermore, it can be used for regularization [Gouk *et al.*, 2021] and the construction of invertible networks and Residual Normalizing Flows [Behrmann *et al.*, 2019; Chen *et al.*, 2019]. Unfortunately, an exact computation of the Lipschitz constant is NP-hard, which was proven in the case of ReLU networks and the $||.||_2$-norm [Virmaux and Scaman, 2018]. The majority of research in the field therefore focuses on scalable methods for increasingly tight approximations of the Lipschitz constant [Fazlyab *et al.*, 2019; Weng *et al.*, 2018b; Shi *et al.*, 2022; Pintore and Després, 2024]. Newly proposed methods usually come with certain theoretical guarantees and are also benchmarked on example networks, ideally with a known true Lipschitz constant. To our knowledge, there are currently only two approaches that are able to exactly compute this true constant for non-constructed examples: LipMIP [Jordan and Dimakis, 2020] and LipBaB [Bhowmick *et al.*, 2021]. LipMIP assumes that the $||.||_1$ or $||.||_\infty$ vector norms are used to define the Lipschitz constant, proposing a generalization to norms which they call "linear". The LipBaB algorithm works for arbitrary p-norms. However, both methods are only applicable to NNs using the ReLU activation. In the context of Lipschitz-constrained NNs, ReLU is however seen with increasing skepticism as authors have proven inapproximability results for layerwise-constrained ReLU-networks [Huster *et al.*, 2019] and raised issues of diminishing gradients for deep networks [Anil *et al.*, 2019].

In practical applications in which the Lipschitz constant only needs to be computed once, like robustness analyses [Pauli *et al.*, 2023] of NNs working with sensitive (small) data (e.g. [Zhang *et al.*, 2024]), which would benefit from an exact computation, as well as in benchmarks mimicking such applications, we therefore argue that there is a need for more general exact computation algorithms to obtain best possible benchmarks and robustness bounds. Even in cases where other activation functions could theoretically be expressed as ReLU-networks, this would result in a sharp increase in computational cost due to increased network size and existing methods would nonetheless have to be adapted to residual networks [Pauli *et al.*, 2024]. We therefore propose in this paper a generalization of the LipBaB algorithm [Bhowmick *et al.*, 2021], which we call **Ex**tended **LipBaB** (ExLipBaB)[1] and which is able to compute the exact local and global Lipschitz constant w.r.t. arbitrary p-norms for any continuous piecewise linear NN . ExLipBaB is therefore in particular also applicable to networks using the GroupSort activation

[1] Supplementary material, including the code for our experiments can be found under: https://github.com/tsplittg/ExLipBaB_Code.

or the related FullSort and MaxMin, which are of particular interest in the field of Lipschitz constraints [Anil *et al.*, 2019; Pauli *et al.*, 2024]. Networks may also include general continuous piecewise linear functions like MaxPool or learned spline activations [Tao *et al.*, 2022; Ducotterd *et al.*, 2024]. Like [Bhowmick *et al.*, 2021], we leverage the Branch-and-Bound framework to split the input space of piecewise linear NNs into local regions on which the Lipschitz constant can be easily computed. This framework results in an exponential runtime in the worst case but a faster runtime in most practical applications, as not all possible regions will actually have to be evaluated. In cases where running our algorithm to completion would be computationally infeasible, it can be stopped at any time and still deliver a hard upper and lower bound for the Lipschitz constant for a highly flexible class of NNs and arbitrary p-norms.

2 Preliminaries

We first define the following [Jordan and Dimakis, 2020]:

Definition 1. *The Lipschitz norm of a function $f : (\mathbb{R}^d, ||.||_p) \to (\mathbb{R}^m, ||.||_q)$ over an open set $\Omega \subset \mathbb{R}^d$ is defined as:*

$$L_{p\to q}(f,\Omega) := \sup_{x_1\neq x_2\in\Omega} \frac{||f(x_1)-f(x_2)||_q}{||x_1-x_2||_p} \quad .$$

If this supremum exists and is finite, f is called Lipschitz continuous on Ω.

For better readability, we will usually omit the subscripts p, q and $p \to q$. Also, as is common, we will use the term *Lipschitz constant* when referring to the *Lipschitz norm*. If the set Ω is equal to the entire space $\mathbb{R}^d$, this is often referred to as the *global* Lipschitz constant. In the more general case in which Ω is an arbitrary open subset, $L(f,\Omega)$ is referred to as the *local* Lipschitz constant [Jordan and Dimakis, 2020; Bhowmick *et al.*, 2021]. This should not be confused with the mathematical concept of local Lipschitz continuity.

Many approximation approaches, like the basic layerwise approximation [Gouk *et al.*, 2021] or LipSDP [Fazlyab *et al.*, 2019; Pauli *et al.*, 2024] only aim at computing the global Lipschitz constant. LipMIP [Jordan and Dimakis, 2020] on the other hand is able to exactly compute the local Lipschitz constant w.r.t. "linear" norms, whereas the LipBaB approach [Bhowmick *et al.*, 2021] is able to exactly compute the local and global constant w.r.t. arbitrary p-norms; both algorithms are however only applicable to ReLU networks.

For our generalization of the LipBaB approach, we assume that all functions contained in our NN are continuous piecewise linear (PWL) [Tao *et al.*, 2022; Gorokhovik *et al.*, 1994]:

Definition 2. *Let $\Omega \subset \mathbb{R}^{d_1}$, then a function $\alpha : \Omega \to \mathbb{R}^{d_2}$ is called PWL if there exists a set $\{\mathcal{P}_1, \dots, \mathcal{P}_k\}$ of polyhedra (we will always assume convex polyhedra) with interiors $\mathcal{P}_i^\circ$ such that:*

i) $\bigcup_{i=1}^k \mathcal{P}_i = \Omega$ and $\mathcal{P}_i^\circ \neq \emptyset \ \forall i = 1, \dots, k$

ii) $\mathcal{P}_i^\circ \cap \mathcal{P}_j^\circ = \emptyset$ for all $i \neq j$

iii) For all $i = 1, \dots, k$, the function α restricted to $\mathcal{P}_i$, is an affine function, i.e. there exist $T^i \in \mathbb{R}^{d_2\times d_1}, t^i \in \mathbb{R}^{d_2}$ such that $\alpha|_{\mathcal{P}_i}(x) = T^i x + t^i$.

In the following sections, we will always assume that we are working with a neural network $f : \mathbb{R}^{d_0} \to \mathbb{R}^{d_L}$ of the form:

$$f(x) = \alpha_L \circ \mathfrak{W}_L \circ \alpha_{L-1} \cdots \circ \alpha_1 \circ \mathfrak{W}_1(x), \qquad (1)$$

where each $\mathfrak{W}_i$ is an affine function $\mathfrak{W}_i(x) = W_i x + w_i$ corresponding to a linear layer and each α_i is a continuous PWL function. One can easily see that the assumption that the linear and PWL layers alternate comes without loss of generality, as we can simply either pad layers with the identity function or join layers (concatenations of affine/PWL functions are again affine/PWL).

NNs based on PWL activations are commonplace in modern machine learning. An overview over the concept can be found in [Tao *et al.*, 2022]. We prove in the supplementary material that componentwise PWL activations like ReLU, LeakyReLU or ParametricReLU (with a slope learned during training) as well as GroupSort (and the related FullSort and MaxMin) satisfy definition 2. A similar argument for the MaxPool function can be found in [Gehr *et al.*, 2018].

For the ReLU activation in d dimensions, a full decomposition of the input space into linear regions would have 2^d polyhedron elements. For use in our algorithm, we will therefore use an equivalent decomposition, where the activation state is only fixed for a subset of neurons in each region. This transfers the approach of [Bhowmick *et al.*, 2021] into the general PWL setting:

Remark 1. *Let $\Omega \subset \mathbb{R}^{d_1}$, then a function $\alpha : \Omega \to \mathbb{R}^{d_2}$ is called PWL if there exists a set $\{\mathcal{Q}_1, \dots, \mathcal{Q}_{\tilde{k}}\}$ of polyhedra such that for every neuron $n \in \{1, \dots, d_2\}$, there exists a subset $\mathcal{I}_n \subset \{1, \dots \tilde{k}\}$, such that:*

1. *$\bigcup_{i\in\mathcal{I}_n} \mathcal{Q}_i = \Omega$*
2. *for all $i \in \mathcal{I}_n$, there exist $T^{ni} \in \mathbb{R}^{1\times d_1}, t^{ni} \in \mathbb{R}$, such that $\pi_n \circ \alpha|_{\mathcal{Q}_i}(x) = T^{ni}x + t^{ni}$ is an affine function, where π_n is the projection onto the n-th vector element.*

2.1 Lipschitz Constant and Jacobian

It is well known that the Lipschitz constant of a linear function $A(.) + b : (\mathbb{R}^d, ||.||_p) \to (\mathbb{R}^m, ||.||_q)$ equals the induced matrix norm $||A||_{p\to q} := \max_{x\neq 0} \frac{||Ax||_q}{||x||_p}$. Such matrix norms are comparatively easy to compute for common p and q [Johnston, 2016]. Our goal is therefore to reduce the computation of $L(f,\Omega)$ to computing the maximum over all matrix norms in the PWL decomposition of f:

Theorem 1. *Let $\alpha : (\mathbb{R}^{d_1}, ||.||_p) \to (\mathbb{R}^{d_2}, ||.||_q)$ be a PWL function. Using the notation from definition 2, the following holds:*

$$L_{p\to q}(\alpha,\Omega) = \max_{i=1,\dots,k} ||T^i||_{p\to q} \quad . \qquad (2)$$

The proof of this theorem is very similar to the arguments used in [Bhowmick *et al.*, 2021] and therefore not discussed in detail here - we instead refer to the supplementary material. Essentially, we use a result from [Jordan and Dimakis, 2020] to relate the Lipschitz constant of f to the supremum over the norm of its generalized (Clarke) Jacobian over Ω. Then, we show that only differentiable points need to be considered for this supremum, which are exactly the points in the interior of polyhedra $\mathcal{P}_i$ in the notation of definition 2.

Algorithm 1 Overview ExLipBaB

Input: Network f, input polyhedron Ω, approximation factor θ

Output: Tuple (glb, gub); global lower- and upper bound for $L(f, \Omega)$ where gub $\leq \theta \cdot$ glb

initial activation pattern $\leftarrow$ SymProp(Ω, f)
initial subproblem $\rho_I \leftarrow$ SubProblem($\Omega, \dots$)
$\rho_I.L_{ub} \leftarrow$ LipschitzBounds($\rho_I, \dots$)
initialize glb, gub $\leftarrow (0, \rho_I.L_{ub})$
initialize: $\varrho \leftarrow$ maxHeap($[\rho_I]$)
if $\tilde{l}$ == L+1 **then**
    glb $\leftarrow \rho_I.L_{ub}$ //trivial case
end if
while gub $> \theta \cdot$ glb **do**
    $\rho' \leftarrow \arg\max_{\rho \in \varrho}(\rho.L_{ub})$
    $\varrho \leftarrow \varrho \backslash \{\rho'\}$
    (new_subproblems, glb) $\leftarrow$ Branch($\rho', \dots$)
    $\varrho \leftarrow (\varrho \cup$ new_subproblems)
    gub $\leftarrow \max_{\rho \in \varrho} \rho.L_{ub}$
end while
return (glb, gub)

3 Methodology

In theorem 1, we have reduced the computation of $L(f, \Omega)$ to the discrete optimization problem $\rho_I : \max_{i \in \mathcal{S}} ||T^i||$, with $\mathcal{S} := \{1, \dots, k\}$ the finite index set of all polyhedra $\mathcal{P}_i$ in the PWL decomposition of the NN f. Like [Bhowmick *et al.*, 2021], we follow the standard branch-and-bound (BnB) framework to recursively solve this problem. A high-level overview of our algorithm is given in Algorithm 1.

At the core of our algorithm is the set ϱ of active subproblems, each representing a discrete optimization problem over a subset $\mathcal{S}_\rho \subseteq \mathcal{S}$. We will impose that the input space associated with each $\rho \in \varrho$ in $\mathbb{R}^{d_0}$ is always a polyhedron. We follow [Lee, 2004], and describe the three main components of our BnB algorithm:

1. *Bounding* $\rho \in \varrho$ with good upper bounds, which is described in section 3.3
2. *Branching* a subproblem $\rho \in \varrho$ into new subproblems ρ' with $\mathcal{S}_{\rho'} \subseteq \mathcal{S}_\rho$, such that an optimal solution for any ρ' is also an optimal solution to ρ, see section 3.4
3. An efficient method to obtain *lower bounds*, see section 3.2

The set ϱ is initialized as $\varrho = \{\rho_I\}$, where the bounding of ρ_I is described in section 3.2. Our algorithm then maintains a global upper bound (gub) and global lower bound (glb) over all subproblems. In the recursive step, the subproblem $\rho \in \varrho$ with the largest upper bound is removed from ϱ and branched into subproblems associated with smaller regions of the input space. For each smaller subproblem ρ', an upper bound is computed and the gub is updated accordingly. Also, if the upper bound is smaller than the glb, we discard ρ' (*fathomed by bounds*). If we can solve ρ' exactly, we also return a lower bound LB and update the glb as $\max(\text{glb}, LB)$; afterwards, we again discard ρ' (*fathomed by optimality*). If a ρ' is not discarded by either of these rules, it is appended into ϱ. The algorithm runs until all subproblems are fathomed or until $\theta \cdot \text{glb} \geq \text{gub}$ for a pre-determined approximation factor θ. Either way, termination is guaranteed and if the algorithm is run to completion, the true maximum is found [Lee, 2004; Gendron and Crainic, 1994]. In the following sections, we describe the components of our algorithm in more detail.

3.1 Abstract Interpretation

At creation, each subproblem ρ is assigned an associated polyhedron $\Omega_\rho \subset \Omega$ in input space. We consider ρ exactly solvable, if f, constrained to Ω_ρ, is fixed affine, in which case we only have to compute a single matrix norm to solve ρ.

To determine whether f is fixed affine on Ω_ρ, we compute a list of neurons in f which we call $\star$-neurons, which potentially take different activation states in Ω_ρ. For this, we leverage the concept of abstract interpretation [Gehr *et al.*, 2018]. Given a layer index $1 \leq l \leq L$, we wish to find an *abstract transformer* $f_l^\#$ that transforms the polyhedron Ω_ρ of possible inputs into f to a d_l-dimensional polyhedron $f_l^\#(\Omega_\rho)$ of potentially possible pre-activation states in the l-th layer. We can then intersect $f_l^\#(\Omega_\rho)$ with all polyhedra in the PWL decomposition of α_l. If for a neuron n, there are at least two polyhedra with non-empty intersection with $f_l^\#(\Omega_\rho)$, such that the activation state of n on these polyhedra is different, then we call n a $\star$-neuron.

An abstract transformer $f_l^\#$ can simply be constructed by concatenating abstract transformers of the individual layers of f. For linear layers, the affine transformation of a polyhedron automatically is a polyhedron. For PWL layers α_m, the polyhedron of potential pre-activations is intersected with all polyhedra of the PWL decomposition of α_m. Then, each intersection is transformed with the corresponding fixed affine state of α_m and finally a polyhedron is constructed that is a superset of all these smaller transformed polyhedra. This process is described in more detail in [Gehr *et al.*, 2018] (note that what the authors call "CAT" functions is simply a subclass of PWL functions).

For each layer, we record the minimum and maximum value of the activation weights and biases for each neuron over all non-empty intersections in an interval matrix $[\Lambda]_l$ and an interval vector $[\lambda]_l$ respectively. In other words, these interval matrices (vectors) contain at each entry an interval of plausible values for the activation state at the given entry (see e.g. [Rump, 2012]). We also record with $\tilde{l}$ the first layer of f in subproblem ρ which contains $\star$-neurons.

In abstract interpretation, one often restricts the type of abstract sets to be boxes or zonotopes instead of polyhedra. This drastically improves performance at the cost of looser bounds, which is why the initialization algorithm which we describe in section 3.2 uses the box region for all newly added abstract variables. Note that the tightness of the bounds in the initial symbolic propagation only affects initialization and does not alter the exactness of the final result.

3.2 Initialization

To initialize the set of subproblems ϱ, an activation pattern and an upper bound have to be computed for the initial subproblem and an initial glb has to be specified. In [Bhowmick

et al., 2021], the initial glb is simply set to zero and updated only when a subproblem is solved exactly. We instead propose to evaluate the norm of the Jacobian of f in a few randomly chosen points in Ω. This is computationally inexpensive and, due to theorem 1, results in a hard lower bound if f is differentiable in all chosen points. Theoretically, the probability of accidentally choosing a non-differentiable point is zero [Jordan and Dimakis, 2020]; in practice, due to floating point inaccuracy, it is only very small. Also, the case of accidentally selecting non-differentiable points could easily be prevented algorithmically. We test this non-trivial lower bound initialization in practice in section 4.3.

For the computation of the initial activation pattern, we adapt the SymProp algorithm from [Bhowmick *et al.*, 2021] to the general PWL setting. This algorithm makes use of symbolic propagation [Li *et al.*, 2019] and does not propagate explicit polyhedra through f as was described in section 3.1. Instead, it propagates linear relations whenever possible. This approach can help to reduce the overapproximation resulting from the well-known dependeny problem. The idea is to represent the state of a neuron as an abstract linear equation of input variables if the activation state of f is fixed affine on that neuron. If the activation state of the neuron isn't fixed, a new abstract input variable is introduced to preserve linear relations in deeper layers. We detail the propagation through the first layer, the other layers follow recursively:

For f, we assume the notation of equation (1). The input is represented as a d_0-dimensional symbolic variable that takes values in a polyhedron Ω_ρ. Let C, c be the half space representation of Ω_ρ, i.e. $\Omega_\rho = \{x \in \mathbb{R}^{d_0} : Cx \leq c\}$ and define $\widetilde{B} := W_1$, $\widetilde{b} := w_1$. Denote also by A_i, a_i the half space constraints of the i-th polyhedron $\mathcal{P}_i$ in the PWL decomposition of α_1 according to remark 1. Formally, all of these variables need an additional index because they are iterated upon in the recursion, but to improve readability we will slightly abuse notation and omit these indices. We then first determine which activation states α_1 possibly takes. For this, we set

$$\widetilde{A}_i := \text{stack}(A_i \cdot \widetilde{B}, C) \quad , \quad \widetilde{a}_i := \text{stack}(a_i - A_i \cdot \widetilde{b}, c) \quad ,$$

where stack(.) represents the vertical stacking of two matrices. It holds that

$$\mathfrak{W}_1(\Omega_\rho) \cap \mathcal{P}_i \neq \emptyset \iff \{x \in \mathbb{R}^{d_0} : \widetilde{A}_i \cdot x \leq \widetilde{a}_i\} \neq \emptyset. \quad (3)$$

Let $\mathcal{I}$ be the set of indices of the polyhedra that fulfill either side of equation (3). The candidate set of $\star$-neurons in the current layer is then the set of neurons whose state is determined in at least two $\mathcal{P}_i, i \in \mathcal{I}$. In order to reduce the number of unnecessary splits, we however only classify a neuron as $\star$-neuron, if the activation states differ in these polyhedra. Using the notation from remark 1, we demand that either T^{ni} or t^{ni} differ between at least two polyhedra. This check is necessary for certain decompositions and less so for others; for example, in the full PWL decomposition of a d-dimensional ReLU activation according to definition 2, any neuron is fixed active in 2^{d-1} polyhedra whereas we describe an alternative decomposition in the supplementary material where it is only fixed active in exactly one polyhedron.

Let $\mathcal{J}$ be the index set of $\star$-neurons at the current layer. For a neuron $n \notin \mathcal{J}$, we denote by T^{ni} and t^{ni} the n-th row and n-th bias element of the unique activation state of α on n respectively. To propagate the symbolic linear relations on non-$\star$-neurons through α_1, we can simply apply these affine functions. For $\star$-neurons however, we can no longer maintain linear relationships. Like [Li *et al.*, 2019] suggests, we therefore concretize the post-activation bounds of these neurons and add new symbolic variables to the input set that take values within these bounds. Thereby, we hope to maintain linear relations in following layers.
Let $j \in \mathcal{J}$ and denote by $\min(j), \max(j)$ the minimal and maximal post-activation value of this $\star$-neuron. We then form $\widehat{\Omega}_\rho := \Omega_\rho \times \left(\times_{j \in \mathcal{J}} [\min(j), \max(j)]\right)$, which is a polyhedron of dimensionality $\widehat{d_0} := d_0 + |\mathcal{J}|$.

Let T be the $d_1 \times d_0$ matrix and t be the d_1-dimensional vector, whose rows, respectively entries, are zero for $\star$-neuron indices and equal to T^{ni}, respectively t^{ni}, for neurons with a unique activation state. We then define a new vector $\widehat{b}$ of length d_1 and a matrix $\widehat{B}$ of shape $d_1 \times \widehat{d_0}$, by setting $\widehat{b} := T \cdot \widetilde{b} + t$ and, for $(n, j) \in d_1 \times \widehat{d_0}$:

$$\widehat{B}_{nj} := \begin{cases} (T \cdot \widetilde{B})_{nj} & \text{, if } j \leq d_0 \\ \mathbb{1}_{i=(j-d_0)} & \text{, else} \end{cases} \quad (4)$$

It is easy to see that the propagation via these matrices is an overapproximation of the propagation by α_1 in the sense that $\alpha_1 \circ \mathfrak{W}_1(\Omega_\rho) \subset \widehat{B} \cdot \widehat{\Omega} + \widehat{b}$. In the previously mentioned slight abuse of notation, we now close the recursive loop by renaming

$$\Omega_\rho := \widehat{\Omega}_\rho, \quad d_0 := \widehat{d_0}, \quad \widetilde{B} := W_2 \cdot \widehat{B}, \quad \widetilde{b} := W_2 \cdot \widehat{b} + w_2.$$

While recursively continuing this propagation through the entire network, we save the indices and layers of the $\star$-neurons and the layer $\widetilde{l} \in \{1, \dots L\}$ in which the first $\star$-neuron appears. We also compute two lists of interval matrices: $[\Lambda]$ and $[\lambda]$. They contain, for each layer, an interval matrix and an interval vector which each contain as their entries an interval defined by the maximum and minimum entries over all possible linear activation states T^{ni} and t^{ni}. These interval matrices are necessary for section 3.3. Note that, unlike in [Bhowmick *et al.*, 2021], the interval matrices we obtain in our algorithm are not necessarily diagonal matrices.

3.3 Lipschitz Bounds

To compute the upper Lipschitz bound for an individual subproblem ρ with associated region Ω_ρ, we first assume that the activation state of f on Ω_ρ, in the form of $\widetilde{l}$, as well as $[\Lambda]$ and $[\lambda]$, has already been computed. These computations are either done in SymProp or FFilter (see section 3.4 below).

As $\widetilde{l}$ is the first layer index with $\star$-neurons, the function $\widetilde{f} := (\mathfrak{W}_{\widetilde{l}} \circ \alpha_{\widetilde{l}-1} \circ \dots \circ \mathfrak{W}_1)|_{\Omega_\rho}$ is affine (where we define $\mathfrak{W}_{L+1}$=Id). We can therefore easily compute the Jacobian $J_{\widetilde{f}}$ of $\widetilde{f}$ through a simple matrix product.

If $\widetilde{l} = L + 1$, i.e. if the network contains no $\star$-neurons on Ω_ρ, then $f = \widetilde{f}$ and we simply return $||J_{\widetilde{f}}||$.

If $\widetilde{l} \leq L$, we leverage interval matrix multiplication to construct a matrix which is in each component absolutely larger than J_f. In interval matrix multiplication, the product of two interval matrices $[A], [B]$ is again an interval matrix which, as its components, contains intervals that overapproximate all possible results $A \cdot B$, where A and B have entries from the intervals of $[A]$ and $[B]$. This product can be computed through a brute force combinatorial approach which is intuitive but lengthy, which is why we refer to [Diep, 2012] for a more comprehensive introduction.

We now define an initial interval matrix $[J_{\widetilde{l}}] := [J_{\widetilde{f}}, J_{\widetilde{f}}]$ and recursively compute

$$[J]_{l+1} = [\Lambda]_l \cdot [W_l, W_l] \cdot [J_l] \tag{5}$$

for $l = \widetilde{l} + 1, \ldots, L$. Notice that, even though we allow both our linear and activation layers to include biases, their presence does not matter for the computation of the (approximation of the) Jacobian. Finally, we set U as the $(d_L \times d_0)$-matrix with entries $U_{ij} := \max(|\underline{[J_{ij}]_L}|, |\overline{[J_{ij}]_L}|)$.

If now $x \in \Omega_\rho$ is an arbitrary point in which f is differentiable, then there exists a polyhedron $\mathcal{P} \subset \Omega_\rho$ on which f is fixed affine, such that $x \in \mathcal{P}^\circ$. Then, for each layer l, the PWL function α_l takes a fixed affine form $A_l x + b_l$ with $A \in [\Lambda]_l$, where we slightly abuse notation to denote by "$\in$" a realization of the interval matrix. Then, due to the properties of interval matrix multiplication,

$$J_f(x) = A_L \circ \cdots \circ W_1 \in [\Lambda]_L \circ \cdots \circ [W_1, W_1] = [J_L]. \tag{6}$$

Therefore, by definition of U, it holds for all i, j that $J_f(x)_{ij} \leq U_{ij}$.

The following lemma therefore proves that $||U||$ is a hard upper bound for $L(f, \Omega_\rho)$:

Lemma 1 ([Bhowmick *et al.*, 2021, Lemma 2])**.** *Let U be a matrix of the same size as the Jacobian $J_f(x)$ of a PWL function f. If U is such that $\sup_{x \in \Omega} |J_f(x)_{ij}| \leq U_{ij}$ for all i, j and all points x in which f is differentiable, then $L_{p \to q}(f, \Omega) \leq ||U||_{p \to q}$.*

In [Bhowmick *et al.*, 2021], this lemma is only proven for $p = q$. However, as their proof easily transfers to the general case, we only discuss it in the supplementary material.

3.4 Branching

After having described the bounding step, we now describe the branching step of the recursion in more detail. At a given iteration, given the current set of subproblems ϱ, we first remove the subproblem ρ with the maximal upper bound from ϱ. We assume that $[\Lambda], [\lambda]$ and $\widetilde{l}$ have been computed for ρ in an earlier iteration and that ρ is associated with a polyhedron Ω_ρ. As ρ will not have be solved, we can get the index n of the first $\star$-neuron in layer $\widetilde{l}$. We then iterate through all polyhedra $\mathcal{P}_i$ in the PWL decomposition of $\alpha_{\widetilde{l}}$ with respect to neuron n as defined in remark 1. Similar to our approach in equation (3), we stack the half-space constraints of Ω_ρ with the constraints of $\mathcal{P}_i$, propagated back to the input dimension, to obtain a new polyhedron $\mathcal{Q}_i$. This propagation is possible because, on ρ, f is fixed affine in all layers before $\widetilde{l}$. If the intersection defined by the stacking of constraints is not empty, we initialize a new subproblem ρ_i associated with $\mathcal{Q}_i$ in input space. Since $\mathcal{Q}_i \subset \Omega_\rho$, we know that the Lipschitz constant of each new subproblem is not larger than that of ρ, meaning that a feasible solution of the new subproblems is also a feasible solution of ρ. Also, per definition, each ρ_i has strictly fewer $\star$-neurons than ρ as the state of n is fixed. We then update the activation states of ρ_i (see below) and compute the upper (lower) bounds. Afterwards, each new subproblem is either terminated according to the rules at the beginning of section 3 or added to ϱ, which completes the recursion.

Note that iterating through all polyhedra in a decomposition according to remark 1 can be intuitively understood as setting the activation state of only that neuron (plus some others that automatically get set with it, depending on the PWL function). One hopes that the constriction enforced by setting one neuron also results in the fixing of others. One could also use a decomposition as in definition 2 to force this, but this would imply a drastic increase in the number of necessary polyhedra. The advantage would be that we would be certain that layer $\widetilde{l}$ is fixed in all newly created subproblems.

FFilter After the branching step, we run a feasibility filter, which combines the LinProp and FFilter algorithms of [Bhowmick *et al.*, 2021], to check whether we have also implicitly set the state of other neurons. We multiply the first $(\widetilde{l} - 1)$ layers, on which f is linear, into one linear function and then, starting with layer $\widetilde{l}$, check whether the other layers contain $\star$-neurons or not. Also, $[\Lambda]_{\widetilde{l}}$ and $[\lambda]_{\widetilde{l}}$ are updated. If a layer does not contain $\star$-neurons, its linear state is multiplied with the propagated linear function and $\widetilde{l}$ increased by one. We terminate this process if we encounter a layer with $\star$-neurons. Activation states of deeper layers are not updated.

3.5 Convergence Guarantee

After the algorithm has been run - either to completion or until a fixed approximation quality has been reached - the global upper and lower bounds are returned. If the algorithm is run to completion, these two bounds match and are an exact solution to the maximization problem in equation (2):

Lemma 2. *The solved subproblem with the greatest upper bound is an optimal solution of the maximization problem, i.e. its upper bound equals the true Lipschitz constant of f.*

This is a standard result from Branch-and-Bound theory [Lee, 2004]. We also mention that our algorithm suffices the theorem of [Gendron and Crainic, 1994]:

Proof. As we only implemented an elimination via upper and lower bounds, we only have to check the first two convergence assumptions of [Gendron and Crainic, 1994]. Firstly, our algorithm only returns a lower bound for a subproblem ρ, if ρ contains no $\star$-neurons. In such a case, the lower bound equals the true Lipschitz constant of f constrained to Ω_ρ and is therefore a feasible solution to the original problem. Secondly, a subproblem is solved if and only if its upper and lower bound have been computed and are equal. □

In particular, if the initial input space is set as $\Omega = \mathbb{R}^{d_0}$, our method computes the true global Lipschitz constant.

	ReLU		GroupSort	
	$L(f)$[std]	time	$L(f)$[std]	time
Layerwise	3.12 [0.50]	0.00s	1.62 [0.32]	0.00s
ExLipBaB	1.15 [0.20]	0.03s	1.01 [0.24]	0.03s
LipSDP	1.62 [0.32]	3.63s	1.30 [0.30]	0.00s

Table 1: Mean [std] global Lipschitz constant estimate of different methods over 20 runs of NNs trained on synthetic data from the absolute value function. Listed: ReLU-NN (mean test RMSE: 0.11, std: 0.04) and GroupSort NN (mean test RMSE: 0.11, std: 0.05).

Abalone, Shape $[9, 16, 16, 1]$

$\mathbb{R}^{d_0}$	ReLU		MaxMin	
	estimate	time (s)	estimate	time (s)
Layerwise	14.27	0.000	10.41	0.000
LipSDP	11.22	30.72	8.827	0.016
SymProp	15.22	0.017	8.918	0.015
ExLipBaB	11.21	20.83	4.487	357.8

Table 2: Lipschitz constant estimates for NNs trained on the Abalone dataset w.r.t. $||.||_2$

4 Experiments

In this section, we apply the ExLipBaB algorithm to networks trained on various datasets and compare the obtained Lipschitz constants to other approximation methods. However, as a test of basic correctness, we first applied our method to the four ReLU-activated networks for which exact Lipschitz constants were reported in [Bhowmick *et al.*, 2021]. We are happy to report that our approach is able to replicate the Lipschitz constants computed by their algorithm to over 10 decimals for all tested norms ($||.||_1, ||.||_2, ||.||_\infty$).

In the other listed results, we frequently compare our method to the LipSDP approach, which is a widely cited global approximation approach. For ReLU networks, we use the original implementation [Fazlyab *et al.*, 2019], if not otherwise mentioned in the most accurate "network" variant. For GroupSort networks with group size two, we use the recently proposed generalization of LipSDP [Pauli *et al.*, 2024], which is based on the more scalable "neuron" variant. All computations were performed under Windows 11 on an AMD Ryzen 9 7900X3D CPU with 32 GB of RAM. The code for all simulations can be found in the supplementary material.

4.1 Absolute Value Function

The absolute value function $|.|$ is of specific importance in the context of Lipschitz-constrained NNs. It has been shown that a ReLU-NN which is constrained via layerwise approximation to be Lipschitz continuous with a constant less than 2 cannot learn this 1-Lipschitz function [Huster *et al.*, 2019]. Other activations, such as GroupSort, do not suffer from this and other theoretical limitation(s) [Anil *et al.*, 2019].

To put these theoretical results into perspective, we show in Table 1 the global Lipschitz constants of two unconstrained NNs, trained on data from $|.|$ for 20 random initializations. It can be seen that the ReLU networks, which had learned $|.|$ reasonably well, had a true Lipschitz constant very close to one for most random seeds. However, in the layerwise approximation, their Lipschitz constant was overestimated by almost 200% on average. For GroupSort networks, the true constant was "only" overestimated by, on average, 60%. It is therefore important to highlight that the aforementioned inapproximability results only speak to the inadequacy of layerwise-constrained ReLU-networks. Still, as restricting a NN's Lipschitz constant usually requires its computation at every iteration of the optimizer, the computationally efficient layerwise approximation will likely remain one of the most common ways of Lipschitz constant estimation during training. Therefore, the ReLU activation remains non-ideal for Lipschitz-constrained NNs.

4.2 Real Data Applications

For realistic data applications, we trained generic, unconstrained NNs on the Abalone [Warwick *et al.*, 1994], Wine Quality [Cortez *et al.*, 2009] and Bike Sharing [Fanaee-T and Gama, 2014] datasets. We then compare the exactly computed Lipschitz constant with a layerwise approximation and estimations by SymProp and LipSDP. We always use the $||.||_2$-norm and, to give a meaningful comparison, always consider the global Lipschitz constant on $\mathbb{R}^{d_0}$ if not otherwise specified. In section 4.4, we discuss the implications of different input region sizes for the ExLipBaB algorithm.

As can be seen in the listed results, the bounds approximated by the LipSDP algorithm were relatively tight relative to exact computation in many cases, especially when compared to the layerwise and SymProp approximations. In the relatively simple Abalone/ReLU case (Table 2), the approximation was almost exact. However, in the moderately more complex small Wine/ReLU example (Table 3) and large Wine/GroupSort (Table 4), the approximation by LipSDP was improved by 10% through exact computation and in the Abalone/GroupSort example by almost 50%.

It is also important to mention that the exact global computation through ExLipBaB did not fully converge in two cases (see Tables 4 and 5) and the LipSDP algorithm did not (properly) converge in one case (Table 5). We discuss the impact of the size of the input region in section 4.4.

Generally, one can unsurprisingly see that an increase in network size (Table 3 vs. Table 4 and Table 6.A) results in an increase in runtime. However, dataset complexity seems to be even more impactful (compare Table 2 vs. Table 3), likely due to a more segmented PWL representation.

4.3 Lower Bound Effectiveness

We also test the effect of a simple initial lower bound as proposed in section 3.2 by comparing the cardinality of ϱ during a full run of ExLipBaB with and without an initial lower bound computed on $5 \cdot 10^6$ grid points. On the GroupSort NN from Table 4, the lower bound computation took 5.66 seconds, resulting in a bound of 1.629 and only one fewer subproblem was added to ϱ. On the ReLU NN from Table 3, the lower bound computation took 2.65 seconds with a bound of 1.677. As can be seen in figure 1, this resulted in a drastic reduction of the cardinality of ϱ but the number of iterations

Wine, Network shape $[11, 12, 12, 1]$				
$\mathbb{R}^{d_0}$	ReLU		MaxMin	
	estimate	time (s)	estimate	time (s)
Layerwise	5.86	0.000	3.78	0.000
LipSDP (*)	1.84	26.65	2.87	0.014
LipSDP (**)	1.30	4.257	-	-
SymProp	4.78	0.015	11.09	0.013
ExLipBaB	1.74	36.65	2.79	2.444

Table 3: Global Lipschitz constant estimates for NNs trained on the Wine dataset w.r.t. $||.||_2$. LipSDP ("network") (**) converged to an implausible value, therefore we also list the "neuron" variant (*).

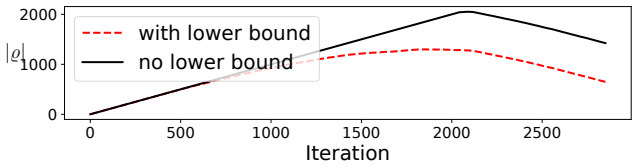

Figure 1: Number of subproblems for ExLipBaB on the ReLU-NN from Table 3 with and without initial lower bound.

needed for an exact ExLipBaB run did not change. The usefulness of a lower bound seems therefore highly dependent on its quality and will most likely only lie in a reduction of required memory. Given its comparatively quick computation time, there is however little downside to using it and especially for low-dimensional inputs a reasonably good coverage of the input space can likely be obtained fairly easily.

4.4 Influence of Input Space Size

Unlike the layerwise and LipSDP estimators, our method is not just able to compute the global Lipschitz constant $L(f, \mathbb{R}^{d_0})$, but also local constants $L(f, \Omega)$ for $\Omega \subsetneq \mathbb{R}^{d_0}$. Since generally only a subset of linear regions of f intersects Ω, this can drastically reduce computational cost. Also, local robustness guarantees can potentially be more helpful (tighter) than global ones. We illustrate the aforementioned implications of the size of Ω in Table 6.B for the GroupSort NN from Table 4 and show in Table 6.A the increasing computational cost for an increasing network size. The latter is also further discussed in the supplement.

5 Conclusion

The ExLipBaB algorithm we discussed can compute the exact Lipschitz constant for general continuous piecewise linear NNs. This generalization is of special importance in the field of Lipschitz-constrained NNs in which the ReLU function, which most existing methods are tailored towards, has been shown to have theoretical limitations.

Our generalization also allows a relatively straightforward extension to convolutional and residual NNs in future research. The importance of Lipschitz constant estimation for convolutional neural networks (CNNs) specifically, even for small ones, has been a topic of study in recent years, often in the context of adversarial robustness. In the context of residual NNs, Lipschitz constant estimation is often performed in the method of Residual Normalizing Flows. Such Flows need to keep a balance in their allowed Lipschitz constant to be both invertible and expressive. Both fields would benefit from tighter Lipschitz bounds. However, future research would likely have to leverage the specific structure and sparsity of such NNs, as Branch-and-Bound methods would likely not scale to any but the smallest networks otherwise. Still, our algorithm lends itself naturally to parallelization. In combination with reduced RAM requirements through tighter lower bounds, this could allow computations for more complex NNs and larger input regions.

Wine, Network shape $[11, 24, 24, 1]$				
$\mathbb{R}^{d_0}$	ReLU		MaxMin	
	estimate	time (s)	estimate	time (s)
Layerwise	5.86	0.000	6.10	0.002
LipSDP	1.19	4.630	4.70	0.012
SymProp	4.78	0.016	25.3	0.040
ExLipBaB	2.62	(*)	4.23	377.3

Table 4: Global Lipschitz constant estimates for NNs trained on the Wine Quality dataset w.r.t. $||.||_2$. (*) The ExLipBaB algorithm did not converge after 12 hours.

Bike sharing, Network shape [13, 20, 16, 8, 3]				
$\mathbb{R}^{d_0}$	ReLU		MaxMin	
	estimate	time (s)	estimate	time (s)
Layerwise	30848	0.000	13145	0.000
LipSDP	5895(*)	5.041	9946	0.018
SymProp	20846	0.043	58484	0.038
ExLipBaB	10802	39283	-	(**)

Table 5: Global Lipschitz constant estimates for NNs trained on the Bike sharing data w.r.t. $||.||_2$. (*) The LipSDP algorithm in the "neuron" and "layer" variants did not converge; the network variant gave a bound below the true Lipschitz constant computed by ExLipBaB. (**) The algorithm had not converged after 24 hours.

(A) $L(f)$, $f = [13, h_1, h_2, h_3, 1]$ (ReLU) on $[0, 0.1]^{13}$				
h_j	20, 16, 8	32, 32, 32	48, 48, 48	64, 64, 64
time	7.42s	48.1s	2321s	6177s
$L(f)$	3440	4605	47.84	740.3
(B) $L(f)$, $f = [13, 20, 16, 8, 3]$ (GroupSort) on $[-i, i]^{13}$				
i	0.1	0.2	0.4	1
time	5.1s	40.4s	136s	365s
$L(f)$	3.32	3.61	4.12	4.23

Table 6: ExLipBaB computation time for NNs trained on the Bike Sharing data for (A) increasing size of hidden layers $h_j, j = 1, 2, 3$ and (B) increasing input region size. For comparison for (A): computing $L(f)$ for $f = [13, 20, 16, 8, 1]$ on $\mathbb{R}^{13}$ took 29993 seconds with $L(f) = 6320$. Comparative values for full computation for (B) can be found in Table 5.

Acknowledgements

This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 459360854 as part of the Research Unit "Lifespan AI: From Longitudinal Data to Lifespan Inference in Health" (DFG FOR 5347), University of Bremen (https://lifespanai.de/).

We also thank the U Bremen Research Alliance (UBRA) for fostering our research into this topic in their AI Toolbox Hackathon.

References

[Anil *et al.*, 2019] Cem Anil, James Lucas, and Roger Grosse. Sorting Out Lipschitz Function Approximation. In *Proceedings of the 36th International Conference on Machine Learning*, pages 291–301. PMLR, May 2019.

[Behrmann *et al.*, 2019] Jens Behrmann, Will Grathwohl, Ricky T. Q. Chen, David Duvenaud, and Joern-Henrik Jacobsen. Invertible Residual Networks. In *Proceedings of the 36th International Conference on Machine Learning*, pages 573–582. PMLR, May 2019.

[Bhowmick *et al.*, 2021] Aritra Bhowmick, Meenakshi D'Souza, and G. Srinivasa Raghavan. LipBaB: Computing Exact Lipschitz Constant of ReLU Networks. In Igor Farkaš, Paolo Masulli, Sebastian Otte, and Stefan Wermter, editors, *Artificial Neural Networks and Machine Learning – ICANN 2021*, pages 151–162, Cham, 2021. Springer International Publishing.

[Chen *et al.*, 2019] Ricky T. Q. Chen, Jens Behrmann, David K Duvenaud, and Joern-Henrik Jacobsen. Residual Flows for Invertible Generative Modeling. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[Cisse *et al.*, 2017] Moustapha Cisse, Piotr Bojanowski, Edouard Grave, Yann Dauphin, and Nicolas Usunier. Parseval Networks: Improving Robustness to Adversarial Examples. In *Proceedings of the 34th International Conference on Machine Learning*, pages 854–863. PMLR, July 2017.

[Cortez *et al.*, 2009] Paulo Cortez, António Cerdeira, Fernando Almeida, Telmo Matos, and José Reis. Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, 47(4):547–553, November 2009.

[Diep, 2012] Nguyen Hong Diep. Efficient Implementation of Interval Matrix Multiplication. In Kristján Jónasson, editor, *Applied Parallel and Scientific Computing*, pages 179–188, Berlin, Heidelberg, 2012. Springer.

[Ducotterd *et al.*, 2024] Stanislas Ducotterd, Alexis Goujon, Pakshal Bohra, Dimitris Perdios, Sebastian Neumayer, and Michael Unser. Improving Lipschitz-Constrained Neural Networks by Learning Activation Functions. *Journal of Machine Learning Research*, 25(65):1–30, 2024.

[Fanaee-T and Gama, 2014] Hadi Fanaee-T and Joao Gama. Event labeling combining ensemble detectors and background knowledge. *Progress in Artificial Intelligence*, 2(2):113–127, June 2014.

[Fazlyab *et al.*, 2019] Mahyar Fazlyab, Alexander Robey, Hamed Hassani, Manfred Morari, and George Pappas. Efficient and Accurate Estimation of Lipschitz Constants for Deep Neural Networks. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[Gehr *et al.*, 2018] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai 2: Safety and robustness certification of neural networks with abstract interpretation. In *Security and Privacy (SP), 2018 IEEE Symposium on*, 2018.

[Gendron and Crainic, 1994] Bernard Gendron and Teodor Gabriel Crainic. Parallel Branch-And-Bound Algorithms: Survey and Synthesis. *Operations Research*, 42(6):1042–1066, 1994.

[Gorokhovik *et al.*, 1994] V. V. Gorokhovik, O. I. Zorko, and G. Birkhoff. Piecewise affine functions and polyhedral sets. *Optimization*, 31(3):209–221, January 1994.

[Goujon *et al.*, 2024] Alexis Goujon, Arian Etemadi, and Michael Unser. On the number of regions of piecewise linear neural networks. *Journal of Computational and Applied Mathematics*, 441:115667, May 2024.

[Gouk *et al.*, 2021] Henry Gouk, Eibe Frank, Bernhard Pfahringer, and Michael J. Cree. Regularisation of neural networks by enforcing Lipschitz continuity. *Machine Learning*, 110(2):393–416, February 2021.

[Hanin and Rolnick, 2019] Boris Hanin and David Rolnick. Deep ReLU Networks Have Surprisingly Few Activation Patterns. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[Huster *et al.*, 2019] Todd Huster, Cho-Yu Jason Chiang, and Ritu Chadha. Limitations of the Lipschitz Constant as a Defense Against Adversarial Examples. In Carlos Alzate, Anna Monreale, Haytham Assem, Albert Bifet, Teodora Sandra Buda, Bora Caglayan, Brett Drury, Eva García-Martín, Ricard Gavaldà, Irena Koprinska, Stefan Kramer, Niklas Lavesson, Michael Madden, Ian Molloy, Maria-Irina Nicolae, and Mathieu Sinn, editors, *ECML PKDD 2018 Workshops*, pages 16–29, Cham, 2019. Springer International Publishing.

[Johnston, 2016] Nathaniel Johnston. How to compute hard-to-compute matrix norms. https://njohnston.ca/2016/01/how-to-compute-hard-to-compute-matrix-norms/, 2016. Accessed: 2024-05-22.

[Jordan and Dimakis, 2020] Matt Jordan and Alexandros G Dimakis. Exactly Computing the Local Lipschitz Constant of ReLU Networks. In *Advances in Neural Information Processing Systems*, volume 33, pages 7344–7353. Curran Associates, Inc., 2020.

[Lee, 2004] Jon Lee. *A First Course in Combinatorial Optimization*. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, 2004.

[Li *et al.*, 2019] Jianlin Li, Jiangchao Liu, Pengfei Yang, Liqian Chen, Xiaowei Huang, and Lijun Zhang. Analyzing Deep Neural Networks with Symbolic Propagation:

Towards Higher Precision and Faster Verification. In Bor-Yuh Evan Chang, editor, *Static Analysis*, pages 296–319, Cham, 2019. Springer International Publishing.

[Pauli *et al.*, 2023] Patricia Pauli, Dennis Gramlich, and Frank Allgöwer. Lipschitz constant estimation for 1D convolutional neural networks. In *Proceedings of The 5th Annual Learning for Dynamics and Control Conference*, pages 1321–1332. PMLR, June 2023.

[Pauli *et al.*, 2024] Patricia Pauli, Aaron J. Havens, Alexandre Araujo, Siddharth Garg, Farshad Khorrami, Frank Allgöwer, and Bin Hu. Novel Quadratic Constraints for Extending LipSDP beyond Slope-Restricted Activations. In *The Twelfth International Conference on Learning Representations*, 2024.

[Pintore and Després, 2024] Moreno Pintore and Bruno Després. Computable Lipschitz Bounds for Deep Neural Networks, October 2024. arXiv:2410.21053 [cs].

[Rump, 2012] Siegfried M. Rump. Fast interval matrix multiplication. *Numerical Algorithms*, 61(1):1–34, September 2012.

[Shi *et al.*, 2022] Zhouxing Shi, Yihan Wang, Huan Zhang, J. Zico Kolter, and Cho-Jui Hsieh. Efficiently Computing Local Lipschitz Constants of Neural Networks via Bound Propagation. *Advances in Neural Information Processing Systems*, 35:2350–2364, December 2022.

[Stargalla *et al.*, 2025] Moritz Stargalla, Christoph Hertrich, and Daniel Reichman. The Computational Complexity of Counting Linear Regions in ReLU Neural Networks. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen, editors, *Advances in Neural Information Processing Systems*, volume 38, pages 125970–125988. Curran Associates, Inc., 2025.

[Szegedy *et al.*, 2014] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks, February 2014. arXiv:1312.6199 [cs].

[Tanielian and Biau, 2021] Ugo Tanielian and Gerard Biau. Approximating Lipschitz continuous functions with GroupSort neural networks. In *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*, pages 442–450. PMLR, March 2021.

[Tao *et al.*, 2022] Qinghua Tao, Li Li, Xiaolin Huang, Xiangming Xi, Shuning Wang, and Johan A. K. Suykens. Piecewise linear neural networks and deep learning. *Nature Reviews Methods Primers*, 2(1):42, June 2022.

[Tsuzuku *et al.*, 2018] Yusuke Tsuzuku, Issei Sato, and Masashi Sugiyama. Lipschitz-Margin Training: Scalable Certification of Perturbation Invariance for Deep Neural Networks. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[Virmaux and Scaman, 2018] Aladin Virmaux and Kevin Scaman. Lipschitz regularity of deep neural networks: analysis and efficient estimation. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[Wang, 2022] Yuan Wang. Estimation and Comparison of Linear Regions for ReLU Networks. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence*, volume 4, pages 3544–3550, July 2022.

[Warwick *et al.*, 1994] Nash Warwick, Sellers Tracy, Talbot Simon, Cawthorn Andrew, and Ford Wes. Abalone, 1994. Published: UCI Machine Learning Repository.

[Weng *et al.*, 2018a] Lily Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane Boning, and Inderjit Dhillon. Towards Fast Computation of Certified Robustness for ReLU Networks. In *Proceedings of the 35th International Conference on Machine Learning*, pages 5276–5285. PMLR, July 2018.

[Weng *et al.*, 2018b] Tsui-Wei Weng, Huan Zhang, Pin-Yu Chen, Jinfeng Yi, Dong Su, Yupeng Gao, Cho-Jui Hsieh, and Luca Daniel. Evaluating the Robustness of Neural Networks: An Extreme Value Theory Approach. In *International Conference on Learning Representations*, 2018.

[Zhang *et al.*, 2024] Zeyu Zhang, Khandaker Asif Ahmed, Md Rakibul Hasan, Tom Gedeon, and Md Zakir Hossain. A Deep Learning Approach to Diabetes Diagnosis. In Ngoc Thanh Nguyen, Richard Chbeir, Yannis Manolopoulos, Hamido Fujita, Tzung-Pei Hong, Le Minh Nguyen, and Krystian Wojtkiewicz, editors, *Recent Challenges in Intelligent Information and Database Systems*, pages 87–99, Singapore, 2024. Springer Nature.