Convex Hull Approximation for Activation Functions

ZHONGKUI MA[*], The University of Queensland, Australia
ZIHAN WANG[*], The University of Queensland and CSIRO's Data61, Australia
GUANGDONG BAI[†], The University of Queensland, Australia

The wide adoption of deep learning in safety-critical domains has driven the need for formally verifying the robustness of neural networks. A critical challenge in this endeavor lies in addressing the inherent non-linearity of activation functions. The convex hull of the activation function has emerged as a promising solution, as it effectively tightens variable ranges and provides multi-neuron constraints, which together enhance verification precision. Given that constructing exact convex hulls is computationally expensive and even infeasible in most cases, existing research has focused on over-approximating them. Several ad-hoc methods have been devised for specific functions such as ReLU and Sigmoid. Nonetheless, there remains a substantial gap in developing broadly applicable approaches for general activation functions.

In this work, we propose WraAct, an approach to efficiently constructing tight over-approximations for activation function hulls. Its core idea is to introduce linear constraints to smooth out the fluctuations in the target function, by leveraging *double-linear-piece* (DLP) functions to simplify the local geometry. In this way, the problem is reduced to over-approximating DLP functions, which can be efficiently handled. We evaluate WraAct against SBLM+PDDM, the state-of-the-art (SOTA) multi-neuron over-approximation method based on decomposing functions into segments. WraAct outperforms it on commonly-used functions like Sigmoid, Tanh, and MaxPool, offering superior efficiency (average 400X faster) and precision (average 150X) while constructing fewer constraints (average 50% reduction). It can complete the computation of up to 8 input dimensions in 10 seconds. We also integrate WraAct into a neural network verifier to evaluate its capability in verification tasks. On 100 benchmark samples, it significantly enhances the single-neuron verification from under 10 to over 40, and outperforms the multi-neuron verifier PRIMA with up to additional 20 verified samples. On large networks like ResNets with 22k neurons, it can complete the verification of one sample within one minute.

CCS Concepts: • **Security and privacy → Logic and verification**; • **Computing methodologies → Neural networks**.

Additional Key Words and Phrases: Robustness, Neural Networks, Convexity, Polytope

ACM Reference Format:
Zhongkui Ma, Zihan Wang, and Guangdong Bai. 2025. Convex Hull Approximation for Activation Functions. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 308 (October 2025), 33 pages. https://doi.org/10.1145/3763086

1 Introduction

Deep neural networks (DNNs) have achieved remarkable success in a range of fields, such as image recognition [Krizhevsky et al. 2012], game playing [Silver et al. 2016], and natural language processing [Hinton et al. 2012; Sutskever et al. 2014]. These models present an appealing alternative

[*]Equal contribution.
[†]Corresponding author.

Authors' Contact Information: Zhongkui Ma, zhongkui.ma@uq.edu.au, The University of Queensland, Brisbane, Queensland, Australia; Zihan Wang, zihan.wang@uq.edu.au, The University of Queensland and CSIRO's Data61, Brisbane, Queensland, Australia; Guangdong Bai, baiguangdong@gmail.com, The University of Queensland, Brisbane, Queensland, Australia.

ACM 2475-1421/2025/10-ART308
https://doi.org/10.1145/3763086

to handcrafted software, offering ease of development alongside excellent performance [Wang et al. 2025, 2024]. However, despite their impressive capabilities, their application in safety-critical contexts has been limited. This is mainly due to their "black box" nature that makes the decision-making process opaque and raises concerns of unexpected behaviors [Feng et al. 2024]. Indeed, DNNs can be misled by imperceptible perturbations into producing wrong classifications [Cao et al. 2021; Carlini and Wagner 2017; Ma et al. 2023], or even generating harmful outputs [Yang et al. 2024; Yu et al. 2024].

As neural network components in safety-critical systems are set to expand dramatically, verifying their robustness is becoming increasingly indispensable. Some advancements have been achieved with approaches such as bound propagation [Singh et al. 2019b; Weng et al. 2018; Zhang et al. 2018] and linear programming [Ma et al. 2024; Müller et al. 2022; Singh et al. 2019a]. They all involve linear constraints to over-approximate non-linear functions, and two types of constraints have been widely used [Li et al. 2023; Meng et al. 2022], *i.e.*, *single-neuron constraints*[Ehlers 2017; Katz et al. 2019; Singh et al. 2019b; Weng et al. 2018; Zhang et al. 2018], which involve a single activation function, and *multi-neuron constraints* [Ma 2023; Ma et al. 2024; Müller et al. 2022; Salman et al. 2019; Singh et al. 2019a; Tjandraatmadja et al. 2020], which take into account the correlation among multiple activation functions. Constructing single-neuron constraints is straightforward in the space of a single input and a single output, but it often leads to lower precision in the verification due to the absence of constrained correlation among multiple variables. Multi-neuron constraints enhance the verification precision by capturing non-trivial correlations among multiple variables. Nonetheless, constructing multi-neuron constraints is costly due to the complex architecture of networks, particularly in high-dimensional space. State-of-the-art approaches [Ma et al. 2024; Müller et al. 2022; Singh et al. 2019a] typically construct linear constraints for each layer, which involve its inputs and outputs. Since each layer takes the outputs from the preceding layer as inputs, these constraints cover neurons in adjacent layers.

Several studies have been conducted on the effective construction of multi-neuron constraints [Ma et al. 2024; Müller et al. 2022; Singh et al. 2019a; Tjandraatmadja et al. 2020], by analyzing neurons within each layer. Since processing one layer with many neurons is still expensive, the neuron grouping strategy, which partitions neurons within the same layer, is further proposed [Singh et al. 2019a] to avoid the cost of handling high-dimensional multi-neuron cases. These approaches have proven effective for piecewise linear activation functions, such as the ReLU function, given that their piecewise linearity facilitates reducing the number of required linear constraints to enhance verification efficiency. In practice, however, neural networks often involve a broader variety of non-linear activation functions, which can be roughly categorized into S-shaped functions (typically Sigmoid and Tanh) and ReLU-like functions (typically ELU [Clevert et al. 2016] and leaky ReLU [Maas et al. 2013]). These functions lack simple geometry properties like piecewise linearity, making over-approximating their convex hulls challenging. This challenge has hindered the development of *generally applicable approaches to constructing tight hull over-approximations of activation functions*, and thus limits the broad applicability of neural network verification in practice.

Our work. In this work, we explore a *generally applicable* over-approximation of the convex hulls of activation functions, referred to as *function hull*s. We introduce an approach named WraAct (wrapping activation functions). Given an activation function σ and a bounded convex polytope $\mathcal{X}$ as the input domain, it constructs an over-approximation of $\mathrm{Conv}((\mathcal{X}, \mathcal{Y})) = \mathrm{Conv}((\mathcal{X}, \sigma(\mathcal{X})))$, where $(\mathcal{X}, \mathcal{Y})$ represents the the set of input-output pairs of σ. The core idea of WraAct is to leverage a *divide-and-conquer* strategy to handle the target function from local to global. It starts by segmenting the target function into local pieces such that each exhibits minimal fluctuation. For each pair of adjacent local pieces, it constructs a *double-linear-piece* (DLP) function to closely approximate each piece while capturing the changes between them. These DLP functions are then over-approximated using an efficient algorithm to obtain constraints of their upper and lower bounds. The constraints are expanded and integrated across the global input domain, yielding an over-approximation of the target function over its entire domain.

To illustrate WraAct's solution, consider an S-shaped function $\mathcal{Y} = \sigma(\mathcal{X})$. WraAct segments $\mathcal{Y}$ into three pieces. The first piece approaches its lower asymptote, the second piece depicts its main increase around the origin, and the third approaches its upper asymptote. To address the non-linearity of these three pieces as a whole, one DLP function, which is either locally convex or concave, is constructed to approach a pair of adjacent pieces. For the first two pieces, WraAct constructs a convex DLP function as its upper bound, *i.e.*, $\overline{\sigma}(\mathcal{X}) \geq \sigma(\mathcal{X})$, and for the last two pieces, a concave DLP function is as its lower bound, *i.e.*, $\underline{\sigma}(\mathcal{X}) \leq \sigma(\mathcal{X})$. These DLP functions are used to derive constraints that over-approximate the target function.

The geometry properties of DLP functions greatly contribute to WraAct's precision and efficiency. For precision, the DLP functions can well depict the local geometry of the target function based on its local pieces. Introducing them enhances the representation of complex nonlinear behaviors using linear constraints and thus facilitates the construction of precise multi-neuron constraints. In terms of efficiency, the DLP functions' feature of two linear pieces (similar to the ReLU hull [Ma et al. 2024]) facilitates an efficient construction of their over-approximation. When expanding to the global input domain, the constraints from the over-approximation of the DLP functions naturally become constraints of the target function if the constraints do not cross the target function. Such constraints can be efficiently identified using coefficient signs of the constraints.

We conduct a comprehensive evaluation of WraAct, with both intrinsic and extrinsic analyses. Our evaluation covers almost all extensively-used activation functions, including Sigmoid, Tanh, leaky ReLU [Maas et al. 2013], ELU [Hendrycks and Gimpel 2023], and MaxPool. The intrinsic evaluation focuses on four metrics that have been commonly used for function hull over-approximation, *i.e.*, precision, efficiency, constraint complexity, and scalability [Ma et al. 2024; Müller et al. 2022]. It is applied to Sigmoid, Tanh, and MaxPool, given their complexity and representativeness in terms of geometry. We compare WraAct with SBLM+PDDM [Müller et al. 2022], the SOTA multi-neuron over-approximation method. With input dimensions from 2 to 4, WraAct consistently achieves an average 150X tighter over-approximation than SBLM+PDDM and an average 400X faster computation than SBLM+PDDM. It constructs succinct over-approximations with fewer constraints, which is an average of 50% by SBLM+PDDM. WraAct remains salable with up to 8 input dimensions, in which the computation is completed within 10 seconds.

The extrinsic evaluation focuses on the performance of WraAct in local robustness verification. To this end, we implement a verifier integrating WraAct for generating multi-neuron constraints, which provides non-trivial constraints that take into consideration correlated variables among multiple activation functions. We evaluate the verifier on two fully-connected networks and two convolutional networks on the MNIST and CIFAR10 datasets. It is benchmarked in comparison with two single-neuron verifiers, DeepPoly [Singh et al. 2019b] and CROWN [Zhang et al. 2018], and the SOTA multi-neuron verifier PRIMA [Müller et al. 2022], in terms of the number of successfully verified samples out of 100 benchmark samples, and the runtime of verification. The results demonstrate that WraAct significantly outperforms single-neuron verifiers with an average of 30 and up to 40 samples and PRIMA with an average of an extra 20 samples and a speed of 4X–10X faster. Notably, the WraAct-based verifier takes a further step in verifying real-world neural networks. In residual neural networks with over 22k neurons, it can verify a sample within one minute, showing its practicality and scalability.

Contributions. The main contributions of this work are summarized as follows.

- **A valuable step forward in the generally applicable framework of over-approximating the convex hull of activation functions.** We propose an approach to over-approximating function hulls of non-linear activation functions with high-dimensional dependent variables by taking advantage of the tractable properties of DLP functions. This approach represents a significant advancement towards establishing function hull over-approximations of general activation functions, the major open problem in neural network verification.
- **A solid theoretical foundation on the function hulls of activation functions.** We analyze the function hull of the DLP function by proving the equivalence of the topological properties between the DLP and ReLU functions from the view of linear transformation and demonstrate the soundness of our approach. With strategies designed for ReLU-like and S-shaped functions, WraAct can handle widely-used activation functions, including Sigmoid, Tanh, ELU, leaky ReLU, and MaxPool.
- **An evaluation with a range of activation functions and network architectures.** We integrate WraAct into a neural network verifier and test it on various activation functions, including Sigmoid, Tanh, leaky ReLU, ELU, and MaxPool, and a broad range of network architectures, *e.g.*, fully-connected, convolutional, and residual networks. The results show that WraAct-based verifier is significantly more precise and efficient than SOTA verifiers and scales to large residual networks.

Notation. This paper adheres to the following notation conventions: normal letters are for representing individual scalar variables such as a, b, c, x and y; bold letters for column vectors like $\boldsymbol{x}$ and $\boldsymbol{b}$, where $\boldsymbol{b}_{:n}$ denotes the subvector consisting of the first n entries, and b_n denotes the n-th scalar entry of $\boldsymbol{b}$; bold capital letters for matrices like $\boldsymbol{W}$ and $\boldsymbol{A}$, where $\boldsymbol{A}_i$ is the i-th row vector and A_{ij} is the j-th entry in the i-th row; $\boldsymbol{I}_n$ is an n-dimensional identity matrix; $\mathbf{0}_{m\times n}$ is an $m \times n$ all-zero matrix; $\mathbf{0}_n$ is an n-dimensional all-zero column vector; $\mathbf{1}_{m\times n}$ is an $m \times n$ all-one matrix; $\mathbf{1}_n$ is an n-dimensional all-one column vector; calligraphic capital letters for point sets like $\mathcal{M}$, $\mathcal{X}$, $\mathcal{Y}$; script capital letter $\mathscr{L}$ for linear transformation; $(x_1, x_2, \cdots, x_n)$-space denotes a specific space with variable $x_1, x_2, \ldots, x_n$; $m..n$ to represent $\{m \le i \le n \mid i \in \mathbb{Z}\}$ for simple notation. The operator $\mathrm{Conv}(\cdot)$ is used to get the convex hull of the given point set. When a set of linear constraints represents polytopes (high-dimensional polyhedra), the intersection of polytopes is the union of their linear constraints. For instance, the intersection of $\{x_1 \ge 0\}$ and $\{x_2 \ge 0\}$ is $\{x_1 \ge 0, x_2 \ge 0\}$. f' represents the derivative function of a function f.

2 Preliminaries

This section presents the formal definitions relevant to neural network verification using linear constraints. While we focus on fully connected networks for brevity, convolutional networks are similar due to their mathematical equivalence in terms of linear transformation.

2.1 Neural Networks with Linear Constraints

2.1.1 Neural Network Verification. Neural network verification seeks to confirm that a neural network satisfies a specified output property under given input conditions, represented by linear constraints. *Local robustness* verification is a common and critical property that needs to be guaranteed in many scenarios, especially in classification tasks, which is the focus of this work. Formally, for a neural network $\boldsymbol{y} = f(\boldsymbol{x})$, given an input $\boldsymbol{x}$ with a perturbation radius $\epsilon \in \mathbb{R}$, the verification aims to ensure that $\forall \boldsymbol{x}' \in \{\boldsymbol{x}' \mid \|\boldsymbol{x}' - \boldsymbol{x}\|_p \le \epsilon\}$, $f(\boldsymbol{x}') = f(\boldsymbol{x})$, where $p = \infty$ (the ℓ_∞ norm) is a common choice because it allows each input dimension to be perturbed independently, providing a sufficiently strong condition.

Soundness and completeness. Verification methodologies can be categorized into two main types, *exact* and *approximate*. Exact approaches determine the precise output range using satisfiability modulo theories (SMT) or mixed integer linear programming (MILP). However, these methods are inefficient when applied to large-scale neural networks. In contrast, approximate approaches determine variable ranges using linear constraints, rendering them efficient and scalable. While exact approaches offer *complete* verification, which yields merely true positive and true negative samples and provides exactly "yes" or "no" responses to the verified property, approximate approaches deliver *sound but incomplete* verification, furnishing only partial true positive samples and responding with "yes" or "undecidable" to the verified property. Note that an approximate approach can be combined with the branch-and-bound (BaB) strategy to achieve complete verification [Bunel et al. 2020, 2018] for piecewise linear functions (such as ReLU and MaxPool) due to the finite number of linear pieces that can be verified. Nevertheless, the BaB strategy cannot branch a general activation function into a finite number of exactly verifiable linear pieces and cannot numerically bound the exact output ranges, implying that no universal complete verification exists.

2.1.2 Single-neuron and Multi-neuron Constraints.

Our discussion focuses on a hidden layer of a neural network, $\boldsymbol{y}^{(i)} = \sigma(\boldsymbol{y}^{(i-1)})$, where $\boldsymbol{y}^{(i)}$ and $\boldsymbol{y}^{(i-1)}$ are respectively the output and input vector of the i-th hidden layer and σ is the activation function. A hidden layer can be decomposed into a linear and a non-linear operation as $\boldsymbol{x}^{(i)} = \boldsymbol{W}^{(i)}\boldsymbol{y}^{(i-1)} + \boldsymbol{b}^{(i)}$ and $\boldsymbol{y}^{(i)} = \sigma(\boldsymbol{x}^{(i)})$, where $\boldsymbol{x}^{(i)}$ and $\boldsymbol{y}^{(i)}$ are *pre-* and *post-activated* variables, $\boldsymbol{W}^{(i)}$ and $\boldsymbol{b}^{(i)}$ are weight and bias[1].

Single-neuron constraints. The common approach is to construct linear constraints involving x_j and y_j in a two-dimensional (x_j, y_j)-space, where x_j is the pre-activated variable and y_j is the output of the j-th neuron, omitting the layer indexing. The linear constraints obtained in this way are called single-neuron constraints because these constraints only involve the input and output variables of one neuron (examples shown in Figure 1). These linear constraints are commonly used in bound-propagation-based approaches due to their simplicity [Singh et al. 2018, 2019b; Weng et al. 2018; Zhang et al. 2018].

Multi-neuron constraints. Multi-neuron constraints involve multiple neuron variables simultaneously. This study defines multi-neuron constraints as linear constraints involving multiple output variables y_j and pre-activation variables x_j. Specifically, a *single-neuron* constraint has the form $ax_j + by_j + c \geq 0$, where a, b, c are constants. In contrast, a *multi-neuron* constraint has the form $\boldsymbol{a}^\top\boldsymbol{x} + \boldsymbol{b}^\top\boldsymbol{y} + c \geq 0$, where $\boldsymbol{a}$ and $\boldsymbol{b}$ are constant vectors, and c is a scalar.

2.2 Convex Polytope and Convex Hull

2.2.1 Convex Polytope.

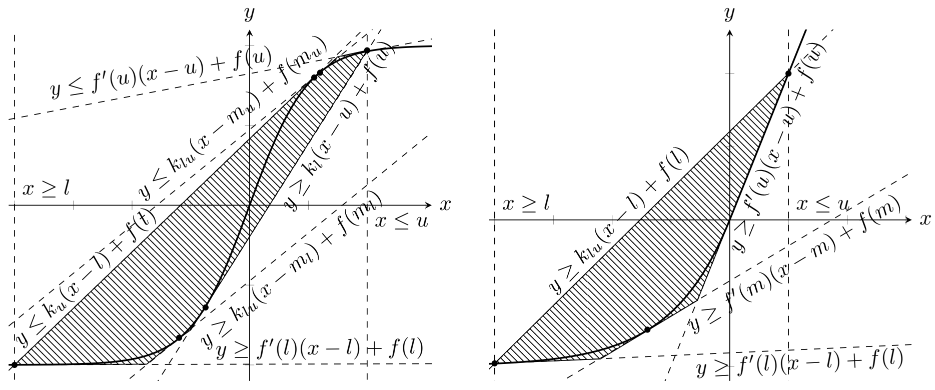
A convex polytope is, geometrically, the intersection of *halfspaces*, where a halfspace is a linear constraint that divides the space into two. The *H-representation* (halfspace representation) of a convex polytope is defined as follows.

Definition 1 (H-representation of Polytope). *A convex polytope is a set $\mathcal{X} \subset \mathbb{R}^n$ defined by a set of m halfspaces, $\mathcal{X} = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{A}_i\boldsymbol{x} + b_i \geq 0, i \in 1..m\}$, where $\boldsymbol{A}_i \in \mathbb{R}^n$ is a constant vector, and $b_i \in \mathbb{R}$ is a scalar.*

In this work, we consider the bounded convex polytope, which can be viewed as a convex set defined by finite *vertices*, also known as the V-representation (vertex representation), defined as follows.

Definition 2 (V-representation of Bounded Polytope). *A bounded convex polytope is a set $\mathcal{X} \subset \mathbb{R}^n$ defined by a set of m points, $\mathcal{V} \subset \mathbb{R}^n$ ($|\mathcal{V}| = m$), called vertices of $\mathcal{X}$, such that $\mathcal{X} = \{\sum_{i=1}^m \lambda_i\boldsymbol{v}_i \mid \sum_i^m \lambda_i = 1, \boldsymbol{v}_i \in \mathcal{V}, i \in 1..m\}$, where $\lambda_i \in \mathbb{R}$ is a non-negative constant.*

[1]A scalar function outputs a vector (or a point set) when taking a vector (or a point set) as input.



(a) $y = \tanh(x)$ ($x \in [-4, 2]$). (b) $y = \mathrm{ELU}(x)$ ($x \in [-4, 1]$).

Fig. 1. Single-neuron constraints of Tanh and ELU (more cases and details in Table 3). The hatched area is an over-approximation of the function hull determined by linear constraints, which are represented by dotted lines. The single-neuron constraints only consider the activation function shape in the 2-dimensional input-output space.

2.2.2 Convex Hull. The convex hull is the minimal convex set that contains a given set of points. Non-piecewise-linear functions, such as Sigmoid and Tanh, cannot be exactly represented within a convex polytope with flat faces because their non-linear nature necessitates a non-linear analytical representation. We define the *convex hull* as follows to maintain clarity and applicability across various contexts.

Definition 3 (Convex Hull). *The convex hull is the minimal convex set containing the given points.*

Note that this definition does not require the given points to be finite in number; therefore, the convex hull can be a general convex set.

3 Our Approach: WraAct

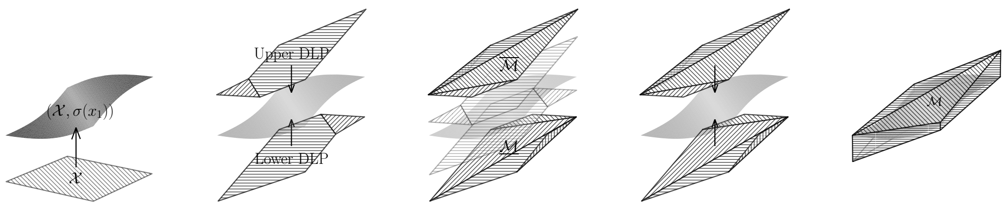
This section begins with definitions of the function hull and double-linear-piece (DLP) functions, and a taxonomy of activation functions (Section 3.1), and presents our main algorithm with an example to facilitate understanding (Section 3.2). Each step of the main algorithm regarding how WraAct handles general activation functions is presented in Section 3.3.

3.1 Definitions

3.1.1 Function Hull. This work aims to calculate a convex polytope in H-representation that over-approximates the function hull $\mathrm{Conv}((\mathcal{X}, \mathcal{Y}))$, where $\mathcal{X}$ is a given input polytope and σ is an activation function.

Definition 4 (Function Hull). *Given a bounded convex polytope $\mathcal{X} \subset (x_1, x_2, \cdots, x_n)$-space as the input domain and its image $\mathcal{Y} = \sigma(\mathcal{X}) \subset (y_1, y_2, \cdots, y_n)$-space under an activation function σ with $y_i = \sigma(x_i)$ ($i \in 1..n$), the convex hull* $\mathrm{Conv}((\mathcal{X}, \mathcal{Y})) \in \mathbb{R}^{2n}$ *is called function hull under σ.*

3.1.2 Double-linear-piece (DLP) Function. A *DLP function* is a piecewise linear function with two linear pieces, defined by two linear functions combined using the max or min function.



(a) Activating Input. (b) Constructing DLPs. (c) Approx. DLPs. (d) Identifying constr. (e) Function hull.

Fig. 2. Illustrative overview of WraAct's pipeline, which includes activating inputs (a), constructing and approximating the DLP (b-c, Step 1-2), identifying constraints (d, Step 3), and forming the function hull over-approximation (e). The multi-neuron constraints consider the space of multiple inputs and outputs, and here, three dimensions (two inputs and one output) are considered.

Definition 5 (DLP Function). *A DLP function $g : \mathbb{R}^n \to \mathbb{R}$ is a piecewise linear function with two linear pieces, defined as*

$$g(\boldsymbol{x}) = \max\{\boldsymbol{a}^T\boldsymbol{x} + c,\ \boldsymbol{b}^T\boldsymbol{x} + d\} \quad \text{or} \quad g(\boldsymbol{x}) = \min\{\boldsymbol{a}^T\boldsymbol{x} + c,\ \boldsymbol{b}^T\boldsymbol{x} + d\},$$

where $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^n$ ($\boldsymbol{a} \neq k\boldsymbol{b}$, $k \neq 0$) and $c, d \in \mathbb{R}$ are constant vectors and scalars.

Here, *max* functions are convex, and *min* functions are concave. The definition applies to both single-variable and multi-variable DLP functions. The condition $\boldsymbol{a} \neq k\boldsymbol{b}$ ensures that the two linear pieces are not parallel, avoiding degeneration to a single linear function. DLP functions are used by WraAct because they can be linearly transformed to a ReLU function (proved later in Lemmas 2 and 3), and the ReLU hull properties [Ma et al. 2024] can be *generalized to DLP functions with soundness.*

Table 1. Activation functions taxonomy.

Category	Examples	
S-shaped: - They are monotonically increasing. - They are concave on the positive axis and convex. - They asymptotically approach a constant, respectively, on the positive and negative axes.	Sigmoid	$y = 1/(1+e^{-x})$
	Tanh	$y = (e^x - e^{-x})/(e^x + e^{-x})$
	Sign	$y = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}$
	Softsign	$y = x/(1+\|x\|)$
	ReLU6 [Howard et al. 2017]	$y = \min\{\mathrm{ReLU}(x), 6\}$
ReLU-like: - They are monotonically increasing†. - They are concave on the positive axis and convex on the negative axis†. - They asymptotically approach a constant on the negative axis and a linear function on the positive axis.	Leaky ReLU [Maas et al. 2013]	$y = \max\{\alpha x, x\}$
	ELU [Clevert et al. 2016]	$y = \max\{e^x - 1, x\}$
	GELU [Hendrycks and Gimpel 2023]‡	$y = x\Phi(x)$
	Softplus [Glorot et al. 2011]	$y = \log(1+e^x)$
	SiLU [Hendrycks and Gimpel 2023]	$y = x/(1+e^{-x})$
	Hardswish [Howard et al. 2019]	$y = \begin{cases} \mathrm{ReLU}(x), & x \leq -3 \vee x \geq 3 \\ x(x+3)/6, & -3 < x < 3 \end{cases}$
Multi-variable: - They have multiple input variables.	MaxPool	$y = \max\{x_1, x_2, \cdots, x_n\}$

† GELU, SiLU, and Hardwish have almost the same characters without considering small local differences. ‡ $\Phi(x)$ is the cumulative distribution function for the Gaussian distribution.

3.1.3 Activation Function Taxonomy.

Existing commonly used activation functions are mainly single-variable functions and can be broadly categorized into two types: *S-shaped* and *ReLU-like* functions (see Table 1). S-shaped functions, characterized by their "S"-like shape, include the Sigmoid (logistic) function, Tanh (hyperbolic tangent) function, Sign function, Softsign function, and ReLU6 [Howard et al. 2017]. ReLU-like functions are characterized by their horizontally flipped L-shape, including leaky ReLU [Maas et al. 2013], ELU (exponential linear unit) [Clevert et al. 2016], GELU (Gaussian error linear unit) [Hendrycks and Gimpel 2023], Softplus function [Glorot et al. 2011], SiLU (sigmoid linear unit) [Hendrycks and Gimpel 2023], and Hardswish [Howard et al. 2019].

Multi-variable functions are rarely used as activation functions, and MaxPool is one of the notable examples. It selects the maximum value from a set of inputs and features its convexity. This work focuses on exemplary implementations of Sigmoid, Tanh, leaky ReLU, ELU, and MaxPool activation functions, demonstrating a methodology that can be generalized to other similar functions.

3.2 The Main Algorithm

3.2.1 Overview. The main algorithm of WraAct is presented in Algorithm 1. WraAct takes as input a polytope $\mathcal{X}$ with the lower bounds $\boldsymbol{l}$ and upper bounds $\boldsymbol{u}$ (shown in Figure 2a), and aims to construct a function hull over-approximation $\mathcal{M} \subset (\boldsymbol{x}, \boldsymbol{y})$-space, such that

$$\mathcal{M} \supseteq \mathrm{Conv}((\mathcal{X}, \mathcal{Y})) = \mathrm{Conv}((\mathcal{X}, \sigma(\mathcal{X}))).$$

The input (*i.e.*, $\mathcal{X}$) and output (*i.e.*, $\mathcal{M}$) are both in H-representation to provide linear constraints to facilitate the verification. Note that the algorithm is valid for *any ordering of output dimensions* and supports the computation of *partial output dimensions*. WraAct processes the output dimension iteratively, so the ordering $\boldsymbol{o}$ of the output dimensions (decreasingly ordered by input ranges) is provided to the algorithm as an input argument.

Algorithm 1: WraAct ($\mathcal{X}$, $\boldsymbol{l}$, $\boldsymbol{u}$, $\boldsymbol{o}$)

```
Input  : $\mathcal{X}$: Input polytope; $\boldsymbol{l}$: Lower bounds of input variables; $\boldsymbol{u}$: Upper bounds of input variables; $\boldsymbol{o}$:
         Indices of output dimensions.
Output : $\mathcal{M}$: Function hull over-approximation
$\underline{\mathcal{M}} \leftarrow \mathcal{X}$, $\overline{\mathcal{M}} \leftarrow \mathcal{X}$.copy();
$\underline{\mathcal{V}} \leftarrow$ GetVertices($\mathcal{X}$);
$\overline{\mathcal{V}} \leftarrow \underline{\mathcal{V}}$.copy();
while $i \leftarrow$ GetNextOutputDimension($\boldsymbol{o}$) do
    // Step 1: Segmenting the function and constructing DLP functions as bounds.
    $\underline{\sigma}, \overline{\sigma} \leftarrow$ BoundFunction($i$, $l_i$, $u_i$);
    // Step 2: Over-approximating convex hulls of DLP functions.
    $\underline{\mathcal{M}}$, $\underline{\mathcal{V}} \leftarrow$ DLPHull($\underline{\mathcal{M}}$, $\underline{\mathcal{V}}$, $\underline{\sigma}$, $i$);
    $\overline{\mathcal{M}}$, $\overline{\mathcal{V}} \leftarrow$ DLPHull($\overline{\mathcal{M}}$, $\overline{\mathcal{V}}$, $\overline{\sigma}$, $i$);
// Step 3: Identifying constraints from DLP function hull approximations.
$\mathcal{M}_m \leftarrow \emptyset$;
foreach constraint $\boldsymbol{a}^T\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{y} + c \geq 0$ in $\underline{\mathcal{M}}$ do
    if all entries in $\boldsymbol{b}$ are positive then $\mathcal{M}_m$.add($\boldsymbol{a}^T\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{y} + c \geq 0$);
foreach constraint $\boldsymbol{a}^T\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{y} + c \geq 0$ in $\overline{\mathcal{M}}$ do
    if all entries in $\boldsymbol{b}$ are negative then $\mathcal{M}_m$.add($\boldsymbol{a}^T\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{y} + c \geq 0$);
// Step 4: Complementing with single-neuron constraints.
$\mathcal{M}_s \leftarrow$ SingleNeuronConstraints($\boldsymbol{l}, \boldsymbol{u}$);
$\mathcal{M} \leftarrow \mathcal{M}_m \cap \mathcal{M}_s$;
return $\mathcal{M}$
```

Algorithm 1 consists of four main steps, *i.e.*, 1) constructing DLP functions, 2) over-approximating DLP functions, 3) identifying linear constraints from DLP Over-approximation, and 4) complementing with single-neuron constraints. The key innovation lies in the DLP function construction when processing various activation functions and the construction of over-approximation of the activation function based on the over-approximation of constructed DLP functions. For the former, WraAct adopts different strategies for S-shaped and ReLU-like functions (Step 1), which are discussed in Section 3.3. For the latter, WraAct first obtains the DLP function over-approximation with a precise and efficient method inspired by WraLU [Ma et al. 2024] (Step 2). As DLP functions are constructed locally, the linear constraints from the DLP function over-approximation may not be sound for the target function (**Challenge #1**), and the linear constraints are not tight for the whole input domain of the target function (**Challenge #2**). For Challenge #1, WraAct identifies constraints sound for the target function (Step 3), and for Challenge #2, it uses single-neuron constraints as complements to achieve a tight over-approximation for the target function.

3.2.2 Algorithm Steps with a Running Example. Below, we brief each step of Algorithm 1. To facilitate the understanding, we present the calculation of a convex hull of the Tanh function σ in the (x_1, x_2, y_1, y_2)-space as a running example (detailed numerical results are provided in Appendix A.1). Step 1 (Figure 2b) involves constructing DLP functions, $\underline{\sigma}_i$ and $\overline{\sigma}_i$, as lower and upper bounds for the i-th coordinate. Step 2 (Figure 2c) calculates the convex over-approximations $\underline{\mathcal{M}}$ and $\overline{\mathcal{M}}$ of these DLP functions. In Step 3 (Figure 2d and Figure 2e), constraints are identified from $\underline{\mathcal{M}}$ and $\overline{\mathcal{M}}$ to form a convex hull over-approximation of the target function σ, denoted as $\mathcal{M}_m \supseteq \mathrm{Conv}((\mathcal{X}, \mathcal{Y}))$. Since the first three steps only supply constraints for the local geometry, additional linear constraints from single-neuron over-approximations $\mathcal{M}_s$ are incorporated in Step 4 to construct a tighter function hull over-approximation $\mathcal{M} = \mathcal{M}_m \cap \mathcal{M}_s$ of the target function.

Step 1: Segmenting the function and constructing DLP functions as bounds. First, we construct DLP functions to serve as the lower and upper bounds for the target function $y_i = \sigma(x_i)$ within the domain $x_i \in [-2, 2]$ for $i = 1, 2$, as shown in Figure 2b and 3a (function `Boundfunction(·)` in Algorithm 1). Based on the S-shape of the Tanh function, we segment it into three pieces. Specifically, for the locally convex segment of the target function, we formulate a convex DLP function $\overline{\sigma}_i(x_i)$ that consistently lies above the target function, and for the concave segment, a concave DLP function $\underline{\sigma}_i(x_i)$. The DLP functions $\underline{\sigma}_i(x_i)$ and $\overline{\sigma}_i(x_i)$ satisfy

$$\underline{\sigma}_1(x_1) \le \sigma(x_1) \le \overline{\sigma}_1(x_1), \quad \underline{\sigma}_2(x_2) \le \sigma(x_2) \le \overline{\sigma}_2(x_2). \tag{1}$$

Step 2: Over-approximating convex hulls of DLP functions. We then calculate the function hull approximations $\underline{\mathcal{M}}_1$ and $\overline{\mathcal{M}}_1$ of the DLP functions as follows (function `DLPHull(·)` in Algorithm 1; detailed in Algorithm 2),

$$\underline{\mathcal{M}}_1 \supseteq \mathrm{Conv}(\{(\boldsymbol{x}, \underline{\sigma}_1(x_1)) \mid \boldsymbol{x} \in \mathcal{X}\}), \quad \overline{\mathcal{M}}_1 \supseteq \mathrm{Conv}(\{(\boldsymbol{x}, \overline{\sigma}_1(x_1)) \mid \boldsymbol{x} \in \mathcal{X}\}), \tag{2}$$

using an efficient algorithm inspired by WraLU [Ma et al. 2024] (illustrated in Figure 2c and discussed in Section 4.2).

Step 1 and Step 2 are iteratively applied for y_2 after y_1, and each output dimension is considered for general cases. For the case of y_2, the convex approximations

$$\begin{aligned} \underline{\mathcal{M}} = \underline{\mathcal{M}}_2 &\supseteq \mathrm{Conv}(\{(\boldsymbol{x}, \underline{\sigma}_1(x_1), \underline{\sigma}_2(x_2)) \mid \boldsymbol{x} \in \mathcal{X}\}), \\ \overline{\mathcal{M}} = \overline{\mathcal{M}}_2 &\supseteq \mathrm{Conv}(\{(\boldsymbol{x}, \overline{\sigma}_1(x_1), \overline{\sigma}_2(x_2)) \mid \boldsymbol{x} \in \mathcal{X}\}), \end{aligned} \tag{3}$$

are constructed using $\underline{\sigma}_2(x_2)$ and $\overline{\sigma}_2(x_2)$.

Step 3: Identifying constraints from DLP function hull approximations. Intuitively, we construct two convex polytopes, $\underline{\mathcal{M}}$ and $\overline{\mathcal{M}}$, in Steps 1 and 2, which lie below and above $(\mathcal{X}, \mathcal{Y})$, respectively. Subsequently, the convex approximation to $\mathrm{Conv}((\mathcal{X}, \mathcal{Y}))$ is defined by the constraints identified from $\underline{\mathcal{M}}$ and $\overline{\mathcal{M}}$ (lines 7–10 in Algorithm 1). In this process, the constraints from $\overline{\mathcal{M}}$ that provide upper bounds for σ and the constraints from $\underline{\mathcal{M}}$ that provide lower bounds for σ (shown in Figure 2d) are selected. They together form an over-approximation in the (x_1, x_2, y_1)-space (see Figure 2e). The convex approximation incorporating multi-neuron constraints along with the input constraints from $\mathcal{X}$ is

$$\mathcal{M}_m \supseteq \mathrm{Conv}((\mathcal{X}, \mathcal{Y})). \tag{4}$$

Step 4: Complementing with single-neuron constraints. The above process only considers the upper and lower constraints for local convexity or concavity, without considering the lower and upper constraints for convexity or concavity. Therefore, we complement the multi-neuron constraints with the single-neuron over-approximation (function `SingleNeuronConstraints(·)` in Algorithm 1)

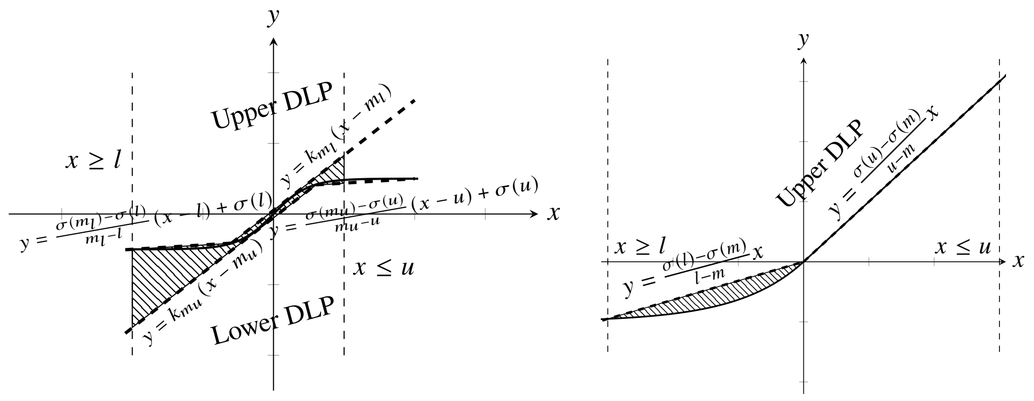
$$\mathcal{M}_s \supseteq \mathrm{Conv}((\mathcal{X}, \mathcal{Y})). \tag{5}$$

This results in a tighter function hull approximation,

$$\mathcal{M} = \mathcal{M}_m \cap \mathcal{M}_s. \tag{6}$$

3.3 Key Components of WraAct

This section details key techniques in each step when handling different activation functions. This demonstrates the application of our approach to general activation functions based on the taxonomy in Section 3.1 for constructing DLP functions (Step 1), over-approximating DLP functions (Step 2), identifying constraints from the DLP function approximations (Step 3), and constructing single-neuron constraints (Step 4).



(a) $y = \tanh(x)$ ($x \in [-4, 2]$). (b) $y = \mathrm{ELU}(x)$ ($x \in [-3, 3]$).

Fig. 3. DLP function construction of Tanh and ELU (more cases and details in Table 2). The hatched areas show the fit degree between the DLP function(s) and the target function. The lower (or upper) DLP function of the Tanh function only has a good approaching when $x \geq -0.5$ (or $x \leq 0.5$), so a further tighter approximation needs single-neuron constraints.

3.3.1 *Construction of DLP Functions.*

We first present the construction of DLP functions as bounds (Step 1) for typical functions, including Sigmoid, Tanh, leaky ReLU, and ELU functions. WraAct takes different strategies in handling different activation functions. For S-shaped functions, it segments them into three pieces, *i.e.*, the lower asymptote, the increasing piece, and the upper asymptote. As each pair of two adjacent pieces forms a convex/concave function, it constructs one convex and one concave DLP function to approach the S-shaped function. For ReLU-like functions, as they naturally have two segmented pieces that can form a convex function, WraAct constructs one convex DLP function instead. Table 2 demonstrates the constructed DLP functions for typical activation functions of each category, and an example of ELU is in Figure 3b. For multi-variable functions (*i.e.*, MaxPool, see Section 3.1.3), we introduce multi-variable inequalities to construct multi-variable DLP functions (detailed in Section 4.1.1).

Table 2. Demonstration of constructing DLP functions as bounds for various activation functions (S-shaped functions have a symmetrical case of lower DLP functions, and ReLU-like functions only have upper DLP functions). Examples are shown in Figure 3.

Case	Upper DLP Function $\overline{\sigma}$
	$\mathrm{sigmoid}(x)$, $\tanh(x)$[†]
$k_u \geq k_{lu} \wedge u > 0$	$\begin{cases} \frac{\sigma(m_l)-\sigma(l)}{m_l-l}(x-l)+\sigma(l), & x \leq m_l \\ k_{m_l}(x-m_l), & x \geq m_l \end{cases}$
$k_u \geq k_{lu} \wedge u \leq 0$	$\begin{cases} \frac{\sigma(m_l)-\sigma(l)}{m_l-l}(x-l)+\sigma(l), & x \leq m_l \\ \frac{\sigma(m_l)-\sigma(u)}{m_l-u}(x-u)+\sigma(u), & x \geq m_l \end{cases}$
other	No DLP function construction for multi-neuron constraints, but using single-neuron constraints for efficiency. Other domains (*e.g.*, $[-3,-2]$) exhibit minor fluctuations, so multi-neuron constraints offer little benefit considering their cost.
	$\mathrm{LeakyReLU}(x)$ ($\alpha = 0.01$)
$l \leq 0 \leq u$	$\begin{cases} \alpha x, & x \leq 0 \\ x, & x \geq 0 \end{cases}$
other	The leaky ReLU function degenerates as $y = x$ when $l \geq 0$ and $y = \alpha x$ when $u \leq 0$.
	$\mathrm{ELU}(x)$[§]
$l \leq 0$	$\begin{cases} \frac{\sigma(l)-\sigma(m)}{l-m}x, & x \leq m \\ \frac{\sigma(u)-\sigma(m)}{u-m}x, & x \geq m \end{cases}$
other	The ELU function degenerates as $y = x$ when $l \geq 0$.

[†] The lower DLP function has a symmetrical case with m_u because S-shaped functions are centrally symmetrical. $k_{lu} = \frac{\sigma(u)-\sigma(l)}{u-l}$. m_l and m_u are tangent points of the tangent lines such that $\sigma'(m_l) = \sigma'(m_u) = k_{lu}$ such that $m_l \leq 0 \leq m_u$. k_{m_l} is the slope of a tangent line crossing $(m_l, \sigma(m_l))$ with a positive tangent point. k_{m_u} is the slope of a tangent line crossing $(m_u, \sigma(m_u))$ with a negative tangent point. [§] $m = (u + l)/2$.

DLP functions for S-shaped functions. WraAct exploits the property of S-shaped functions that two tangent lines crossing the given point $(x', \sigma(x'))$ excluding $(0, \sigma(0))$. One tangent line is $y = \sigma'(x')(x-x')+\sigma(x')$ taking $(x', \sigma(x'))$ as tangent point and another one is $y = \sigma'(m)(x-x')+\sigma(x')$ taking $(m, \sigma(m))$ as tangent point, where $l < m < u$. Various approaches have leveraged this property to develop methods for calculating tangent lines [Henriksen and Lomuscio 2020; Zhang et al. 2018]. An S-shaped function is convex when $x \leq 0$ and concave when $x \geq 0$. At most two DLP functions are needed as the upper bound for the convex segment and the lower bound for the concave segment. Details are in Table 2, and an example of Tanh is in Figure 3a.

DLP functions for ReLU-like functions. Based on the shape of ReLU-like functions, only one convex DLP function will be constructed as the upper bound of the target function. Details are in Table 2, and an example of ELU is in Figure 3b.

DLP functions for MaxPool function. One multi-variable DLP function $g(\boldsymbol{x}) = \max\{\boldsymbol{a}^T\boldsymbol{x} + c, \boldsymbol{b}^T\boldsymbol{x} + d\}$ such that $g(\boldsymbol{x}) \geq \text{MaxPool}(\boldsymbol{x})$ with the input domain $\mathcal{X}$, is constructed due to the convexity of MaxPool (detailed in Section 4.1.1).

3.3.2 Over-approximation of DLP Functions. The algorithm to over-approximate the DLP function (Algorithm 2) is inspired by a recent study of ReLU hull over-approximation named WraLU [Ma et al. 2024], given that the two linear pieces of a ReLU function resemble those of the DLP function. The algorithm process one output dimension y_i to collaborate with Step 2 in Algorithm 1. In Section 4.2, we provide a theoretical analysis to show that this resembling can establish the soundness of WraAct from that of WraLU.

For trivial cases, *i.e.*, all vertices of the input polytope $\mathcal{M}_{i-1}$ are in the same piece of the DLP function, WraAct only needs to add an equality constraint determined by the linear piece to the input polytope $\mathcal{M}_{i-1}$ (line 3). For non-trivial cases, WraAct adds two constraints determined by the two linear pieces ($\mathcal{P}_1$ and $\mathcal{P}_2$) of the DLP function (g) (line 5) by the convexity/concavity (`GetPieces` in line 3). Then, WraAct uses the vertices and edges of the two linear pieces from the DLP function to pinpoint the faces, determining constraints for the over-approximation. Specifically, WraAct select open edges, *i.e.*, those edges yield new faces, and combine each open edge and one vertex to form a face (lines 8–11) and check the faces' validity (lines 12–14). Here, if a new face $\mathcal{L}$ does not cross $\mathcal{P}_2$ or $\mathcal{P}_2$, producing two sets with non-empty interior, the new face is valid. The valid faces determine constraints based on the convexity/concavity of the DLP function (`GetConstr`$(\cdot)$ in line 14). Then, the constraints determined by valid faces are added to $\mathcal{M}_i$, completing the over-approximation for the DLP function for the i-th output dimension.

```
Algorithm 2: DLPHull ($\mathcal{M}_{i-1}$, $\mathcal{V}_{i-1}$, $g$, $i$)
Input  : $\mathcal{M}_{i-1}$: Over-approximation for the $(i-1)$-th dimension; $\mathcal{V}$: Vertices of input polytope; $g$:
         DLP function for the $i$-th dimension; $i$: the $i$-th output dimension.
Output : $\mathcal{M}_i$: Over-approximation for the $i$-th dimension
1  $\mathcal{V}_i \leftarrow$ ExtendDimension($\mathcal{V}_{i-1}, g, i$);
2  $\mathcal{M}_i \leftarrow \mathcal{M}_{i-1}$;
3  $\mathcal{P}_1, \mathcal{P}_2 \leftarrow$ GetPieces($g$);
4  $\mathcal{M}_i$, isTrivial $\leftarrow$ TrivialCases($\mathcal{M}_i, \mathcal{V}_i, \mathcal{P}_1, \mathcal{P}_2, g$);
5  if isTrivial then return $\mathcal{M}_i$;
6  $\mathcal{M}_i$.add(GetConstr([$\mathcal{P}_1, \mathcal{P}_2$], $g$));
7  $\mathcal{F} \leftarrow \emptyset$;
8  foreach constraint $\boldsymbol{p}^T\boldsymbol{x} + \boldsymbol{q}^T\boldsymbol{y} + r \geq 0$ in $\mathcal{M}_{i-1}$ do
9      $\mathcal{E} \leftarrow \boldsymbol{p}^T\boldsymbol{x} + \boldsymbol{q}^T\boldsymbol{y} + r = 0$;
10     if $\mathcal{P}_1 \cap \mathcal{E} \neq \emptyset$ then $\mathcal{F}$.add($\mathcal{E}$);
11     if $\mathcal{P}_2 \cap \mathcal{E} \neq \emptyset$ then $\mathcal{F}$.add($\mathcal{E}$);
12 foreach face $\mathcal{L} \in \mathcal{F} \times \mathcal{V}$ do
13     if $\mathcal{L}$ splits $\mathcal{P}_1$ or $\mathcal{P}_2$ into halves then continue;
14     $\mathcal{M}_i$.add(GetConstr($\mathcal{L}, g$));
15 return $\mathcal{M}_i$, $\mathcal{V}_i$
```

3.3.3 Constraints Identification from DLP Function Hull Approximations. After obtaining the over-approximation of the DLP functions (Step 2), WraAct identifies the constraints from the DLP function over-approximations for constructing the over-approximation of the target function (Step 3). To this end, it filters out unsound constraints for the target function within the global input domain. When a DLP function is an upper bound of the target function, the upper constraints of the DLP function are taken as the upper constraints of the target function. The case for the lower bound is handled similarly. Because only one output dimension is considered in one iteration of

WraAct, we can identify the lower/upper constraints for each output dimension by the signs of y_i in the linear constraints.

Table 3. Single-neuron constraints for various activation functions. Examples are shown in Figure 1.

Case ($x \in [l, u]$)	Upper Constraints	Lower Constraints
	sigmoid(x), tanh(x)[†‡]	
$\sigma'(u) \geq k_{lu} \wedge u \leq 0$	$y \leq k_{lu}(x - l) + \sigma(l)$	$y \geq \sigma'(l)(x - l) + \sigma(l)$, $y \geq \sigma'(u)(x - u) + \sigma(u)$, $y \geq k_{lu}(x - m_l) + \sigma(m_l)$
$\sigma'(u) \geq k_{lu} \wedge u > 0$	$y \leq k_{lu}(x - l) + \sigma(l)$	$y \geq \sigma'(l)(x - l) + \sigma(l)$, $y \geq k_l(x - u) + \sigma(u)$, $y \geq k_{lu}(x - m_l) + \sigma(m_l)$
$\sigma'(l) \geq k_{lu} \wedge l \geq 0$	$y \leq \sigma'(l)(x - l) + \sigma(l)$, $y \leq \sigma'(u)(x - u) + \sigma(u)$, $y \leq k_{lu}(x - m_u) + \sigma(m_u)$	$y \geq k_{lu}(x - u) + \sigma(u)$
$\sigma'(l) \geq k_{lu} \wedge l < 0$	$y \leq k_u(x - l) + \sigma(l)$, $y \leq \sigma'(u)(x - u) + \sigma(u)$, $y \leq k_{lu}(x - m_u) + \sigma(m_u)$	$y \geq k_{lu}(x - u) + \sigma(u)$
$k_{lu} > \sigma'(l) \wedge k_{lu} > \sigma'(u)$	$y \leq \sigma'(u)(x - u) + \sigma(u)$, $y \leq k_u(x - l) + \sigma(l)$, $y \leq k_{lu}(x - m_u) + \sigma(m_u)$	$y \geq \sigma'(l)(x - l) + \sigma(l)$, $y \geq k_l(x - u) + \sigma(u)$, $y \geq k_{lu}(x - m_l) + \sigma(m_l)$
	LeakyReLU(x) ($\alpha = 0.01$)	
$l \geq 0$	$y \leq x$	$y \geq x$
$u \leq 0$	$y \leq \alpha x$	$y \geq \alpha x$
$l < 0 < u$	$y \leq k_{lu}(x - l) + \sigma(l)$	$y \geq \alpha x$, $y \geq x$
	ELU(x)[†§]	
$l \geq 0$	$y \leq x$	$y \geq x$
$l < 0$	$y \leq k_{lu}(x - l) + \sigma(l)$	$y \geq \sigma'(l)(x - l) + \sigma(l)$, $y \geq \sigma'(u)(x - u) + \sigma(u)$, $y \geq \sigma'(m)(x - m) + \sigma(m)$

[†] $k_{lu} = \frac{\sigma(u)-\sigma(l)}{u-l}$. [‡] m_l and m_u are tangent points of the tangent lines such that $\sigma'(m_l) = \sigma'(m_u) = k_{lu}$ such that $m_l \leq 0 \leq m_u$. k_l (or k_u) is the slope of another tangent line across $(u, \sigma(u))$ (or $(l, \sigma(l))$) such that $k_l \neq \sigma'(u)$ (or $k_u \neq \sigma'(l)$). [§] $m = \frac{u+l}{2}$.

3.3.4 *Construction of Single-neuron Constraints.*

This section provides the technique of constructing single-neuron constraints (Step 4 in Algorithm 1) for typical Sigmoid, Tanh, leaky ReLU, and ELU functions. Table 3 demonstrates the single-neuron constraints implemented by this work for different activation functions.

Single-neuron constraints of S-shaped functions. Despite the single-neuron constraints of S-shaped functions being deeply studied [Henriksen and Lomuscio 2020; Zhang et al. 2018] for bound propagation with two linear constraints as the lower and upper bounds, we propose our specific implementation for over-approximation as tight as possible with more than two linear constraints. Several linear constraints are constructed depending on the input domains. Details are provided in Table 3.

Single-neuron constraints of ReLU-like functions. For single-neuron constraints, a similar relaxation of the ReLU function is constructed: one constraint as the upper bound and two or three constraints as the lower bound. In this work, we accept a single-neuron constraints selection similar to ReLU and select $y \leq k_{lu}(x - l) + \sigma(l)$ and $y \geq \sigma'(m)(x - u) + \sigma(m)$ as the upper and lower linear constraints for bound propagation, which only applicable for one upper and one lower linear constraints. Details are in Table 3.

Single-neuron constraints of MaxPool function. The MaxPool function $y = \text{MaxPool}(\boldsymbol{x})$ is a convex piece-wise linear and multi-variable function. The lower bound is trivial and is $y \geq x_i$ for each input x_i. This work accepts the commonly used upper constraint $y \leq u_{\max}$, where $u_{\max}$ is the largest upper bound of all input variables [Singh et al. 2019b; Zhang et al. 2018].

4 Soundness of WraAct

This section analyzes the soundness of WraAct to over-approximate the function hull of a given activation function. It aims to conclude the soundness of Algorithm 1 as Theorem 1.

Theorem 1 (Soundness of Main Algorithm). *Given an bounded polytope $\mathcal{X} \subset \mathbb{R}^n$ as input polytope and an activation function σ, the polytope $\mathcal{M}$ output by Algorithm 1 is an over-approximation to* $\text{Conv}((\mathcal{X}, \mathcal{Y})) = \text{Conv}((\mathcal{X}, \sigma(\mathcal{X})))$*, i.e.,* $\mathcal{M} \supseteq \text{Conv}((\mathcal{X}, \mathcal{Y}))$.

Proof Overview. We break down the soundness of WraAct into the merits of each of the four main steps outlined in Algorithm 1. By combining them, we can establish the proof.

(1) **DLP functions construction.** The DLP function constructed in Step 1 is a lower or upper bound of the target function (*i.e.*, Formula (1)). This is formulated by Lemma 1 in Section 4.1.
(2) **Over-approximation of DLP functions.** The over-approximation to the DLP function is sound (*i.e.*, Formulas (2) and (3)). This is formulated by Lemmas 2 and 3 in Section 4.2.
(3) **Constraints identification.** The constraints identified by Step 3 formulate a sound over-approximation for the function hull of the target function in the entire input domain (*i.e.*, Formula (4)). This is formulated in Lemmas 4 and 5 in Section 4.3.
(4) **Single-neuron constraints identification.** Step 4 provides complementary single-neuron constraints to tighten the over-approximation constructed in Step 3. The soundness of these constraints is formulated by Lemma 6 in Section 4.4.

The input bounded polytope is $\mathcal{X} \subset \mathbb{R}^n$ and the output polytope is $\mathcal{Y} \subset \mathbb{R}^n$ with the target activation function σ. Without losing generality, in our proof, we considers $y = \sigma(\boldsymbol{x})$ for unary activation function and multi-variable function $y = \text{MaxPool}(\boldsymbol{x}) = \max\{x_1, \ldots, x_n\}$ with $\boldsymbol{x} \in \mathcal{X}$. The discussion focuses on one output dimension for simplicity and can be extended to all output dimensions sequentially.

4.1 Boundedness of Constructed DLP Functions

We prove that the constructed DLP function by Step 1 is either an upper bound or a lower bound by partitioning the input domain and considering the linear pieces separately from the DLP function.

Lemma 1. *Given an activation function $\sigma : \mathbb{R}^n \to \mathbb{R}$ with an input domain $\mathcal{X} \subset \mathbb{R}^n$, WraAct constructs a DLP function $\underline{\sigma}$ or $\overline{\sigma}$ that satisfies*

$$\sigma(\boldsymbol{x}) \geq \underline{\sigma}(\boldsymbol{x}) \quad \text{or} \quad \sigma(\boldsymbol{x}) \leq \overline{\sigma}(\boldsymbol{x}),$$

where $\boldsymbol{x} \in \mathcal{X}$ is the input variable.

Proof. We partition the input domain $\mathcal{X}$ into two parts, each with only one linear piece of the DLP function $\overline{\sigma}$ or $\underline{\sigma}$. The linear function determined by each linear piece always has a higher and lower output value than the target function in the partitioned input domain.

A ReLU-like function is segmented into two pieces when constructing DLP functions. WraAct uses these three points for determining the DLP function, *i.e.*, two endpoints of the input domain and one point between two endpoints. Because of the convexity of the two pieces (for the special case of non-convex functions like SiLU, we need to specialize the middle point to make sure the two pieces do not cross the function), the line segments connecting these three points always have a greater value than the ReLU-like function. Therefore, the DLP function determined by these three points is an upper bound for the ReLU-like function within the input domain.

For the MaxPool function $y = \max\{x_1, x_2, \ldots, x_n\}$, the inequality $x_i + x_j \geq \max\{x_i, x_j\}$ ($x_i, x_j \geq 0$; here we only consider the MaxPool after ReLU functions) is used to merge input variables into a linear form. Consequently, we derive the inequality $\max\{\sum_{i\in\mathcal{S}_1} x_i, \sum_{j\in\mathcal{S}_2} x_j\} \geq \max\{x_1, x_2, \ldots, x_n\}$, where $\mathcal{S}_1$ and $\mathcal{S}_2$ form a partition of the set $\{1, 2, \ldots, n\}$. The expression $\max\{\sum_{i\in\mathcal{S}_1} x_i, \sum_{j\in\mathcal{S}_2} x_j\}$ consists of two linear pieces and defines a DLP function that serves as an upper bound for the MaxPool function (detailed in Section 4.1.1).

A S-shaped function is segmented into three pieces by WraAct. Unlike in ReLU-like functions, the two endpoints cannot be taken to determine a DLP function because the S-shaped function is not convex or concave in the whole input domain. Therefore, one tangent point is taken to replace one endpoint with another point between the endpoint and the tangent point, determining a DLP function. The first two pieces are convex, and three points determine a DLP function as the upper bound. The last two pieces have a symmetrical scenario, obtaining a lower bound. □

4.1.1 DLP Functions for MaxPool Functions. In the following, we only talk about the case when any entry in $\boldsymbol{x}$ can become the output. Note that we only consider the case of a MaxPool after ReLU functions, where all inputs of the MaxPool are non-negative. This is a common practical setting. Based on the formula of $\max\{x_i, x_j\} \leq x_i + x_j$ ($x_i, x_j \geq 0$), we partition the input variables into two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ ($\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$ and $\mathcal{S}_1 \cup \mathcal{S}_2 = 1..n$) such that

$$\max\{x_1, \ldots, x_n\} \leq \max\left\{\sum_{i\in\mathcal{S}_1} x_i, \sum_{i\in\mathcal{S}_2} x_i\right\}.$$

To partition the input variables, we order the input variables by their ranges $u_i - l_i$ and let $\mathcal{S}_1$ be a set of variables with odd indices and $\mathcal{S}_2$ be a set of the rest variables because we aim to make the upper bound of $\sum_{i\in\mathcal{S}_1} x_i$ and $\sum_{i\in\mathcal{S}_2} x_i$ as small as possible to achieve a tight approximation.

Trivial inputs of MaxPool functions.. Because of the bounded input domain, some input variables, called trivial inputs, will never be the output of a MaxPool function. The following property shows how to filter those trivial input variables by the vertices of the given input domain.

Property 1 (Trivial Input Determined by Vertices of Input Polytope for MaxPool Function). *Given the input domain $\mathcal{X} \subset (x_1, \cdots, x_n)$-space defined by a bounded convex polytope of the MaxPool function* $\mathrm{MaxPool}(\boldsymbol{x}) = \max\{x_1, \cdots, x_n\}$*, if the vertices of $\mathcal{X}$ never take the i-th coordinate x_i as the output, i.e., for any vertex $\boldsymbol{v} = (v_1, \cdots, v_n)$, v_i never be as the output of* $\mathrm{MaxPool}(\boldsymbol{x})$*, then*

$$\mathrm{MaxPool}(\boldsymbol{x}) = \mathrm{MaxPool}(x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n).$$

Proof. On one hand, if any point $\boldsymbol{p} = (p_1, \cdots, p_n) \in \mathcal{X}$ satisfies $\mathrm{MaxPool}(\boldsymbol{p}) \neq p_i$, then there is no vertex $\boldsymbol{v}$ such that $y\mathrm{MaxPool}(\boldsymbol{v}) = v_i$ because any point $\boldsymbol{p}$ includes any vertex $\boldsymbol{v}$.

On the other hand, we prove that if any vertex $\boldsymbol{v}$ satisfies $\mathrm{MaxPool}(\boldsymbol{v}) \neq v_i$, then there is no point $\boldsymbol{p}$ such that $\mathrm{MaxPool}(\boldsymbol{p}) = p_i$. We only need to consider the case without multiple equal inputs and prove it by contradiction. Suppose there is one point $\boldsymbol{p}$ such that $\mathrm{MaxPool}(\boldsymbol{p}) = p_i$, and there is another p_j as the second maximum value. We only consider the space of (x_i, x_j, y)-space. There will be two sets, $\mathcal{X} \cap \{x_i > x_j\} \neq \emptyset$ and $\mathcal{X} \cap \{x_i < x_j\} \neq \emptyset$, because there is a point $\boldsymbol{p} \in \mathcal{X} \cap \{x_i < x_i\} \neq \emptyset$ and all vertices are in $\mathcal{X} \cap \{x_i > x_j\} \neq \emptyset$. However, this is impossible because $\mathcal{X}$ is a convex polytope defined by all vertices (Definition 2) and one hyperplane cannot cut out a subset of $\mathcal{X}$, which does not contain any vertex, due to convexity. Specifically, by the V-representation of the convex polytope, any point p is a weighted sum of all vertices and such p cannot satisfy the above situation. □

Note that we can apply this property to each input coordinate to exclude all non-trivial ones. In the practical implementation, 1) we first remove these trivial variables by their scalar bounds. Specifically, given each entry x_i in $\boldsymbol{x}$ satisfy $x_i \in [l_i, u_i]$, where l_i and u_i are the lower and upper scalar bound of x_i, we can omit the x_i whose upper bound u_i is smaller than the lower bound of any other entry x_j ($j \neq i$), which is not yet the exact method. 2) After that, we calculate the vertices of the input polytope and filter out all trivial input variables by Property 1.

4.2 Soundness of DLP Over-approximation

The soundness of the function hull over-approximation of a DLP function is indirectly guaranteed by the case of a ReLU function [Ma et al. 2024]. Therefore, this reduces to proving the equivalence of topological properties between a DLP function and a ReLU function. We consider two aspects of the proof. 1) The DLP function can be linearly transformed to a ReLU function (formulated in Lemma 2). 2) The convex hull/over-approximation of a ReLU function can also be linearly transformed to that of the DLP function (formulated in Lemma 3). In the following, we use two functions g and h in $(\boldsymbol{x}, y)$-space connected by a linear transformation to demonstrate our proof.

4.2.1 Linear Transformation between DLP and ReLU. The following lemma constructs a linear transformation between DLP and ReLU based on the normal vectors of their linear pieces.

Lemma 2. *Given a DLP function $g : \mathbb{R}^n \to \mathbb{R}$, there exists a linear transformation $\mathscr{L} : \mathbb{R}^{n+1} \to \mathbb{R}^{n+1}$ such that $\mathscr{L}((\mathcal{X}, g(\mathcal{X}))) = (\mathcal{X}', h(\mathcal{X}'))$, and h is a ReLU function satisfying $h(\boldsymbol{x}') = h(x'_1) = \mathrm{ReLU}(x'_1)$, where $\boldsymbol{x} \in \mathcal{X}$ and $\boldsymbol{x}' \in \mathcal{X}'$. $\mathcal{X}$ and $\mathcal{X}' \subset \mathbb{R}^n$ are the input domains of g and h , respectively.*

Proof. We take $y = g(\boldsymbol{x}) = \max\{\boldsymbol{a}^T\boldsymbol{x} + c, \boldsymbol{b}^T\boldsymbol{x} + d\}$ to explain our proof, and the case defined by *min* is similar. All points in $(\mathcal{X}, g(\mathcal{X}))$ are either on the hyperplane $y = \boldsymbol{a}^T\boldsymbol{x} + c$ or $y = \boldsymbol{b}^T\boldsymbol{x} + d$. We construct a linear transformation $\mathscr{L}$ to make $y = \boldsymbol{a}^T\boldsymbol{x} + c$ and $y = \boldsymbol{b}^T\boldsymbol{x} + d$ transform to $y' = 0$ and $y' = x'_1$. Specifically, given a point $(\boldsymbol{x}, y)$, we aim to construct $\mathscr{L} : (\boldsymbol{x}, y) \mapsto \boldsymbol{A} \cdot ((\boldsymbol{x}, y) - \boldsymbol{o})$, where the matrix $\boldsymbol{A} \in \mathbb{R}^{(n+1)\times(n+1)}$ is for the affine transformation and the vector $\boldsymbol{o} \in \mathbb{R}^{n+1}$ is to translate the intersection of $y = \boldsymbol{a}^T\boldsymbol{x} + c$ and $y = \boldsymbol{b}^T\boldsymbol{x} + d$ to crossing the the original.

First, we aim to translate all points on $y = \boldsymbol{a}^T\boldsymbol{x} + c$ (or $y = \boldsymbol{b}^T\boldsymbol{x} + d$) to the points on $y = \boldsymbol{a}^T$ (or $y = \boldsymbol{b}^T$). We need to set $\boldsymbol{o}$ as a solution of the equation system $\begin{cases} \boldsymbol{a}^T(\boldsymbol{x}-\boldsymbol{o}_{:n})+c-(y-o_{n+1})=\boldsymbol{a}^T\boldsymbol{x}-y \\ \boldsymbol{b}^T(\boldsymbol{x}-\boldsymbol{o}_{:n})+d-(y-o_{n+1})=\boldsymbol{b}^T\boldsymbol{x}-y \end{cases} \Leftrightarrow$ $\begin{cases} \boldsymbol{a}^T\boldsymbol{o}_{:n}-o_{n+1}=c \\ \boldsymbol{b}^T\boldsymbol{o}_{:n}-o_{n+1}=d \end{cases}$. Second, we aim to transform all points on $y = \boldsymbol{a}^T\boldsymbol{x}$ (or $y = \boldsymbol{b}^T\boldsymbol{x}$) to the points on $y = 0$ (or $y = x_1$). Specifically, we set

$$\boldsymbol{V} = \left[\begin{array}{c:c:c} 1 & 1 & \boldsymbol{0}^T_{(n-1)} \\ \hdashline \boldsymbol{1}_{\{\boldsymbol{a}\neq 0\}} & \boldsymbol{1}_{\{\boldsymbol{a}\neq 0\}} & \boldsymbol{I}_{(n-1)\times(n-1)} \\ \hdashline \sum_{i=1}^n a_i & \sum_{i=1}^n b_i & \boldsymbol{0}^T_{(n-1)} \end{array}\right] \quad \text{and} \quad \boldsymbol{V}' = \left[\begin{array}{c:c:c} 1 & 1 & \boldsymbol{0}^T_{(n-1)} \\ \hdashline \boldsymbol{0}_{(n-1)} & \boldsymbol{0}_{(n-1)} & \boldsymbol{I}_{(n-1)\times(n-1)} \\ \hdashline 0 & 1 & \boldsymbol{0}^T_{(n-1)} \end{array}\right],$$

where $\boldsymbol{1}_{\{\boldsymbol{a}\neq 0\}}$ (or $\boldsymbol{1}_{\{\boldsymbol{b}\neq 0\}}$) denote the vector whose entries are 1 at positions where $\boldsymbol{a}$ (or $\boldsymbol{b}$) is nonzero and 0 elsewhere, $\boldsymbol{V}$ represents a set of basis vectors in the given space. Note that $\boldsymbol{V}$ is invertible because any two of its column vectors point in different directions, *i.e.*, all column vectors are linearly independent ($\boldsymbol{a}$ and $\boldsymbol{b}$ cannot be all-zero); so does $\boldsymbol{V}'$. $\boldsymbol{V}^{-1} \cdot (\boldsymbol{x}, y)$ provides the coordinates by taking two vectors (the first two columns of $\boldsymbol{V}$) on the planes of $y = \boldsymbol{a}^T\boldsymbol{x}$ and $y = \boldsymbol{b}^T\boldsymbol{x}$ as basis vectors and other axes remain unchanged. Next, $\boldsymbol{V}'$ represents a set of basis vectors of the target space and transforms points to the target space, where two vectors (the first two columns of $\boldsymbol{V}'$) on the planes of $y = 0$ and $y = x_1$ as basis vectors, and other axes remain unchanged. Hence, by considering $\boldsymbol{A}\boldsymbol{V} = \boldsymbol{V}'$, we set $\boldsymbol{A} = \boldsymbol{V}'\boldsymbol{V}^{-1}$. Such $\boldsymbol{A}$ transforms all points on the plane of $y = \boldsymbol{a}^T\boldsymbol{x}$ (or $y = \boldsymbol{b}^T\boldsymbol{x}$) to the points on $y = 0$ (or $y = x_1$). Because all points can be represented by the basis vectors determined by these column vectors in $\boldsymbol{V}$ or $\boldsymbol{V}'$, $\boldsymbol{A}$ induces a linear transformation between the corresponding coordinate systems, where $y = \boldsymbol{a}^T \boldsymbol{x}$ (or $y = \boldsymbol{b}^T \boldsymbol{x}$) is taken as $y = 0$ (or $y = x_1$). Therefore, we complete the proof. □

Lemma 2 connects the function hull between ReLU and DLP functions. The invertible transformation $\boldsymbol{A} = \boldsymbol{V}'\boldsymbol{V}^{-1}$ with $\boldsymbol{o}$ transforms the DLP function to a ReLU function, and $\boldsymbol{A}^{-1}$ with $\boldsymbol{o}$ transforms the ReLU hull to a DLP function hull, which connects their properties (analyzed in the next Section 4.2.2). Two examples (2-dimensional and 4-dimensional cases) are provided in Appendix A.2.

4.2.2 Over-approximation from ReLU to DLP. The following lemma analyzes the relative geometry positions between the over-approximation and the target function before and after the linear transformation.

Lemma 3. *Given $(\mathcal{X}, g(\mathcal{X}))$ and $(\mathcal{X}', h(\mathcal{X}'))$ with a linear transformation $\mathscr{L}$ in Lemma 2, if there is a convex polytope $\mathcal{M}' \in \mathbb{R}^{n+1}$ such that $\mathcal{M}' \supseteq \text{Conv}((\mathcal{X}', h(\mathcal{X}')))$, then the convex polytope $\mathcal{M} \in \mathbb{R}^n$ under the inverse linear transformation $\mathscr{L}^{-1} : \mathbb{R}^{n+1} \longrightarrow \mathbb{R}^{n+1}$ (considering the constant term as an extra dimension) satisfies $\mathscr{L}^{-1}(\mathcal{M}') = \mathcal{M} \supseteq \text{Conv}((\mathcal{X}, g(\mathcal{X})))$.*

Proof. We first define the constraints under the linear transformation. Given a linear constraint $\boldsymbol{r}'^T \boldsymbol{x} + s'y + t' \geq 0$ defined in $\mathcal{M}'$, its corresponding linear constraints $\boldsymbol{r}^T \boldsymbol{x} + sy + t \geq 0$ determining $\mathcal{M}$ is determined by all the points linearly transformed by all points of $\boldsymbol{r}'^T \boldsymbol{x} + s'y + t' = 0$. One point in $(\mathcal{X}, g(\mathcal{X}))$ is needed to determine the inequality sign ($\geq$ or $\leq$) by holding soundness of over-approximation.

We consider the constraints individually and analyze the position between their supporting planes and the function surface within the input domain, *i.e.*, we will prove that the function surface will always be on one side of the supporting plane determined by a linear constraint. Then, it is to prove the fact that two continuous surfaces, *i.e.*, $(\mathcal{X}', h(\mathcal{X}'))$ and $\boldsymbol{r}'^T \boldsymbol{x} + s'y + t' = 0$, without crossing still have no crossing after linear transformation, *i.e.*, $(\mathcal{X}, g(\mathcal{X}))$ and $\boldsymbol{r}^T \boldsymbol{x} + sy + t = 0$, even though they have tangent points. We prove this by categorizing the points into non-tangent and tangent points.

1) For any tangent point $\boldsymbol{u}'$, its two tangent hyperplanes on $(\mathcal{X}', g(\mathcal{X}'))$ and $\boldsymbol{r}'^T \boldsymbol{x} + s'y + t' = 0$ are the same one. After linear transformation, the two tangent hyperplanes are still the same one on $(\mathcal{X}, g(\mathcal{X}))$ and $\boldsymbol{r}^T \boldsymbol{x} + sy + t \geq 0$. This shows the tangent point still holds after the transformation.

2) For non-tangent points, we prove by contradiction. We assume that there are two non-tangent points $\boldsymbol{p} = (\boldsymbol{p}_x, p_y)$, $\boldsymbol{q} = (\boldsymbol{p}_x, p_y) \in (\mathcal{X}, g(\mathcal{X}))$ such that $\boldsymbol{r}^T \boldsymbol{p}_x + sp_y + t < 0 < \boldsymbol{r}^T \boldsymbol{q}_x + sq_y + t$ and any pre-transformed point satisfy the constraints $\boldsymbol{r}'^T \boldsymbol{x} + s'y + t' > 0$ (or $\boldsymbol{r}^T \boldsymbol{x} + sy + t < 0$). Because the continuity of $(\mathcal{X}, g(\mathcal{X}))$, there exists one point $\boldsymbol{u} = (\boldsymbol{u}_x, u_y) \in (\mathcal{X}, g(\mathcal{X}))$ on the line crossing $\boldsymbol{p}$ and $\boldsymbol{q}$ such that $\boldsymbol{r}^T \boldsymbol{u}_x + su_y + t = 0$ and $\boldsymbol{u}$ is not a tangent point. Then, $\boldsymbol{u}' = \mathscr{L}(\boldsymbol{u}) \in (\mathcal{X}', h(\mathcal{X}'))$ such that $\boldsymbol{r}^T \boldsymbol{u}'_x + su'_y + t = 0$ is not a tangent point. However, this contradicts the assumption of $\boldsymbol{r}'^T \boldsymbol{x} + s'y + t' > 0$ (or $\boldsymbol{r}^T \boldsymbol{x} + sy + t < 0$) hold for any pre-transformed point. □

By Lemma 2 and Lemma 3, all properties of ReLU hulls [Ma et al. 2024] have equivalent ones in the DLP functions hulls.

4.3 Soundness of Constraint Identification

The following lemmas consider the bound relation among the constraints of the DLP function, the DLP function itself, and the target function to prove the soundness of constraint identification.

4.3.1 Upper Constraints Identification.

Lemma 4 (Upper Constraints Identification). *Given a convex polytope $\overline{\mathcal{M}} \supseteq \mathrm{Conv}((\mathcal{X}, \overline{\sigma}(\mathcal{X})))$ from Algorithm 1, for any constraint $\boldsymbol{a}^T\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{y} + c \geq 0$ of $\overline{\mathcal{M}}$, if all components of $\boldsymbol{b}$ are non-positive, then any points in $(\mathcal{X}, \sigma(\mathcal{X}))$ satisfies this constraint.*

Proof. We focus on any one constraint $\boldsymbol{a}^T\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{y} + c \geq 0$ and any one output dimension i. The other constraints and output dimensions are similar.

1) When $b_i = 0$, the points in $\mathcal{Y}$ satisfy the constraint since no bound on y_i.

2) When $b_i < 0$ and all entries of $\boldsymbol{b}$ are non-positive, then any point $(\boldsymbol{x}, \overline{\boldsymbol{y}}) \in \overline{\mathcal{M}}$ satisfies $\overline{y}_i \leq (\boldsymbol{a}^T\boldsymbol{x} + \sum_{j\in 1..n \wedge j\neq i} b_j\overline{y}_j + c)/b_i$. By the construction of $\overline{\sigma}_i$ (Lemma 1), we have $\sigma(x_i) = y_i \leq \overline{\sigma}_i(x_i) = \overline{y}_i$ for any $i \in 1..n$. Consequently, $y_i \leq (\boldsymbol{a}^T\boldsymbol{x} + \sum_{j\in 1..n\wedge j\neq i} b_j y_j + c)/b_i$. Thus, the coordinate y_i satisfies the constraint. □

4.3.2 *Lower Constraints Identification.* We have the lemma below for the symmetrical case of concave DLP functions as the lower bounds of the target function.

Lemma 5 (Lower Constraints Identification). *Given a convex polytope $\underline{\mathcal{M}} \supseteq \mathrm{Conv}((\mathcal{X}, \underline{\sigma}(\mathcal{X})))$ in Algorithm 1, for any constraint $\boldsymbol{a}^T\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{y} + c \leq 0$ of $\underline{\mathcal{M}}$, if all components of $\boldsymbol{b}$ are non-negative, then any points in $(\mathcal{X}, \mathcal{Y})$ satisfy this constraint.*

Proof. The proof is symmetrical to that of Lemma 4. □

4.4 Construction of Single-neuron Constraints

The soundness of the single-neuron constraints (in Section 3.3.4) is analyzed by taking each constraint as a linear function and checking if the linear function is always less than or greater than the target function. The detailed proof of single-neuron constraints is provided in Appendix A.3.

Lemma 6. *Given an activation function $\sigma : \mathbb{R}^n \longrightarrow \mathbb{R}$ with an input domain $\mathcal{X} \subset \mathbb{R}$, the corresponding single-neuron constraints presented in Section 3.3 form a sound over-approximation.*

Proof. We prove each linear constraint is sound to the activation function σ in the input domain $\mathcal{X}$. Each linear constraint determines a linear function. If the linear function value is always less than the target function value, it determines a lower linear constraint for the target function. It is symmetrical for the upper constraints. We summarize the cases of constraint construction below and omit the detailed proof due to space limitations.

1) If the linear constraint is determined by the lower/upper bound of the output variable, it is naturally sound, *e.g.*, the upper bound of the MaxPool function is determined by the upper output bound.

2) If the linear constraint is determined by the endpoints of the input domain, the convexity/concavity of the target function guarantees the soundness, *e.g.*, the upper constraint of the ELU function.

3) If the linear constraint is determined by a tangent point, we need to divide the target function by the tangent point and analyze each part, considering the monotonicity. □

5 Evaluation

We conduct a comprehensive evaluation of WraAct, with both intrinsic and extrinsic analyses. The intrinsic evaluation focuses on its capability in function hull approximation. We compare it with the state-of-the-art multi-neuron over-approximation method SBLM+PDDM [Müller et al. 2022] on Sigmoid, Tanh, and MaxPool functions (Section 5.1). The extrinsic evaluation focuses on its performance in local robustness verification of neural networks. To this end, we implement a verifier integrating WraAct to verify various network architectures and activation functions and compare it with SOTA verifiers (Section 5.2).

5.1 Function Hull Over-approximation

5.1.1 Experiment Settings. The input data in our experiments is a bounded polytope presented in H-representation with bounds of each input variable, yielding a convex polytope in H-representation as the resulting approximation of the function hull.

Input Data Generation. Each polytope sample has 3^n randomly generated constraints with $2n$ constraints for ranges $[-6, 6]$ of every variable to yield a bounded polytope, where n is the input dimension. Specifically, for a randomly generated constraint $\boldsymbol{a}\boldsymbol{x} + b \geq 0$, each component of $\boldsymbol{a}$ follows a uniform distribution $\mathcal{U}[-1, 1]$, while the constant b follows $\mathcal{U}[-5n, 5n]$. 30 samples for each setting are used.

Volume Estimation. Random points are sampled within a box region defined by the bounds of the variables, where y_i adheres to the bounds of $\sigma(x_i)$. The volume of the function hull approximation $\mathcal{M}$ is estimated using the formula volume$(\mathcal{M}) = 1 - \frac{\#(\text{sample points outside } \mathcal{M})}{\#(\text{all sample points})}$. Our evaluation follows previous studies by taking 1 million random points (see the decimal digits in Table 4).

Baselines. We select two baseline methods for calculating multi-neuron constraints. Since the multi-neuron method SBLM+PDDM from PRIMA [Müller et al. 2022] only provides implementations for Sigmoid, Tanh, and MaxPool, we focus on these three functions. They are considered representative due to their non-linearity and expensive use in current multi-neuron approximation methods.

Evaluation metrics. Performance evaluation is divided into four key metrics, including 1) *precision*, measured by the volume of the resultant polytope, 2) *efficiency*, measured through computation time, 3) *constraints complexity*, measured by the number of constraints of the resulting approximation, and 4) *scalability*, indicative of the method's capacity to handle high-dimensional data.

Table 4. Performance of function hull over-approximation methods (Sigmoid, Tanh, and MaxPool).

Input Dim.	Sigmoid		Tanh		MaxPool	
	SBLM+PDDM	WraAct	SBLM+PDDM	WraAct	SBLM+PDDM	WraAct
			Runtime (*s*)			
2	**0.000669**±(**0.000046**)	0.000862±(0.000096)	**0.000662**±(**0.000054**)	0.001043±(0.001194)	**0.000119**±(**0.000031**)	0.000504±(0.001156)
3	0.009399±(0.001275)	**0.001960**±(**0.000264**)	0.009054±(0.001169)	**0.001894**±(**0.000084**)	0.000824±(0.000061)	**0.000397**±(**0.000032**)
4	1.138171±(0.736798)	**0.003042**±(**0.000376**)	0.841187±(0.392547)	**0.002740**±(**0.000395**)	0.011249±(0.001188)	**0.000908**±(**0.000157**)
			Estimated Volume			
2	0.144941±(0.058328)	**0.039445**±(0.035453)	0.182942±(0.046511)	**0.000508**±(0.002732)	0.098952±(0.035790)	**0.026968**±(0.014586)
3	0.039991±(0.023670)	**0.004283**±(**0.005420**)	0.067376±(0.021906)	**0.000003**±(**0.000013**)	0.071021±(0.028629)	**0.009600**±(**0.009362**)
4	0.003663±(0.002791)	**0.000026**±(**0.000073**)	0.014300±(0.006206)	-	0.026038±(0.009498)	**0.002638**±(**0.002344**)
			Number of Constraints			
2	**30.90**±(4.83)	37.60±(**0.76**)	**30.67**±(5.13)	37.60±(**0.61**)	**8.50**±(1.20)	16.00±(**0.00**)
3	158.40±(32.75)	**83.80**±(**0.95**)	148.93±(26.03)	**83.77**±(**0.42**)	31.07±(5.20)	**37.00**±(**0.00**)
4	1409.97±(530.94)	**202.73**±(**2.10**)	1188.03±(369.98)	**201.57**±(**0.67**)	162.73±(14.84)	**94.00**±(**0.00**)

The symbol "-" indicates that none of the randomly sampled points fall inside the function hull over-approximation.

5.1.2 Results. Table 4 presents the mean values and corresponding standard deviations for precision, efficiency, and constraints complexity over input dimensions from 2 to 4. The mean indicates overall performance, while the standard deviation captures variability in the results, with a lower standard deviation representing greater consistency across different samples. We also illustrate the scalability of WraAct in Figure 4, which measures the time consumption for handling high-dimensional polytopes with input dimensions up to 8. Note that the implementation of other multi-neuron methods (SBLM+PDDM) does not support input dimensions greater than 4.

Precision. All methods aim to over-approximate the function hull, resulting in approximations that inherently have a larger volume than the exact function hull. A more precise over-approximation is indicated by a smaller volume that closely matches the convex hull. Since computing the exact volume is impractical, we estimate volumes using random sampling. As shown in Table 4, WraAct outperforms SBLM+PDDM in terms of precision, exhibiting both lower mean volumes and smaller standard deviations. It achieves 20X–300X compared to the multi-neuron method SBLM+PDDM. WraAct also demonstrates a lower standard deviation compared to WraAct, indicating more stable performance.

Efficiency. As demonstrated in Table 4, WraAct consistently outperforms SBLM+PDDM, especially in higher dimensions, achieving up to 400X faster speeds.

Constraints complexity. The number of constraints in resulting polytopes indicates their simplicity, with fewer constraints often proving advantageous when employing optimization for subsequent verification. Experimental findings in Table 4 reveal that SBLM+PDDM exhibits a higher constraint count. Notably, WraAct reduces constraints by up to 90% compared to SBLM+PDDM. The constraint count of WraAct closely correlates with the characteristics of the input polytopes. Specifically, WraAct incorporates the same number of constraints as the input polytopes plus those stemming from single-neuron constraints.

Scalability in high-dimensional cases. Given the frequent involvement of high-dimensional data in NN verification, scaling effectively presents a significant challenge for approximation methods, as highlighted in other studies [Ma et al. 2024; Müller et al. 2022]. As depicted in Figure 4, WraAct exhibits robust scalability, successfully handling higher dimensions and completing the approximation task within 10s for scenarios up to 8 dimensions. We cap the input dimension at a maximum of 8, aligning with typical application domains like NNs, where a large number of constraints may introduce redundancy under common settings. The constraints of the input polytope influence the calculation, and a simpler input polytope will have better scalability. It is important to note that the current implementation of SBLM+PDDM is limited to an input dimension of no more than 4.

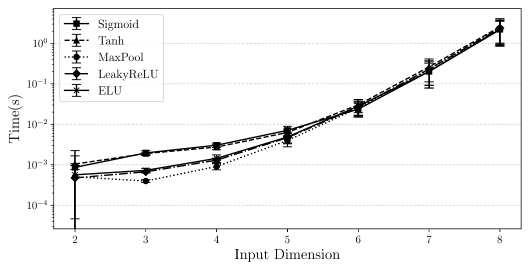


Fig. 4. Scalability analysis of WraAct.

5.2 Local Robustness Verification

This section delves into the performance of WraAct on neural network verification achieved through implementing a verifier based on WraAct (denoted by NNVerif). We focus on assessing 1) the enhancement from multi-neuron constraints compared to the SOTA multi-neuron constraint method SBLM+PDDM [Müller et al. 2022], 2) the applicability of NNVerif on various activation functions, and 3) the scalability of NNVerif with large-scale networks.

5.2.1 Verifier Implementation. We integrate WraAct into our neural network verification framework NNVerif. It combines bound propagation and linear programming approaches, supporting representative activation functions, including ReLU, Sigmoid, Tanh, leaky ReLU, and ELU. The bound propagation for Sigmoid and Tanh is based on the SOTA approaches CROWN [Weng et al. 2018; Zhang et al. 2018] and DeepPoly [Singh et al. 2019b]. NNVerif integrates the SOTA multi-neuron framework PRIMA [Müller et al. 2022] with a neuron grouping strategy. It is implemented as a multiple-staged verifier and attempts to verify a sample with 1) bound propagation, 2) linear programming with single-neuron constraints, and 3) multi-neuron constraints. Because the multi-neuron constraints have a large number, NNVerif adapts lazy constraints addition [Pearce 2019], *i.e.*, it only adds the multi-neuron constraints that contribute to the objective function. We also accept an early-stop setting if the range over-approximated by the bound propagation is too large to be further tightened by linear programming.

5.2.2 Experiment Settings. This section provides the evaluation metrics, benchmarks, and baselines.
Evaluation metrics. We assess the performance of all methods in verifying local robustness under a specified l_∞ perturbation. For each network, we define a specific ϵ perturbation radius (details in Appendix B.2.3). Two key metrics are used to assess the overall performance, *i.e.*, 1) *#Verified*, the number of verified samples within the first 100 correctly classified samples of the test dataset, and 2) *Time*, the total runtime of verification. These metrics provide valuable insights into the verification process's effectiveness and efficiency.
Benchmarks. We conduct experiments using diverse network architectures, including fully-connected networks (FCN), convolutional networks (CNN), and residual networks (ResNet) trained on the MNIST and CIFAR10 datasets. The activation functions of these networks include Sigmoid, Tanh, leaky ReLU, ELU, and MaxPool. Six benchmarks are from PRIMA [Müller et al. 2022], which include one fully-connected network (FCN-deep) and one convolutional network (CNN-base) equipped with Sigmoid and Tanh activation functions, and two MaxPool networks. These networks undergo training on the MNIST [Deng 2012] and CIFAR10 [Krizhevsky and Hinton 2009] datasets. Additionally, we expand our experiments to include other networks such as FCN-base and CNN-deep. Also, we train 8 networks using leaky ReLU and ELU activation functions, employing the same settings as those used for Sigmoid and Tanh functions. All the above networks are trained with normal training settings. To demonstrate the scalability of NNVerif, we introduce two large-scale ResNets trained by adversarial examples generated by the fast gradient sign method (FGSM) [Goodfellow et al. 2015], with an adversarial radius of 1/255. Table 10 in Appendix B.2 presents the network architecture in Section 5.2. The structures of all benchmarks achieve the SOTA level with different activation functions, like in recent studies [Ma et al. 2024; Müller et al. 2022; Wang et al. 2021; Zhang et al. 2022a, 2018].
Baselines. We perform a comparative analysis of NNVerif against single-neuron and multi-neuron approximate approaches. We choose two representative single-neuron approaches, CROWN [Weng et al. 2018; Zhang et al. 2018] and DeepPoly [Singh et al. 2019b], and one SOTA multi-neuron approach, PRIMA [Müller et al. 2022].
Multi-neuron settings. NNVerif employs the neuron grouping strategy, consistent with PRIMA [Müller et al. 2022]. This strategy involves combining neurons within the same layer into smaller groups. Specifically, we flatten the convolutional layer and group all neurons, and we take the inputs from the same MaxPool as a group. Three hyperparameters are used: n_s, k, and s. Here, n_s is the size of each partition, k is the input dimension of each group within each partition, and s is the maximum overlapping size between any two groups. We set $n_s = 50$, $k = 4$, and $s = 1$.

5.2.3 Experiment Results. This section presents the benchmark evaluation of Sigmoid and Tanh between NNVerif and other SOTA verifiers (DeepPoly, CROWN, and PRIMA), the extended evaluation on leaky ReLU and ELU, and scalability evaluation on large-scale residual networks.
Results on Sigmoid and Tanh. Table 5 showcases the performance of various approaches on Sigmoid and Tanh. NNVerif$_{\text{DeepPoly+MN}}$ is a bound propagation based on DeepPoly with linear programming of multi-neuron constraints. NNVerif$_{\text{CROWN+MN}}$ is similar but based on vanilla CROWN. NNVerif$_{\text{CROWN+MN}}$ demonstrates superior performance over its DeepPoly-based counterpart, achieving higher numbers of verified samples with tighter precision and reduced runtime, owing to tighter single neuron constraints from CROWN. In terms of the number of verified samples, NNVerif verifies up to 69 more samples than DeepPoly, 62 more than PRIMA, and 60 more than CROWN. Furthermore, compared to the SOTA multi-neuron verifier PRIMA, we observe improvements in runtime reduction with a 10X–40X faster runtime. Note that because the calculation of multi-neuron constraints has a small proportion of the total verification time, the runtime reduction of the whole verification process is smaller than the calculation in Section 5.1.

Table 5. Performance of local robustness verification enhanced by multi-neuron constraints on SOTA benchmarks.

Dataset	Activation Function	Network	DeepPoly		PRIMA		NNVerif$_{\text{DeepPoly+MN}}$		CROWN		NNVerif$_{\text{CROWN+MN}}$	
			#Verified	Time (s)	#Verified	Time (s)	#Verified	Time (s)	#Verified	Time (s)	#Verified	Time (s)
MNIST	Sigmoid	FCN - base	5	310.40	11	36717.69	46	2138.22	19	35.26	61	2855.09
		FCN - deep	16	1107.87	16	78022.67	43	6121.96	11	74.85	69	3976.76
		CNN - base	21	547.89	21	42069.53	36	3307.86	37	102.07	40	3598.75
		CNN - wide	13	471.80	-	-	58	6414.02	33	84.00	69	5019.85
	Tanh	FCN - base	8	352.46	9	33220.65	71	1218.81	55	100.91	76	544.23
		FCN - deep	10	1328.27	10	96568.91	11	179.80	1	234.66	42	602.26
		CNN - base	11	617.63	25	49998.96	50	3085.32	49	159.95	57	2730.30
		CNN - wide	16	615.39	16	36450.33	42	2715.70	18	100.40	50	2429.59
	MaxPool	CNN - pool	38	864.261	43	5764.36	40	1128.46	15	45.81	40	880.80
CIFAR10	Sigmoid	FCN - base	12	872.68	26	63825.99	81	3494.90	39	36.81	74	1756.03
		FCN - deep	12	2326.76	12	165181.21	32	4692.58	0	80.50	58	4414.35
		CNN - base	24	853.16	-	-	41	4407.01	43	108.70	50	3607.57
		CNN - wide	23	866.94	-	-	66	6152.96	5	92.54	57	5065.96
	Tanh	FCN - base	25	838.57	42	45407.74	78	950.59	72	99.86	76	195.15
		FCN - deep	4	2582.25	11	99343.96	16	543.53	2	240.40	53	1487.03
		CNN - base	0	1034.79	6	68964.61	33	3623.36	46	160.67	59	2609.55
		CNN - wide	5	958.78	-	-	52	5517.69	0	129.35	60	3828.50
	MaxPool	CNN - pool	5	7841.48	6	79978.48	6	7820.28	1	244.88	5	4754.13

"-" indicates that the experiment exceeded the 48-hour time limit due to efficiency considerations.

Table 6. Performance of local robustness verification enhanced by multi-neuron constraints on various functions.

Activation Function	Network	MNIST						CIFAR10					
		NNVerif$_{\text{BP}}$		NNVerif$_{\text{BP+SN}}$		NNVerif$_{\text{BP+MN}}$		NNVerif$_{\text{BP}}$		NNVerif$_{\text{BP+SN}}$		NNVerif$_{\text{BP+MN}}$	
		#Verified	Time (s)	#Verified	Time (s)	#Verified	Time (s)	#Verified	Time (s)	#Verified	Time (s)	#Verified	Time (s)
Leaky ReLU	FCN - base	13	8.23	13	158.71	21	416.08	16	9.39	17	350.17	38	1815.58
	FCN - deep	18	23.84	18	188.55	27	632.06	13	27.52	13	182.44	17	533.24
	CNN - base	59	86.22	59	337.11	89	2342.98	4	199.23	4	364.96	7	567.73
	CNN - wide	77	89.43	77	244.30	91	1395.45	8	203.76	8	818.24	24	1950.57
ELU	FCN - base	51	33.27	51	146.28	77	650.65	54	31.00	54	219.46	55	501.90
	FCN - deep	58	60.00	58	338.97	67	933.66	34	71.55	34	495.75	43	1943.57
	CNN - base	20	116.99	20	820.50	30	2941.16	18	209.57	18	1516.15	18	2020.42
	CNN - wide	32	116.44	32	734.77	43	2859.27	31	193.56	31	1150.41	32	1978.94

Results on leaky ReLU and ELU. The extended evaluation of leaky ReLU and ELU compares results from pure bound propagation (BP) and propagation combined with linear programming for single-neuron constraints (BP+SN) or multi-neuron constraints (BP+MN). As shown in Table 6, the use of multi-neuron constraints consistently enhances performance. With the leaky ReLU function, we observe an improvement of 3–30 verified examples compared to both pure BP and BP+SN. Similarly, for the ELU function, there is an improvement of up to 16 verified examples over pure BP and BP+SN. This modest gain compared to S-shaped functions is due to the relatively simple geometric characteristics of ReLU-like functions.

Table 7. Scalability evaluation of local robustness verification on ResNets.

Network	Activation Function	NNVerif$_{\text{BP}}$		NNVerif$_{\text{BP+SN}}$		NNVerif$_{\text{BP+MN}}$	
		#Verified	Time (s)	#Verified	Time (s)	#Verified	Time (s)
Tanh	ResNet-base	4	12.48	4	1269.88	5	1579.74
	ResNet-deep	0	81.10	0	81.43	0	81.49
Leaky ReLU	ResNet-base	54	12.41	54	259.45	57	445.82
	ResNet-deep	12	81.47	12	460.22	12	461.81
ELU	ResNet-base	85	12.53	85	131.27	87	228.67
	ResNet-deep	46	81.70	46	430.80	46	485.20

Results on large-scale networks. Table 7 presents the scalability evaluation of ResNet, conducted in a GPU-based experimental setup. This evaluation involves multiple verification phases, so a higher number of samples verified through bound propagation results in fewer samples verified using linear programming. Given that the ResNets in our setup have up to 10 hidden layers, the bound propagation approach introduces an unavoidable over-approximation, limiting the potential refinement offered by multi-neuron constraints. However, since the computation of multi-neuron constraints is independent of layers, the overall runtime remains reasonable, demonstrating the scalability of WraAct on large-scale networks. A better exploitation of multi-neuron remains an open future work.

6 Related Work

6.1 Convex Hull and Its Approximation

The convex hull problem involves constructing the minimal set of halfspaces whose intersection forms the convex hull encapsulating a given finite set of points. Convex hull algorithms are broadly categorized into two types: *pivoting algorithms*, which consider all vertices or faces simultaneously, and *incremental algorithms*, which iteratively process vertices or faces.

Several algorithms are built on Dantzig's simplex method. For instance, the reverse search technique [Avis and Fukuda 1991, 1992] applies the simplex method in reverse, while Balinski's algorithm [Balinski 1961] enumerates vertices facet by facet. Additionally, the gift wrapping algorithm [Chand and Kapur 1970] leverages the property that neighboring facets share lower-dimensional faces. Incremental algorithms, such as the Quickhull algorithm [Barber et al. 1996], construct the convex hull by processing one point at a time. The double description method [Fukuda and Prodon 1995; Motzkin et al. 1953] simultaneously considers the primal and dual spaces.

In the context of neural network verification, convex hull algorithms have been tailored to construct multi-neuron constraints. An expedient and effective approximate method WraLU [Ma et al. 2024] treats linear pieces of ReLU functions as known lower faces. It swiftly constructs new upper faces adjacent to these lower ones, leveraging the fact that one $(n-1)$-dimensional hyperplane and one point determine an n-dimensional hyperplane, and this approach ensures rapid computation and maintains a stable number of constraints. Another approach [Müller et al. 2022; Singh et al. 2019a] involves decomposing orthants based on the piecewise linearity property of the ReLU function to obtain vertices. This method, available in both versions of exact SBLM+DDM [Singh et al. 2019a] and approximate SBLM+PDDM [Müller et al. 2022], applies convex hull or approximation algorithms. OptC2V [Tjandraatmadja et al. 2020] exploits the submodularity and convexity of ReLU hulls. These methodologies rely on the piecewise linearity of the ReLU function. While some approaches may extend to Sigmoid and Tanh functions [Müller et al. 2022], they prove inefficient without specifically considering the properties of these functions.

6.2 Approaches of Neural Network Verification

Neural network verification has spawned many studies that have included exact and approximate approaches. Exact approaches are only applicable to linear piecewise functions such as ReLU and MaxPool functions by considering each linear piece [Bunel et al. 2020, 2018; Cheng et al. 2017; Dutta et al. 2018; Ferrari et al. 2021; Katz et al. 2017, 2019; Lomuscio and Maganti 2017; Tjeng et al. 2018; Wang et al. 2021].

Approximate methods offer greater speed and efficiency, and can be extended to a wider range of activation functions including Sigmoid, Tanh, leaky ReLU, and ELU by constructing linear constraints for bound propagation [Paulsen and Wang 2022; Singh et al. 2019b; Weng et al. 2018; Zhang et al. 2018, 2022b] and linear programming [Ferrari et al. 2021; Gehr et al. 2018; Ko et al. 2019; Ma et al. 2024; Müller et al. 2022; Ryou et al. 2021; Singh et al. 2019a; Xu et al. 2022]. Other methods utilize star sets [Tran et al. 2023] and sparse polynomial optimization [Newton and Papachristodoulou

2021, 2023] to over-approximate the function hulls for general activation functions, but they have limited scalability for large-scale networks and require specific prerequisites.

However, most existing approaches focus on single-neuron constraints of general activation functions [Boopathy et al. 2019; Henriksen and Lomuscio 2020; Paterson et al. 2021; Paulsen and Wang 2022; Shi et al. 2023; Singh et al. 2018, 2019b; Zhang et al. 2018, 2022b], and only a few consider multi-neuron constraints [Müller et al. 2022], which only consider Sigmoid and Tanh functions and are not specifically designed for general functions. This work concentrates on constructing function hull approximations for activation functions.

7 Conclusion

In summary, our novel approach, WraAct, addresses the challenge of verifying neural networks using general activation functions by constructing tight function hull over-approximations. Leveraging linear constraints to smooth curve fluctuations, WraAct achieves superior efficiency and precision compared to existing techniques. We pioneer the integration of ELU and leaky ReLU functions within this framework. The evaluation demonstrates its significant impact through enhanced performance by integrating WraAct into our verifier NNVerif, with more verified samples and reduced runtime. This advancement bolsters the reliability and efficiency of neural network verification methodologies.

Acknowledgment

We thank our anonymous reviewers for their constructive comments. This work is partially supported by UQ-IITD Research Academy and Australian Research Council Discovery Projects (DP230101196, DP240103068).

Data Availability Statement

The source code of WraAct is available at https://github.com/Trusted-System-Lab/WraAct.

References

David Avis and Komei Fukuda. 1991. A basis enumeration algorithm for linear systems with geometric applications. *Applied Mathematics Letters* 4, 5 (1991), 39–42. doi:10.1016/0893-9659(91)90141-h

David Avis and Komei Fukuda. 1992. A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. *Discrete & Computational Geometry* 8 (1992), 295–313. doi:10.1007/bf02293050

M. L. Balinski. 1961. An algorithm for finding all vertices of convex polyhedral sets. *J. Soc. Indust. Appl. Math.* 9, 1 (March 1961), 72–88. doi:10.1137/0109008

C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. 1996. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)* 22, 4 (1996), 469–483. doi:10.1145/235815.235821

Akhilan Boopathy, Tsui-Wei Weng, Pin-Yu Chen, Sijia Liu, and Luca Daniel. 2019. CNN-Cert: An efficient framework for certifying robustness of convolutional neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 3240–3247.

Rudy Bunel, P. Mudigonda, Ilker Turkaslan, Philip Torr, Jingyue Lu, and Pushmeet Kohli. 2020. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research (JMLR)* 21 (2020).

Rudy Bunel, Ilker Turkaslan, Philip HS Torr, Pushmeet Kohli, and M. Pawan Kumar. 2018. A unified view of piecewise linear neural network verification. *Advances in Neural Information Processing Systems (NeurIPS)* (2018), 4795–4804.

Yulong Cao, Ningfei Wang, Chaowei Xiao, Dawei Yang, Jin Fang, Ruigang Yang, Qi Alfred Chen, Mingyan Liu, and Bo Li. 2021. Invisible for both camera and LiDAR: Security of multi-sensor fusion based perception in autonomous driving under physical-world attacks. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 176–194. doi:10.1109/sp40001.2021.00076

Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 39–57. doi:10.1109/sp.2017.49

Donald R. Chand and Sham S. Kapur. 1970. An algorithm for convex polytopes. *Journal of the ACM (JACM)* 17, 1 (1970), 78–86.

Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. 2017. Maximum resilience of artificial neural networks. In *International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Springer, 251–268.

Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. 2016. Fast and accurate deep network learning by exponential linear units (ELUs). doi:10.48550/arXiv.1511.07289

Li Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142. doi:10.1109/msp.2012.2211477

Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. 2018. Output Range Analysis for Deep Feedforward Neural Networks. In *NASA Formal Methods*. Vol. 10811. Springer International Publishing, Cham, 121–138. doi:10.1007/978-3-319-77935-5_9 Series Title: Lecture Notes in Computer Science.

Ruediger Ehlers. 2017. Formal verification of piece-wise linear feed-forward neural networks. In *International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Springer, 269–286. doi:10.1007/978-3-319-68167-$2_1$9

Xinguo Feng, Zhongkui Ma, Zihan Wang, Eu Joe Chegne, Mengyao Ma, Alsharif Abuadbba, and Guangdong Bai. 2024. Uncovering Gradient Inversion Risks in Practical Language Model Training. In *Proceedings of the on ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, Salt Lake City, UT, USA, 3525–3539. doi:10.1145/3658644.3690292

Claudio Ferrari, Mark Niklas Mueller, Nikola Jovanović, and Martin Vechev. 2021. Complete verification via multi-neuron relaxation guided branch-and-bound. In *The International Conference on Learning Representations (ICLR)*.

Komei Fukuda and Alain Prodon. 1995. Double description method revisited. In *Franco-Japanese and Franco-Chinese Conference on Combinatorics and Computer Science*. Springer, 91–111.

Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. 2018. AI2: Safety and robustness certification of neural networks with abstract interpretation. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 3–18. doi:10.1109/sp.2018.00058

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep sparse rectifier neural networks. In *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*. JMLR Workshop and Conference Proceedings, 315–323.

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples. doi:10.48550/arXiv.1412.6572

Gurobi Optimization, LLC. 2023. Gurobi Optimizer Reference Manual. https://www.gurobi.com

Charles R. Harris, K. Jarrod Millman, Stéfan J. Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, and Nathaniel J. Smith. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

Dan Hendrycks and Kevin Gimpel. 2023. Gaussian error linear units (GELUs). doi:10.48550/arXiv.1606.08415

Patrick Henriksen and Alessio Lomuscio. 2020. Efficient neural network verification via adaptive refinement and adversarial search. In *European Conference on Artificial Intelligence (ECAI)*. IOS Press, 2513–2520.

Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, and Tara N. Sainath. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine* 29, 6 (2012), 82–97.

Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, and Vijay Vasudevan. 2019. Searching for MobileNetV3. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1314–1324.

Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. doi:10.48550/arXiv.1704.04861

Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer-Aided Verification (CAV)*. Springer, 97–117. doi:10.1007/978-3-319-63387-9_5

Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, and Aleksandar Zeljić. 2019. The Marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer-Aided Verification (CAV)*. Springer, 443–452. doi:10.1007/978-3-030-25540-$4_2$6

Ching-Yun Ko, Zhaoyang Lyu, Lily Weng, Luca Daniel, Ngai Wong, and Dahua Lin. 2019. POPQORN: Quantifying robustness of recurrent neural networks. In *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR, 3468–3477.

Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. (2009).

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems (NeurIPS)* 25 (2012).

Linyi Li, Tao Xie, and Bo Li. 2023. SoK: Certified robustness for deep neural networks. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 94–115. doi:10.1109/sp46215.2023.10179303

Alessio Lomuscio and Lalit Maganti. 2017. An approach to reachability analysis for feed-forward ReLU neural networks. doi:10.48550/arXiv.1706.07351

Zhongkui Ma. 2023. Verifying Neural Networks by Approximating Convex Hulls. In *Formal Methods and Software Engineering*. Vol. 14308. Springer Nature Singapore, Singapore, 261–266. doi:10.1007/978-981-99-7584-$6_1$7 Series Title: Lecture Notes in Computer Science.

Zhongkui Ma, Xinguo Feng, Zihan Wang, Shuofeng Liu, Mengyao Ma, Hao Guan, and Mark Huasong Meng. 2023. Formalizing Robustness Against Character-Level Perturbations for Neural Network Language Models. In *Formal Methods and Software Engineering*. Vol. 14308. Springer Nature Singapore, Singapore, 100–117. doi:10.1007/978-981-99-7584-6_7 Series Title: Lecture Notes in Computer Science.

Zhongkui Ma, Jiaying Li, and Guangdong Bai. 2024. ReLU Hull Approximation. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 2260–2287. doi:10.1145/3632917

Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. 2013. Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of the International Conference on Machine Learning (ICML)*, Vol. 30. PMLR, 3.

Mark Huasong Meng, Guangdong Bai, Sin Gee Teo, Zhe Hou, Yan Xiao, Yun Lin, and Jin Song Dong. 2022. Adversarial robustness of deep neural networks: A survey from a formal verification perspective. *IEEE Transactions on Dependable and Secure Computing* (2022).

Theodore S. Motzkin, Howard Raiffa, Gerald L. Thompson, and Robert M. Thrall. 1953. The Double Description Method. In *Contributions to the Theory of Games*. Vol. 2. 51–74.

Mark Niklas Müller, Gleb Makarchuk, Gagandeep Singh, Markus Püschel, and Martin Vechev. 2022. PRIMA: general and precise neural network certification via scalable convex hull approximations. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–33. doi:10.1145/3498704

Matthew Newton and Antonis Papachristodoulou. 2021. Neural network verification using polynomial optimisation. In *Conference on Decision and Control (CDC)*. IEEE, 5092–5097.

Matthew Newton and Antonis Papachristodoulou. 2023. Sparse polynomial optimisation for neural network verification. *Automatica* 157 (2023), 111233.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, and Luca Antiga. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems (NeurIPS)* 32 (2019).

Colin Paterson, Haoze Wu, John Grese, Radu Calinescu, Corina S. Păsăreanu, and Clark Barrett. 2021. DeepCert: Verification of contextually relevant robustness for neural network image classifiers. In *Computer Safety, Reliability, and Security: 40th International Conference (SAFECOMP)*. Springer, 3–17.

Brandon Paulsen and Chao Wang. 2022. LinSyn: Synthesizing tight linear bounds for arbitrary neural network activation functions. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 357–376.

Robin Harris Pearce. 2019. *Towards a general formulation of lazy constraints*. Ph. D. Dissertation. The University of Queensland.

Wonryong Ryou, Jiayu Chen, Mislav Balunovic, Gagandeep Singh, Andrei Dan, and Martin Vechev. 2021. Scalable polyhedral berification of recurrent neural networks. In *International Conference on Computer-Aided Verification (CAV)*. Springer, 225–248. doi:10.1007/978-3-030-81685-$8_1$0

Hadi Salman, Greg Yang, Huan Zhang, Cho-Jui Hsieh, and Pengchuan Zhang. 2019. A convex relaxation barrier to tight robustness verification of neural networks. *Advances in Neural Information Processing Systems (NeurIPS)* (2019), 9835–9846.

Zhouxing Shi, Qirui Jin, Huan Zhang, Zico Kolter, Suman Jana, and Cho-Jui Hsieh. 2023. Formal verification for neural networks with general nonlinearities via branch-and-bound. In *Workshop on Formal Verification of Machine Learning (WFVML)*.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, and Marc Lanctot. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.

Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. 2019a. Beyond the single neuron convex barrier for neural network certification. *Advances in Neural Information Processing Systems (NeurIPS)* 32 (2019).

Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. 2018. Fast and effective robustness certification. *Advances in Neural Information Processing Systems (NeurIPS)* 31 (2018).

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019b. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–30. doi:10.1145/3290354

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems (NeurIPS)* 27 (2014).

Christian Tjandraatmadja, Ross Anderson, Joey Huchette, Will Ma, Krunal Kishor Patel, and Juan Pablo Vielma. 2020. The convex relaxation barrier, revisited: Tightened single-neuron relaxations for neural network verification. *Advances in Neural Information Processing Systems (NeurIPS)* 33 (2020), 21675–21686.

Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. 2018. Evaluating robustness of neural networks with mixed integer programming. In *The International Conference on Learning Representations (ICLR).*

Hoang Dung Tran, Sung Woo Choi, Xiaodong Yang, Tomoya Yamaguchi, Bardh Hoxha, and Danil Prokhorov. 2023. Verification of recurrent neural networks with star reachability. In *International Conference on Hybrid Systems: Computation and Control (HSCC).* ACM, San Antonio TX USA, 1–13. doi:10.1145/3575870.3587128

Matthias Troffaes. 2018. pycddlib: Python wrapper for Komei Fukuda's cddlib. https://pypi.org/project/pycddlib/

Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J. Zico Kolter. 2021. Beta-CROWN: Efficient bound propagation with per-neuron split constraints for neural network robustness verification. *Advances in Neural Information Processing Systems (NeurIPS)* 34 (2021), 29909–29921.

Zihan Wang, Zhongkui Ma, Xinguo Feng, Zhiyang Mei, Ethan Ma, Derui Wang, Minhui Xue, and Guangdong Bai. 2025. AI Model Modulation with Logits Redistribution. In *Proceedings of the ACM on Web Conference 2025.* ACM, Sydney NSW Australia, 4699–4709. doi:10.1145/3696410.3714737

Zihan Wang, Zhongkui Ma, Xinguo Feng, Ruoxi Sun, Hu Wang, Minhui Xue, and Guangdong Bai. 2024. Corelocker: Neuron-level usage control. In *IEEE Symposium on Security and Privacy (S&P).* IEEE, 2497–2514.

Lily Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane Boning, and Inderjit Dhillon. 2018. Towards fast computation of certified robustness for relu networks. In *Proceedings of the International Conference on Machine Learning (ICML).* PMLR, 5276–5285.

Zhiwu Xu, Yazheng Liu, Shengchao Qin, and Zhong Ming. 2022. Output range analysis for feed-forward deep neural networks via linear programming. *IEEE Transactions on Reliability* (2022).

Yuchen Yang, Bo Hui, Haolin Yuan, Neil Gong, and Yinzhi Cao. 2024. Sneakyprompt: Jailbreaking text-to-image generative models. In *IEEE Symposium on Security and Privacy (S&P).* IEEE, 897–912. doi:10.1109/SP54263.2024.00123

Zhiyuan Yu, Xiaogeng Liu, Shunning Liang, Zach Cameron, Chaowei Xiao, and Ning Zhang. 2024. Don't Listen To Me: Understanding and Exploring Jailbreak Prompts of Large Language Models. In *USENIX Security Symposium (USENIX Security).* USENIX Association, 4675–4692.

Huan Zhang, Shiqi Wang, Kaidi Xu, Linyi Li, Bo Li, Suman Jana, Cho-Jui Hsieh, and J. Zico Kolter. 2022a. General cutting planes for bound-propagation-based neural network verification. *Advances in Neural Information Processing Systems (NeurIPS)* 35 (2022), 1656–1670.

Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. 2018. Efficient neural network robustness certification with general activation functions. *Advances in Neural Information Processing Systems (NeurIPS)* 31 (2018), 4944–4953.

Zhaodi Zhang, Yiting Wu, Si Liu, Jing Liu, and Min Zhang. 2022b. Provably tightest linear approximation for robustness verification of sigmoid-like neural networks. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE).* ACM, Rochester MI USA, 1–13. doi:10.1145/3551349.3556907

Appendix

A Details of WraAct

A.1 Numerical Results of Running Example

This section presents the calculation details of important variables in Section 3.2. The coefficients and constants from the real algorithm are rounded for simplicity. The input domain is defined by a polytope as

$$\mathcal{X} = \{x_1 + x_2 \geq -2,\ -x_1 - x_2 \geq -2,\ x_1 - x_2 \geq -2,\ -x_1 + x_2 \geq -2\}.$$

The DLP functions as bounds with the first input dimension and output dimension,

$$\underline{\sigma}_1(x_1) = \min\{0.22x_1 + 0.52,\ 0.84x_1 - 0.04\}, \quad \overline{\sigma}_1(x_1) = \max\{0.22x_1 - 0.52,\ 0.84x_1 + 0.04\}.$$

and ones with the second input dimension and output dimension,

$$\underline{\sigma}_2(x_2) = \min\{0.22x_2 + 0.52,\ 0.84x_2 - 0.04\}, \quad \overline{\sigma}_2(x_2) = \max\{0.22x_2 - 0.52,\ 0.84x_2 + 0.04\}.$$

The convex over-approximations of two DLP functions in (x_1, x_2, y_1)-space are

$$\underline{\mathcal{M}}_1 = \{ \begin{array}{r} 0.22x_1 - y_1 \geq -0.52, \\ 0.84x_1 - y_1 \geq 0.04, \\ -x_1 - x_2 + y_1 \geq -2, \\ -3.98x_1 + x_2 + 5.92y_1 \geq -2.26, \\ -x_1 + x_2 + y_1 \geq -2, \\ -3.98x_1 - x_2 + 5.92y_1 \geq -2.26 \end{array} \}, \quad \overline{\mathcal{M}}_1 = \{ \begin{array}{r} -0.22x_1 + y_1 \geq 0.52, \\ -0.84x_1 + y_1 \geq -0.04, \\ x_1 + x_2 + y_1 \geq -2, \\ 3.98x_1 - x_2 - 5.92y_1 \geq -2.26, \\ x_1 - x_2 + y_1 \geq -2, \\ 3.98x_1 + x_2 - 5.92y_1 \geq -2.26 \end{array} \}.$$

The convex over-approximations of two DLP functions in (x_1, x_2, y_1, y_2)-space are

$$\underline{\mathcal{M}} = \{ \begin{array}{r} 0.22x_1 - y_1 \geq -0.52, \\ 0.84x_1 - y_1 \geq 0.04, \\ 0.22x_2 - y_2 \geq -0.52, \\ 0.84x_2 - y_2 \geq 0.04, \\ -3.98x_1 - 3.98x_2 + 5.92y_1 + 5.92y_2 \geq -2.52, \\ -x_1 - 3.98x_2 + 5.92y_2 \geq -2.26, \\ -3.98x_1 - x_2 + 5.92y_1 \geq -2.26, \\ -x_1 - x_2 \geq -2 \end{array} \}, \quad \overline{\mathcal{M}} = \{ \begin{array}{r} -0.22x_1 + y_1 \geq 0.52, \\ -0.84x_1 + y_1 \geq -0.04, \\ -0.22x_2 + y_2 \geq 0.52, \\ -0.84x_2 + y_2 \geq -0.04, \\ 3.98x_1 + 3.98x_2 - 5.92y_1 - 5.92y_2 \geq -2.52, \\ x_1 + 3.98x_2 - 5.92y_2 \geq -2.26, \\ 3.98x_1 + x_2 - 5.92y_1 \geq -2.26, \\ x_1 + x_2 \geq -2 \end{array} \}.$$

The over-approximation with multi-neuron constraints of the activation function in (x_1, x_2, y_1, y_2)-space is

$$\mathcal{M}_m = \{ \begin{array}{r} -3.98x_1 - 3.98x_2 + 5.92y_1 + 5.92y_2 \geq -2.52, \\ 3.98x_1 + 3.98x_2 - 5.92y_1 - 5.92y_2 \geq -2.52, \\ -x_1 - 3.98x_2 + 5.92y_2 \geq -2.26, \\ -3.98x_1 - x_2 + 5.92y_1 \geq -2.26, \\ x_1 + 3.98x_2 - 5.92y_2 \geq -2.26, \\ 3.98x_1 + x_2 - 5.92y_1 \geq -2.26, \\ -x_1 - x_2 \geq -2, \\ x_1 + x_2 \geq -2 \end{array} \}.$$

A.2 Soundness of DLP Over-approximation

2-dimensional Example of Lemma 2. Considering a 2-dimensional example, if we have $y = g(x) = \max\{-x + 1, 2x + 1\}$ in the space of (x, y), we aim to get a linear transformation defined by a square matrix $\boldsymbol{A} \in \mathbb{R}^{2\times 2}$ and a vector $\boldsymbol{o} \in \mathbb{R}^2$ such that $\boldsymbol{A}$ and $\boldsymbol{o}$ transforms the point (x, y) on $y = -x$ (or $y = 2x + 1$) to the point (x', y') on $y = 0$ (or $y = x$) by $(x', y') = \boldsymbol{A} \cdot ((x, y) - \boldsymbol{o})$. Specifically, the vector $\boldsymbol{o}$ is a translation transformation to translate the intersection of $y = 2x + 1$ and $y = -x$ to the original and is defined as $\boldsymbol{o} = (0, 1)$. Next, we construct $\boldsymbol{V} = \left[\begin{smallmatrix} 1 & 1 \\ -1 & 2 \end{smallmatrix}\right]$, where the first column vector $(1, -1)$ is the direction vector of $y = -x + 1$ and the second column vector $(1, 2)$ is the direction vector of $y = 2x + 1$ and $\boldsymbol{V}' = \left[\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right]$, where the first column vector $(1, 0)$ is the direction vector of $y = 0$ and the second column vector $(1, 1)$ is the direction vector of $y = x$. Then, let $\boldsymbol{A} = \boldsymbol{V}'\boldsymbol{V}^{-1} = \boldsymbol{V}' \left[\begin{smallmatrix} 2/3 & -1/3 \\ 1/3 & 1/3 \end{smallmatrix}\right] = \left[\begin{smallmatrix} 1 & 0 \\ 1/3 & 1/3 \end{smallmatrix}\right]$. So, for instance, we can transform $(1, 3)$ on $y = 2x + 1$ to $(1, 1)$ on $y = x$ by $\boldsymbol{A} \cdot ((1, 3) - \boldsymbol{o}) = \boldsymbol{A} \left[\begin{smallmatrix} 1 \\ 2 \end{smallmatrix}\right] = (1, 1)$; we can transform $(-1, 2)$ on $y = -x + 1$ to $(-1, 0)$ on $y = 0$ by $\boldsymbol{A} \cdot ((-1, 2) - \boldsymbol{o}) = \boldsymbol{A} \left[\begin{smallmatrix} -1 \\ 1 \end{smallmatrix}\right] = (-1, 0)$; we can transform $(0, 4) = (1, 4) + (-1, 1)$ on the coordinate system of line $y = -x + 1$ and line $y = 2x + 1$ to $(0, 1)$ on the coordinate system of line $y = 0$ and $y = x$ by $\boldsymbol{A} \cdot ((0, 4) - \boldsymbol{o}) = \boldsymbol{A} \left[\begin{smallmatrix} 0 \\ 3 \end{smallmatrix}\right] = (0, 1)$.

4-dimensional Example of Lemma 2. Considering a 4-dimensional example, if we have $y = g(x_1) = \max\{-x_1 - x_2 + 1, 2x_1 + x_2 + 2\}$ in the space of (x_1, x_2, x_3, y), we aim to get a linear transformation defined by a square matrix $\boldsymbol{A} \in \mathbb{R}^{4\times 4}$ and a vector $\boldsymbol{o} \in \mathbb{R}^4$ transforms the point (x_1, x_2, x_3, y) on $y = -x_1 - x_2 + 1$ (or $y = 2x_1 + x_2 + 2$) to the point (x_1', x_2', x_3', y') on $y = 0$ (or $y = x_1$) by $(x_1', x_2', x_3', y') = \boldsymbol{A} \cdot ((x_1, x_2, x_3, y) - \boldsymbol{o})$. Specifically, to determine the vector $\boldsymbol{o}$, we need to translate the interaction of $y = -x_1 - x_2 + 1$ and $y = 2x_1 + x_2 + 2$ to crossing the original. Then, we need to find a solution to the equation system $\left\{\begin{smallmatrix} y=-x_1-x_2+1 \\ y=2x_1+x_2+2 \end{smallmatrix}\right.$ to translate these two planes to $\left\{\begin{smallmatrix} y=-x_1-x_2 \\ y=2x_1+x_2 \end{smallmatrix}\right.$ So we set $\boldsymbol{o} = (-1, 1, 0, 1)$. Next, we construct $\boldsymbol{V} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -2 & 3 & 0 & 0 \end{bmatrix}$, where the first column vector $(1, 1, 0, -2)$ is a vector parallel to $y = -x_1 - x_2 + 1$, the second column vector $(1, 1, 0, 3)$ is a vector parallel to $y = 2x_1 + x_2 + 2$ (note that the first two column maintain the measurement of $\boldsymbol{V}$), and the third and the fourth column vectors are determined by x_2-axis and x_3-axis, which makes $\boldsymbol{V}$ invertible. Let $\boldsymbol{V}' = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$, where the first column vector $(1, 0, 0, 0)$ is a vector parallel to $y = 0$ and the second column vector $(1, 0, 0, 1)$ is a vector parallel to $y = x_1$, and the third and the fourth column vectors are determined by x_2-axis and x_3-axis, which makes $\boldsymbol{V}'$ invertible. Such $\boldsymbol{V}'$ will not change the coordinate values on x_2 and x_3. Further, let $\boldsymbol{A} = \boldsymbol{V}'\boldsymbol{V}^{-1} = \boldsymbol{V}' \begin{bmatrix} 3/5 & 0 & 0 & -1/5 \\ 2/5 & 0 & 0 & 1/5 \\ -1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2/5 & 0 & 0 & 1/5 \end{bmatrix}$. So, for instance, we can transform $(0, 2, 0, 4)$ on $y = 2x_1 + x_2 + 2$ (note that $(0, 2, 0, 4) - \boldsymbol{o} = (1, 1, 0, 3)$ on $y = 2x_1 + x_2$) to $(1, 0, 0, 1)$ on $y = x_1$ by $\boldsymbol{A} \cdot ((0, 2, 0, 4) - \boldsymbol{o}) = \boldsymbol{A} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 3 \end{bmatrix} = (1, 0, 0, 1)$.

A.3 Single-neuron Constraints of Activation Functions

Detailed Proof of Lemma 6. The following shows the detailed proof of Lemma 6, showing the soundness of single neuron constraints for Sigmoid, Tanh, LeakyReLU, and ELU functions.

Proof. We prove each linear constraint is sound to the activation function σ in the input domain $\mathcal{X}$. Each linear constraint determines a linear function. If the linear function value is always less than the target function value, it determines a lower linear constraint for the target function; it is symmetrical for the upper constraints. Before diving into the detailed cases, we clarify several properties of convex/concave functions.

Property 1. ❶ For a function $f(x)$ and a linear function $g(x)$, if $f(x)$ is convex, *i.e.*, $f''(x) \geq 0$, within $x \in [a, b]$ and $g(a) \geq f(a)$ and $g(b) \geq f(b)$, we have $g(x) \geq f(x)$ and $g(x)$ provides an upper constraint for $f(x)$. We can prove this by the difference $d(x) = g(x) - f(x)$. Because $d(a) \geq 0$, $d(b) \geq 0$ and $d''(x) = -f''(x) \leq 0$, we have $d(x)$ is concave. Then, we have $d(x) \geq 0$ within $x \in [a, b]$ by the definition of concavity. So, we prove $f(x) \leq g(x)$ within $x \in [a, b]$. ❷ The case of a $f(x)$ and a linear function $g(x)$ as the lower bound, such that $g(a) \leq f(a)$, $g(b) \leq f(b)$, and $f(x)$ is concave within $x \in [a, b]$, is symmetrical.

Property 2. ❶ For a function $f(x)$ and a linear function $g(x)$, if $f(x)$ is convex, *i.e.*, $f''(x) \geq 0$, within $x \in [a, b]$ and $g(x)$ is a tangent line of $f(x)$ satisfying $g(m) = f(m)$ and $g'(m) = f'(m)$, where $m \in [a, b]$, we have $g(x) \leq f(x)$ and $g(x)$ provides a lower constraint for $f(x)$. We can prove this by the difference $d(x) = f(x) - g(x)$. Because $d''(x) = f''(x) \geq 0$, $d(x)$ is convex. Considering $d(m) = f(m) - g(m) = 0$, we have $d(x) \geq 0$ within $x \in [a, b]$ by the definition of convexity. So, we prove $g(x) \leq f(x)$ within $x \in [a, b]$. ❷ The case of a $f(x)$ and a linear function $g(x)$ as the upper bound, such that $g(m) = f(m)$, $g'(m) = f(m)$ ($m \in [a, b]$), and $f(x)$ is concave within $x \in [a, b]$, is symmetrical.

Property 3. ❶ For a function $f(x)$ and a linear function $g(x)$, if $f(x)$ is convex, *i.e.*, $f''(x) \geq 0$, within $x \in [a, b]$ and $h(x)$ is a line satisfying $h(a) = f(a)$ and $h'(a) < f'(a)$, we have $h(x) \leq f(x)$ and $h(x)$ provides a lower constraint for $f(x)$. This can be proved by setting $g(x)$ in Property 2 such that $m = a$ and $h(x) \leq g(x) \leq f(x)$ within $x \in [a, b]$. Similarly, ❷ for a function $f(x)$ and a linear function $g(x)$, if $f(x)$ is convex, *i.e.*, $f''(x) \geq 0$, within $x \in [a, b]$ and $h(x)$ is a line satisfying $h(b) = f(b)$ and $h'(b) > f'(b)$, we have $h(x) \leq f(x)$ and $h(x)$ provides a lower constraint for $f(x)$.

Symmetrically, we have that, ❸ for a function $f(x)$ and a linear function $g(x)$, if $f(x)$ is concave, *i.e.*, $f''(x) \leq 0$, within $x \in [a, b]$ and $h(x)$ is a line satisfying $h(a) = f(a)$ and $h'(a) > f'(a)$, we have $h(x) \geq f(x)$ and $h(x)$ provides a upper constraint for $f(x)$; ❹ for a function $f(x)$ and a linear function $g(x)$, if $f(x)$ is concave, *i.e.*, $f''(x) \leq 0$, within $x \in [a, b]$ and $h(x)$ is a line satisfying $h(a) = f(a)$ and $h'(b) < f'(b)$, we have $h(x) \geq f(x)$ and $h(x)$ provides a upper constraint for $f(x)$.

For sigmoid and Tanh functions, we have the following cases.

- When $\sigma'(u) \geq k_{lu} \wedge u \leq 0$, we have $\sigma(x)$ is convex within $x \in [l, u]$. 1) For the upper constraint $y = k_{lu}(x - l) + \sigma(l)$, it satisfies Property 1-❶. 2) For the lower constraint $y \geq \sigma'(l)(x - l) + \sigma(l)$, it satisfies Property 2-❶. 3) For the lower constraint $y \geq \sigma'(u)(x - u) + \sigma(u)$, it satisfies Property 2-❶. 4) For the lower constraint $y \geq k_{lu}(x - m_l) + \sigma(m_l)$, it satisfies Property 2-❶.
- When $\sigma'(u) \geq k_{lu} \wedge u > 0$, we have the following cases. 1) For the upper constraint $y = k_{lu}(x - l) + \sigma(l)$, it satisfies Property 1-❶ when $x \in [l, 0]$ and satisfies Property 2-❹ when $x \in [0, u]$. 2) For the lower constraint $y \geq \sigma'(l)(x - l) + \sigma(l)$, it satisfies Property 2-❶ when $x \in [l, 0]$ and satisfies Property 1-❷ when $x \in [0, u]$. 3) For the lower constraint $y \geq k_l(x - u) + \sigma(u)$, it satisfies Property 2-❶ when $x \in [l, 0]$ and satisfies Property 1-❷ when $x \in [0, u]$. 4) For the lower constraint $y \geq k_{lu}(x - m_l) + \sigma(m_l)$, it satisfies Property 2-❶ when $x \in [l, 0]$ and satisfies Property 1-❷ when $x \in [0, u]$.
- When $\sigma'(l) \geq k_{lu} \wedge l \geq 0$, this is a symmetrical case of $\sigma'(u) \geq k_{lu} \wedge u \leq 0$ and it can be proved by the symmetrical center of the Sigmoid or Tanh functions.
- When $\sigma'(l) \geq k_{lu} \wedge l < 0$, this is a symmetrical case of $\sigma'(u) \geq k_{lu} \wedge u > 0$, and it can be proved by the symmetrical center of the Sigmoid or Tanh functions.
- When $k_{lu} > \sigma'(l) \wedge k_{lu} > \sigma'(u)$, the lower (or upper) bounds have the same proof of the lower constraint when $\sigma'(u) \geq k_{lu} \wedge u > 0$ (or $\sigma'(u) \geq k_{lu} \wedge u > 0$).

For a LeakyReLU function, it is a convex function similar to the ReLU function. It has two linear pieces of $y = \alpha x$ $(0 < \alpha < 1)$ when $x < 0$ and $y = x$ when $x \geq 0$. We only discuss the nontrivial case.

- For the upper constraint $y \geq k_{lu}(x - l) + \sigma(l)$, it satisfies Property 1-❶.
- For the lower constraint $y \geq \alpha x$, it satisfies Property 2-❶.
- For the lower constraint $y \geq x$, it satisfies Property 2-❶.

For an ELU function, it is a convex function similar to the ReLU function. It has two pieces of $y = e^x - 1$ when $x < 0$ and $y = x$ when $x \geq 0$. We only discuss the nontrivial case.

- For the upper constraint $y \leq k_{lu}(x - l) + \sigma(l)$, it satisfies Property 1-❶.
- For the lower constraint $y \geq \sigma'(l)(x - l) + \sigma(l)$, it satisfies Property 2-❶.
- For the lower constraint $y \geq \sigma'(u)(x - u) + \sigma(u)$, it satisfies Property 2-❶.
- For the lower constraint $y \geq \sigma'(m)(x - m) + \sigma(m)$, it satisfies Property 2-❶.

□

A.4 Discussion of Optimality

WraAct constructs DLP functions to serve as bounds for the target activation function. The tighter these DLP bounds approximate the activation function from above and below, the tighter the resulting function hull over-approximation becomes. To improve efficiency, WraAct adopts an iterative procedure that processes each output dimension separately. This strategy, however, neglects potential inter-variable constraints that may arise across different iteration orders.

We further analyze the types of constraints that WraAct cannot capture. For a multi-input multi-output function, accurately approximating the surface often requires linear faces to tightly

Table 8. Ablation study of function hull over-approximation by single-neuron method and WraAct (Sigmoid, Tanh, MaxPool, LeakyReLU, and ELU).

Input Dim.	Sigmoid		Tanh		MaxPool	
	Single-neuron	WraAct	Single-neuron	WraAct	Single-neuron	WraAct
			Runtime (s)			
2	**0.000333**±(**0.000040**)	0.000862±(0.000096)	**0.000277**±(**0.000020**)	0.001043±(0.001194)	**0.000027**±(**0.000007**)	0.000504±(0.001156)
3	**0.000456**±(**0.000030**)	0.001960±(0.000264)	**0.000383**±(**0.000015**)	0.001894±(0.000084)	**0.000026**±(**0.000005**)	0.000397±(0.000032)
4	**0.000599**±(**0.000033**)	0.003042±(0.000376)	**0.000506**±(**0.000024**)	0.002740±(0.000395)	**0.000028**±(**0.000005**)	0.000908±(0.000157)
			Estimated Volume			
2	0.221187±(0.150489)	**0.039445**±(**0.035453**)	0.000904±(0.004827)	**0.000508**±(**0.002732**)	0.398086±(0.049817)	**0.026968**±(**0.014586**)
3	0.133869±(0.098536)	**0.004283**±(**0.005420**)	0.000010±(0.000037)	**0.000003**±(**0.000013**)	0.361226±(0.055736)	**0.009600**±(**0.009362**)
4	0.042201±(0.042275)	**0.000026**±(**0.000073**)	0.000002±(0.000009)	-	0.352711±(0.059871)	**0.002638**±(**0.002344**)
			Number of Constraints			
2	**11.00**±(1.34)	37.60±(**0.76**)	**11.20**±(1.22)	37.60±(**0.61**)	**3.00**±(**0.00**)	16.00±(**0.00**)
3	**16.87**±(1.52)	83.80±(**0.95**)	**17.53**±(0.85)	83.77±(**0.42**)	**4.00**±(**0.00**)	37.00±(**0.00**)
4	**22.27**±(**1.77**)	202.73±(2.10)	**23.13**±(1.33)	201.57±(**0.67**)	**5.00**±(**0.00**)	94.00±(**0.00**)
Input Dim.	LeakyReLU		ELU			
	Single-neuron	WraAct	Single-neuron	WraAct		
			Runtime (s)			
2	**0.000040**±(**0.000009**)	0.000473±(0.000493)	**0.000068**±(**0.000008**)	0.000564±(0.000518)		
3	**0.000040**±(**0.000007**)	0.000670±(0.000045)	**0.000070**±(**0.000008**)	0.000726±(0.000089)		
4	**0.000042**±(**0.000006**)	0.001300±(0.000162)	**0.000071**±(**0.000007**)	0.001441±(0.000314)		
			Estimated Volume			
2	0.059564±(0.031370)	**0.029496**±(**0.018156**)	0.052095±(0.031948)	**0.027033**±(**0.017319**)		
3	0.013626±(0.008140)	**0.002105**±(**0.001300**)	0.012190±(0.010080)	**0.004053**±(**0.003013**)		
4	0.003741±(0.002238)	**0.000147**±(**0.000094**)	0.002664±(0.001231)	**0.000710**±(**0.000737**)		
			Number of Constraints			
2	**6.00**±(**0.00**)	19.00±(**0.00**)	**8.00**±(**0.00**)	21.43±(0.62)		
3	**9.00**±(**0.00**)	42.00±(**0.00**)	**12.00**±(**0.00**)	45.57±(0.72)		
4	**12.00**±(**0.00**)	101.00±(**0.00**)	**16.00**±(**0.00**)	105.97±(0.87)		

"-" means that no randomly sampled point drops inside the function hull over-approximation.

"wrap" local non-convex or non-concave ("bowl-like") shapes. Consider a 4D surface defined as $\mathcal{S} = \{(x_1, x_2, y_1, y_2) \mid y_1 = \text{sigmoid}(x_1),\ \ y_2 = \text{sigmoid}(x_2)\}$. This surface can be partitioned into four regions: Area ① defined by $\mathcal{S} \cap \{(x_1, x_2, y_1, y_2) \mid x_1 \leq 0, x_2 \leq 0\}$, Area ② defined by $\mathcal{S} \cap \{(x_1, x_2, y_1, y_2) \mid x_1 \leq 0, x_2 > 0\}$, Area ③ defined by $\mathcal{S} \cap \{(x_1, x_2, y_1, y_2) \mid x_1 > 0, x_2 \leq 0\}$, and Area ④ defined by $\mathcal{S} \cap \{(x_1, x_2, y_1, y_2) \mid x_1 > 0, x_2 > 0\}$. Considering, the sigmoid function is convex within $(-\infty, 0]$ and concave within $[0, \infty)$. Note that the sigmoid function is convex on $(-\infty, 0]$ and concave on $[0, \infty)$. In Area ① and Area ④, both $y_1 = \text{sigmoid}(x_1)$ and $y_2 = \text{sigmoid}(x_2)$ share the same convexity or concavity. However, in Areas ② and ③, one of the components is convex while the other is concave. Because WraAct handles the upper convex and lower concave DLP functions independently rather than jointly (in Algorithm 1), it cannot capture constraints that depend on such locally mixed convex-concave structures.

B Evaluation

B.1 Function Hull Approximation

B.1.1 Ablation Study with Single-neuron Function Hull Approximation. Table 8 shows the ablation study of function hull over-approximation between the single-neuron method (defined in Table 3) and WraAct for Sigmoid, Tanh, LeakyReLU, ELU, and MaxPool functions. The results show that WraAct yields high performance on precision (smaller volume up to 200X), although it is compared to tight single-neuron constraints.

Table 9. Ablation study of function hull over-approximation by different designs of DLP functions in WraAct (Sigmoid, Tanh, ELU, and MaxPool).

Input Dim.	Sigmoid			Tanh		
	Design A	Design B	WraAct	Design A	Design B	WraAct
			Runtime (s)			
2	0.001068±(0.001105)	0.000878±(**0.000046**)	**0.000862**±(0.000096)	**0.000842**±(0.000037)	0.000844±(**0.000036**)	0.001043±(0.001194)
3	0.001922±(**0.000129**)	**0.001448**±(0.000132)	0.001960±(0.000264)	0.001454±(0.000060)	**0.001453**±(**0.000054**)	0.001894±(0.000084)
4	0.003042±(0.000384)	**0.002242**±(**0.000305**)	0.003042±(0.000376)	0.002434±(**0.000187**)	**0.002429**±(0.000188)	0.002740±(0.000395)
			Estimated Volume			
2	**0.031486**±(**0.027081**)	0.040257±(0.034297)	0.039445±(0.035453)	**0.000477**±(**0.002562**)	0.000529±(0.002843)	0.000508±(0.002732)
3	**0.003754**±(**0.004630**)	0.004605±(0.006236)	0.004283±(0.005420)	**0.000003**±(0.000013)	**0.000003**±(**0.000011**)	**0.000003**±(0.000013)
4	0.000047±(0.000139)	**0.000004**±(**0.000014**)	0.000026±(0.000073)	-	-	-
			Number of Constraints			
2	**37.60**±(0.76)	37.67±(**0.75**)	**37.60**±(0.76)	**37.60**±(**0.61**)	**37.60**±(**0.61**)	**37.60**±(**0.61**)
3	**83.77**±(**0.92**)	84.07±(1.18)	83.80±(0.95)	**83.77**±(**0.42**)	**83.77**±(**0.42**)	**83.77**±(**0.42**)
4	**202.47**±(**1.91**)	203.47±(2.29)	202.73±(2.10)	**201.57**±(**0.67**)	**201.57**±(**0.67**)	**201.57**±(**0.67**)

Input Dim.	ELU		MaxPool	
	Design A	WraAct	Design A	WraAct
		Runtime (s)		
2	**0.000423**±(**0.000049**)	0.000564±(0.000518)	**0.000308**±(**0.000069**)	0.000504±(0.001156)
3	**0.000726**±(**0.000064**)	**0.000726**±(0.000089)	0.000411±(**0.000032**)	**0.000397**±(**0.000032**)
4	**0.001327**±(**0.000159**)	0.001441±(0.000314)	0.001276±(0.000327)	**0.000908**±(**0.000157**)
		Estimated Volume		
2	**0.026940**±(0.021323)	0.027033±(**0.017319**)	0.026977±(**0.014578**)	**0.026968**±(0.014586)
3	0.005645±(0.006581)	**0.004053**±(**0.003013**)	0.010827±(0.010236)	**0.009600**±(**0.009362**)
4	**0.000654**±(**0.000621**)	0.000710±(0.000737)	**0.002472**±(**0.002191**)	0.002638±(0.002344)
		Number of Constraints		
2	**21.43**±(**0.62**)	**21.43**±(**0.62**)	**16.00**±(**0.00**)	**16.00**±(**0.00**)
3	**45.40**±(**0.55**)	45.57±(0.72)	**37.00**±(**0.00**)	**37.00**±(**0.00**)
4	**105.90**±(0.91)	105.97±(**0.87**)	**94.00**±(**0.00**)	**94.00**±(**0.00**)

"-" means that no randomly sampled point drops inside the function hull over-approximation.

B.1.2 Ablation Study with Different Designs of DLP Functions. Table 9 shows the ablation study of different designs of WraAct. We do not further show the ablation study of the LeakyReLU function, because the split point of the two pieces of the LeakyReLU function is the optimal choice, as shown in Table 2, which is similar to the ReLU function. We demonstrate one or two different designs, denoted as Design A and Design B, of DLP functions for Sigmoid and Tanh functions.

- For the S-shaped functions, Design A takes new $m_l' = \frac{l+m_l}{2}$ and new $m_u' = \frac{m_u+u}{2}$, where the new split point $m'l$ (or m_u') is more close to the direction to $-\infty$ (or ∞); Design B takes new $m_l' = \frac{m_l}{2}$ and new $m_u' = \frac{m_u}{2}$, where the new split point $m'l$ (or m_u') is more close to the direction to ∞ (or $-\infty$).
- For the ELU function, Design A takes new $m' = \frac{l+m}{2}$ as the split point.
- For the MaxPool function, Design A adopts a distinct grouping strategy for $\mathcal{S}_1$ and $\mathcal{S}_2$ (as discussed in Section 4.1.1), where $\mathcal{S}_1$ contains the first half of the input variables after sorting the input variable ranges, and $\mathcal{S}_2$ contains the remaining ones.

The results show that different designs of DLP functions affect the precision (volume) but have similar runtime and numbers of constraints in WraAct, showing its stable performance.

B.2 Neural Network Verification

B.2.1 Experiment Implementation Details. WraAct is implemented in Python, using Numpy [Harris et al. 2020] and pycddlib [Troffaes 2018] for over-approximating the function hull. NNVerif is implemented in PyTorch [Paszke et al. 2019] for bound propagation and Gurobi [Gurobi Optimization, LLC 2023] for linear programming. Unless otherwise specified, experiments are conducted without GPU acceleration. All reported CPU experiments are conducted on a workstation equipped

Table 10. Network architectures of benchmarks.

Network Architecture
MNIST-FCN-base (1500 neurons, 3 hidden layers):
FC(784, 500) - FC(500, 500) - FC(500, 500) - FC(500, 10)
MNIST-FCN-deep (3000 neurons, 6 hidden layers):
FC(784, 500) - FC(500, 500) - FC(500, 500) - FC(500, 500) - FC(500, 500) - FC(500, 500) - FC(500, 10)
MNIST-CNN-base (5604 neurons, 4 hidden layers):
CONV(16, 3, 4, 1, 1, 2) - CONV(32, 16, 4, 1, 1, 2) - FC(1568, 800) - FC(800, 100) - FC(100, 10)
MNIST-CNN-wide (5704 neurons, 3 hidden layers):
CONV(16, 3, 4, 1, 1, 2) - CONV(32, 16, 4, 1, 1, 2) - FC(1568, 1000) - FC(100, 10)
MNIST-CNN-pool (28004 neurons and 9 hidden layers):
CONV(3, 8, 3, 1, 0, 1) - CONV(8, 8, 3, 1, 0, 1) - MAXPOOL(2, 0, 2) - CONV(14, 32, 3, 1, 0, 1) - CONV(14, 14, 3, 1, 0, 1) - MAXPOOL(2, 0, 2) - FC(224, 50) - FC(50, 50) - FC(50, 10)
CIFAR10-FCN-base (1500 neurons and 3 hidden layers):
FC(3072, 500) - FC(500, 500) - FC(500, 500) - FC(500, 10)
CIFAR10-FCN-deep (3000 neurons and 6 hidden layers):
FC(3072, 500) - FC(500, 500) - FC(500, 500) - FC(500, 500) - FC(500, 500) - FC(500, 500) - FC(500, 10)
CIFAR10-CNN-base (7044 neurons and 4 hidden layers):
CONV(16, 3, 4, 1, 1, 2) - CONV(32, 16, 4, 1, 1, 2) - FC(1152, 100) - FC(800, 100) - FC(100, 10)
CIFAR10-CNN-wide (7144 neurons and 3 hidden layers):
CONV(16, 3, 4, 1, 1, 2) - CONV(32, 16, 4, 1, 1, 2) - FC(2048, 1000) - FC(100, 10)
CIFAR10-CNN-pool (109312 neurons and 9 hidden layers):
CONV(3, 24, 3, 1, 0, 1) - CONV(24, 24, 3, 1, 0, 1) - MAXPOOL(2, 0, 2) - CONV(24, 32, 3, 1, 0, 1) - CONV(32, 32, 3, 1, 0, 1) - MAXPOOL(2, 0, 2) - FC(800, 100) - FC(100, 100) - FC(100, 10)
CIFAR10-ResNet-base (9316 neurons and 6 hidden layers):
CONV(3, 8, 3, 1, 1, 2) - ResBlock(8, 16) - ResBlock(16, 16)- FC(1024, 100) - FC(100, 10)
CIFAR10-ResNet-deep (22116 neurons and 10 hidden layers):
CONV(3, 16, 3, 1, 1, 2) - ResBlock(16, 32) - ResBlock(32, 32) - ResBlock(32, 32) - ResBlock(32, 32) - FC(512, 100) - FC(100, 10)

† FCN and CNN refer to the fully-connected and convolutional neural network, respectively. The MaxPool neural networks use ReLU as the unary activation function. ‡ CONV(·) refers to a convolutional layer with the specific input channel, output channel, kernel size, dilation, padding, and stride by ordering. The parameters of FC(·) refer to a fully-connected layer with the specific input dimension and output dimension. MAXPOOL(·) refers to a MaxPool layer with a specific kernel size, padding, and stride. ResBlock(·) refers to a residual block with the input channels and output channels. Specifically, A ResBlock has two settings. One is of two paths, one consists of CONV(input channels, output channels, 3, 1, 0, 2) and one is CONV(input channels, output channels, 3, 1, 1, 1) - Act - CONV(output channels, output channels, 3, 1, 1, 2), where Act block is the activation layer. Another setting is of two paths, but the first path has only a direct link without CONV block.

with 20 AMD EPYC 7702P 64-Cores 2.00GHz CPUs with 100G of the main memory. GPU experiments on a workstation with 48 AMD Ryzen Threadripper PRO 5965WX 24-Cores 4.5GHz CPUs with one NVIDIA RTX A6000 GPU and 252G of the main memory.

B.2.2 Network Benchmarks. Table 10 presents the network architecture in Section 5.2.

B.2.3 Local Robustness Settings. Table 11 presents the benchmarks, including neural network and perturbation radius to verify.

Received 2025-03-26; accepted 2025-08-12

Table 11. Benchmarks of neural network verification.

Dataset	Network	Verified Perturbation Radius (ϵ)			
		Sigm.	Tanh	L. ReLU	ELU
MNIST	FCN-base	0.0200	0.0100	0.0300	0.0300
	FCN-deep	0.0300	0.0250	0.0150	0.0200
	CNN-base	0.0100	0.0100	0.0200	0.0300
	CNN-wide	0.0650	0.0400	0.0200	0.0300
CIFAR10	FCN-base	0.0040	0.0015	0.0040	0.0030
	FCN-deep	0.0030	0.0020	0.0025	0.0030
	CNN-base	0.0040	0.0015	0.0040	0.0035
	CNN-wide	0.0100	0.0025	0.0040	0.0035
	ResNet-base	-	1/255	1/255	1/255
	ResNet-deep	-	1/255	1/255	1/255
		MaxPool			
MNIST	CNN-pool	0.0100			
CIFAR10	CNN-pool	0.0010			

"-" indicates training failures or inadequate accuracy.