BERN-NN-IBF: Enhancing Neural Network Bound Propagation Through Implicit Bernstein Form and Optimized Tensor Operations

Wael Fatnassi, Arthur Feeney, Valen Yamamoto, Aparna Chandramowlishwaran, and Yasser Shoukry, *Senior Member, IEEE*

***Abstract*—Neural networks have emerged as powerful tools across various domains, exhibiting remarkable empirical performance that motivated their widespread adoption in safety-critical applications, which, in turn, necessitates rigorous formal verification techniques to ensure their reliability and robustness. Tight bound propagation plays a crucial role in the formal verification process by providing precise bounds that can be used to formulate and verify properties, such as safety, robustness, and fairness. While state-of-the-art tools use linear and convex approximations to compute upper/lower bounds for each neuron's outputs, recent advances have shown that nonlinear approximations based on Bernstein polynomials lead to tighter bounds but suffer from scalability issues. To that end, this article introduces BERN-NN-IBF, a significant enhancement of the Bernstein-polynomial-based bound propagation algorithms. BERN-NN-IBF offers three main contributions: 1) a memory-efficient encoding of Bernstein polynomials to scale the bound propagation algorithms; 2) optimized tensor operations for the new polynomial encoding to maintain the integrity of the bounds while enhancing computational efficiency; and 3) tighter under-approximations of the ReLU activation function using quadratic polynomials tailored to minimize approximation errors. Through comprehensive testing, we demonstrate that BERN-NN-IBF achieves tighter bounds and higher computational efficiency compared to the original BERN-NN and state-of-the-art methods, including linear and convex programming used within the winner of the VNN-COMPETITION.**

***Index Terms*—Formal verification, model checking, neural networks.**

I. Introduction

In RECENT years, neural networks (NNs) have emerged as indispensable tools across a myriad of applications, ranging from computer vision to natural language processing, revolutionizing fields, such as healthcare, finance, and autonomous systems. The remarkable success of NNs is attributed to their ability to learn complex patterns and representations from vast amounts of data, leading to state-of-the-art performance in various tasks. However, the increasing reliance on NNs in safety-critical domains raises significant concerns regarding their trustworthiness, robustness, and reliability. As NNs are deployed in applications where incorrect decisions can have severe consequences, there is a pressing need for rigorous verification techniques to ensure their behavior aligns with safety requirements and user expectations. The importance of NN verification has been emphasized in numerous studies [1], [2], [3] highlighting the potential risks associated with the deployment of unverified models in critical systems.

This article aims to delve into the challenges and opportunities in NN verification, focusing on the necessity of employing tight-bound propagation techniques to enhance the reliability and robustness of NN systems. In particular, state-of-the-art tools for NN verification hinge on the precise propagation of input domain bounds to the outputs of the NN. This bound propagation process is challenged by the networks' nonlinear and nonconvex nature, making exact output bound determination an NP-hard problem [4]. Previous methodologies in the literature can be classified into three categories to harness the inherent NP-hardness of the problem. The first category leaned heavily on linear relaxation of the nonlinear activation functions [5], [6], [7], [8], [9], [10], [11], [12] or reachability analysis based on linear/convex relaxation [13], [14], [15], [16], [17], [18], [19]. While linear and convex relaxations are easy to compute, these techniques result in loose bounds that deteriorate with the NN depth, which diminishes their effectiveness.

The second category detours from traditional practices by harnessing nonconvex approximations of the NN nonlinear activation function [20], [21]. In particular, the work in [20] and [21] harnesses Bernstein polynomials' power for more accurate approximations of nonlinear activations. Such nonconvex approximation was shown to lead to significant improvement in terms of approximation errors, but the quest for enhanced precision and efficiency remained ongoing.

The third category takes a design-for-verifiability approach, focusing on identifying NNs or nonlinear activation functions that allow for precise and tight analysis of NNs. Representative of this category is the work presented in [22], [23], and [24]. For example, the work in [24] explores which NN architectures lead to more scalable verification by identifying NN properties that improve verification and incentivizing these properties

Manuscript received 13 August 2024; accepted 13 August 2024. Date of current version 6 November 2024. This work was supported in part by NSF under Award CNS-2002405 and Award ECCS-2139781. This article was presented at the International Conference on Embedded Software (EMSOFT) 2024 and appeared as part of the ESWEEK-TCAD special issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Yasser Shoukry.)*

The authors are with the Department of Electrical Engineering and Computer Science, University of California at Irvine, Irvine, CA 92697 USA (e-mail: wfatnass@uci.edu; afeeney@uci.edu; vyamamot@uci.edu; amowli@uci.edu; yshoukry@uci.edu).

Digital Object Identifier 10.1109/TCAD.2024.3447577

through a verification loss. The work in [23] exploits the properties of Bernstein polynomials to design novel nonlinear activation functions that help with the computation of tight upper/lower bounds of NN's output efficiently and shift some of the computational efforts from the verification phase to the training phase.

This article introduces BERN-NN-IBF, an improved tool derived from BERN-NN [20]. BERN-NN-IBF introduces the implicit Bernstein form (IBF), a novel representation that employs 2-D tensors for bound propagation, sidestepping the computational hurdles synonymous with the n-dimensional tensors used in BERN-NN. Moreover, we have designed new operations specific to the IBF format, enhancing essential functions like summations, multiplications, and power, which are implemented efficiently using specialized CUDA routines, contributing significantly to the tool's overall speed. Finally, we reduce another source of approximation error by enhancing the coefficients of the quadratic polynomial used for the under-approximates of the rectified linear unit (ReLU) function. We achieve the optimal—in the ℓ_2 sense—approximation error by formulating and solving a dedicated optimization problem, ensuring tighter bounds on the NN's outputs. This sophisticated method surpasses the quadratic Bernstein approximation used in BERN-NN. In summary, our main contributions can be summarized as follows.

1) *Accuracy:* A novel, optimized technique for determining the coefficients of a quadratic polynomial, presenting a more refined under-approximation of the ReLU function and contributing to the precision of output bounds.
2) *Memory and Computational Efficiency:* The integration of the IBF, an enhancement in simplifying and speeding up bound representation and propagation.
3) *Implementation:* The introduction of IBF-specific operations, fine-tuning the accuracy and the efficiency of bound-related calculations.
4) The addition of a custom CUDA routine, fast-tracking the identification of minimum and maximum values within the IBF structure.

Our extensive evaluations confirm that BERN-NN-IBF not only outpaces its predecessor BERN-NN but also outperforms state-of-the-art tools like those used in the winner of the VNN-COMPETITON [25].

II. Neural Network Bound Propagation Using Bernstein Polynomials

Bound propagation in NNs refers to the process of estimating upper and lower bounds on the activations of neurons throughout a ReLU-based NN given a set of input data. These bounds provide insights into the range of possible values each neuron's activation can take, thereby enabling the characterization of the network's behavior under different conditions. In particular, in this article, we seek to encode the upper and lower bounds of neuron activations as high-order polynomials, along with techniques to propagate these polynomial-based bounds through different layers of the network. As shown in [20] and visualized in Fig. 1, higher-order polynomial approximations of ReLU functions outperform state-of-the-art approaches of using linear approximations (e.g., triangulation, crown, and zonotopes) in terms of approximation errors. Albeit promising, the propagation of bounds encoded as higher-order polynomials across different NN layers is computationally challenging compared to linear approximations. This section reviews the basics of higher-order polynomial approximation of the ReLU functions and discusses their challenges.

A. Notation

We use the symbols $\mathbb{N}$ and $\mathbb{R}$ to denote the set of natural and real numbers, respectively. We denote by $x = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ the vector of n real-valued variables, where $x_i \in \mathbb{R}$. We denote by $I_n(\underline{d}, \overline{d}) = [\underline{d}_1, \overline{d}_1] \times \cdots \times [\underline{d}_n, \overline{d}_n] \subset \mathbb{R}^n$ the n-dimensional hyperrectangle, where $\underline{d} = (\underline{d}_1, \ldots, \underline{d}_n)$ and $\overline{d} = (\overline{d}_1, \ldots, \overline{d}_n)$ are the lower and upper bounds of the hyperrectangle, respectively. For a real-valued vector $x = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ and an index-vector $K = (k_1, \ldots, k_n) \in \mathbb{N}^n$, we denote by $x^K \in \mathbb{R}$ the scalar $x^K = x_1^{k_1} \times \ldots \times x_n^{k_n}$. Given two multi-indices $K = (k_1, \ldots, k_n) \in \mathbb{N}^n$ and $L = (l_1, \ldots, l_n) \in \mathbb{N}^n$, we use the following notation throughout this article:

$$K + L = (k_1 + l_1, \ldots, k_n + l_n)$$
$$\binom{L}{K} = \binom{l_1}{k_1} \times \cdots \times \binom{l_n}{k_n}$$
$$\sum_{K \le L} = \sum_{k_1 \le l_1}, \cdots \sum_{k_n \le l_n}.$$

Finally, a real-valued multivariate polynomial $p : \mathbb{R}^n \to \mathbb{R}$ is defined as

$$p(x_1, \ldots, x_n) = \sum_{k_1=0}^{l_1} \sum_{k_2=0}^{l_2} \cdots \sum_{k_n=0}^{l_n} a_{(k_1,\ldots,k_n)} x_1^{k_1} x_2^{k_2} \ldots x_n^{k_n} = \sum_{K \le L} a_K x^K$$

where $L = (l_1, l_2, \ldots, l_n)$ is the maximum degree of x_i for all $i = 1, \ldots, n$. While the multi-index L depends on the input dimension n and the polynomial degree, for simplicity of notation, we drop such dependencies.

B. Bernstein Polynomials as Multidimensional Tensors

In this article, we rely on a class of polynomials called Bernstein polynomials, which are defined as follows.

Definition 1 (Bernstein Polynomials): Given a continuous function $p : \mathbb{R}^n \to \mathbb{R}$, an input domain (hypercube) $I_n(\underline{d}, \overline{d}) \subset \mathbb{R}^n$, and a multi-index $L = (l_1, \ldots, l_n) \in \mathbb{N}^n$, the polynomial

$$B_{p,L}(x) = \sum_{K \le L} b^p_{K,L} Ber_{K,L}(x)$$
$$Ber_{K,L}(x) = \binom{L}{K} \frac{(x - \underline{d})^K (\overline{d} - x)^{L-K}}{(\overline{d} - \underline{d})^L}$$
$$b^p_{K,L} = p\left((\overline{d}_1 - \underline{d}_1)\frac{k_1}{l_1} + \underline{d}_1, \ldots, (\overline{d}_n - \underline{d}_n)\frac{k_n}{l_n} + \underline{d}_n \right)$$

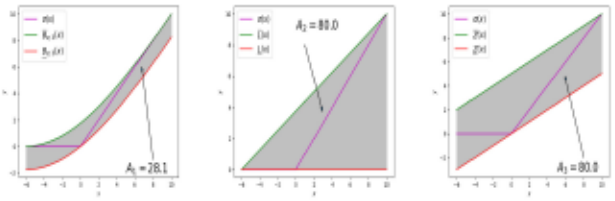

Fig. 1. Illustrations of the over/under-approximation of the ReLU activation functions in the interval $[-6, 10]$ using different approaches: (left) higher-order polynomials, (center) triangulation, and (right) zonotope. The area of the shaded set A_1, A_2, A_3 represents the approximation error for each of the approaches [20].

is called the Lth-order Bernstein polynomial of p, where $Ber_{K,L}(x)$ and $b^p_{K,L}$ are called the Bernstein basis and Bernstein coefficients of p, respectively.

Bernstein polynomials are particularly noted for their capacity to approximate any continuous function on a closed interval. This property is crucial when dealing with functions that are not differentiable, as is the case with certain activation functions in NNs, such as the ReLU function.

Given a Bernstein polynomial of order L, one can represent it as a *dense* multidimensional tensor $\texttt{Den}(B_{p,L})$ of n dimensions, and of a shape of $L = (l_1 + 1, \ldots, l_n + 1)$, where the $K = (k_1, \ldots, k_n)$ component of $\texttt{Den}(B_{p,L})$ is equal to the Bernstein coefficient $b^p_{K,L}$. The multidimensional tensor $\texttt{Den}(B_{p,L})$ represents all the coefficients $b^p_{K,L}$ $\forall K \leq L$.

Example 1: Consider the 2-D Bernstein polynomial

$$B_{p,L}(x_1, x_2) = \sum_{k_1=0}^{2} \sum_{k_2=0}^{3} b^p_{(k_1,k_2),L} Ber_{(k_1,k_2),L}(x_1, x_2)$$

with orders $L = (2, 3)$. Its 2-D tensor representation is written as follows:

$$\texttt{Den}(B_{p,L}) = \begin{bmatrix} b^p_{(0,0),L} & b^p_{(0,1),L} & b^p_{(0,2),L} & b^p_{(0,3),L} \\ b^p_{(1,0),L} & b^p_{(1,1),L} & b^p_{(1,2),L} & b^p_{(1,3),L} \\ b^p_{(2,0),L} & b^p_{(2,1),L} & b^p_{(2,2),L} & b^p_{(2,3),L} \end{bmatrix}. \quad (1)$$

C. Polynomial-Based Interval Bound Propagation Using Bernstein Polynomials

The BERN-NN framework [20] employs Bernstein polynomials to propagate interval bounds through NNs accurately and can be summarized as follows. *Step 1 (Encode Input Domains as Bernstein Polynomials):* Given the input domain $I_n(\overline{d}, \underline{d})$ and a trained NN $NN(x)$, the first step is to represent the upper/lower bounds of the input domain $\underline{d} = (\underline{d}_1, \ldots, \underline{d}_n)$ and $\overline{d} = (\overline{d}_1, \ldots, \overline{d}_n)$ as zero-order polynomials of the form: $\overline{NN}_i^{(0)}(x) = \overline{d}_i, \underline{NN}_i^{(0)}(x) = \underline{d}_i$, where the superscript (0) denotes the zeroth layer (i.e., input layer) of the NN. We encode the Bernstein representation of these polynomials as

$$\texttt{Den}\left(B_{\underline{NN}^{(0)},L^{(0)}}\right) = \left[\texttt{Den}\left(B_{\underline{NN}_1^{(0)},L^{(0)}}\right), \ldots, \texttt{Den}\left(B_{\underline{NN}_n^{(0)},L^{(0)}}\right)\right]$$
$$\texttt{Den}\left(B_{\overline{NN}^{(0)},L^{(0)}}\right) = \left[\texttt{Den}\left(B_{\overline{NN}_1^{(0)},L^{(0)}}\right), \ldots, \texttt{Den}\left(B_{\overline{NN}_n^{(0)},L^{(0)}}\right)\right]$$

where $L^{(0)} = (0, \ldots, 0)$.

Step 2 (Propagate the Bernstein Polynomials Through Linear Weights): Given the weights and biases of the ith layer of an NN $(W^{(i)}, b^{(i)})$ and the n-dimensional tensors representing the bounds of the $i-1$ layer $\texttt{Den}(B_{\underline{NN}^{(i-1)},L^{(i-1)}})$ and $\texttt{Den}(B_{\overline{NN}^{(i-1)},L^{(i-1)}})$, we propagate them through the linear weights of the ith layer using linear interval arithmetic

$$\texttt{Den}\left(B_{\underline{NN}^{*(i)},L^{(i-1)}}\right) = \texttt{Den}\left(B_{\underline{NN}^{(i-1)},L^{(i-1)}}\right) * W_+^{(i)} + \texttt{Den}\left(B_{\overline{NN}^{(i-1)},L^{(i-1)}}\right) * W_-^{(i)} + b^{(i)}$$
$$\texttt{Den}\left(B_{\overline{NN}^{*(i)},L^{(i-1)}}\right) = \texttt{Den}\left(B_{\overline{NN}^{(i-1)},L^{(i-1)}}\right) * W_+^{(i)} + \texttt{Den}\left(B_{\underline{NN}^{(i-1)},L^{(i-1)}}\right) * W_-^{(i)} + b^{(i)}$$
$$W_+^{(i)} = \max\left(W^{(i)}, 0_{i\times(i-1)}\right), W_-^{(i)} = \min\left(W^{(i)}, 0_{i\times(i-1)}\right)$$

where $W_+^{(i)}$ and $W_-^{(i)}$ denote the set of positive and negative weights between the $(i-1)$th layer to the ith layer and $0_{i\times(i-1)}$ denotes the zero matrix of dimension $i \times (i-1)$.

Step 3 (Propagate the Bernstein Polynomials Through the Nonlinear Activations): Next, BERN-NN over/under-approximate the nonlinear ReLU activation function $\sigma(x) = \max(x, 0)$ using Bernstein polynomials $B_{\overline{\sigma},L_\sigma}$ and $B_{\underline{\sigma},L_\sigma}$ with a user-defined approximation order L_σ (see the left subfigure in Fig. 1). Next, it composes the polynomials as

$$\texttt{Den}\left(B_{\underline{NN}^{(i)},L^{(i)}}\right) = \overline{B}_{\underline{\sigma},L_\sigma}\left(\texttt{Den}\left(B_{\underline{NN}^{*(i)},L^{(i-1)}}\right)\right) \quad (2)$$
$$\texttt{Den}\left(B_{\overline{NN}^{(i)},L^{(i)}}\right) = \overline{B}_{\overline{\sigma},L_\sigma}\left(\texttt{Den}\left(B_{\overline{NN}^{*(i)},L^{(i-1)}}\right)\right). \quad (3)$$

Step 4: Repeat steps 2 and 3 until the bounds are propagated through all the NN layers. The final step is to compute the maximum of $\texttt{Den}(B_{\overline{NN}^{(i)},L^{(i)}})$ and the minimum of $\texttt{Den}(B_{\underline{NN}^{(i)},L^{(i)}})$ for all layers i, which will produce the upper/lower bounds for all neuron outputs in the NN.

Unfortunately, implementing the interval bound propagation (steps 1–4) cannot be done using standard tensor operations due to the mathematical definition of Bernstein polynomials. That is why BERN-NN [20] implements special procedures called 1) Bernstein summation of n-dimensional tensors (**Sum_Bern**); 2) Bernstein multiplication of n-dimensional tensors (**Prod_Bern**); and 3) computing the extreme (minimum and maximum) points of n-dimensional tensors (**Min/Max_Bern**).

D. Memory and Computational Complexity of BERN-NN

It is crucial to recall the memory and computational complexity of the algorithms used in BERN-NN.

Proposition 1 [20]: The memory and computational complexity of **Sum_Bern**, **Prod_Bern**, and **Min/Max_Bern** are $O((l_{\max})^n)$, where $l_{\max} = \max_{1\leq i\leq n} l_i$ is the maximum order of the polynomials obtained during the bound propagation. Moreover, $l_{\max} = O(2^k)$ with k the number of NN layers.

In other words, the algorithms needed by BERN-NN scale exponentially with respect to the NN's input dimension n and the NN depth, which makes it hard to use for deep NNs with large input dimensions.

III. Enhanced Accuracy Using Optimal Under-Approximation of ReLU Functions

In this section, we present our first contribution to enhancing the Bernstein-polynomial interval bound propagation accuracy

described in Section II-C. In particular, step 3 of the BERN-NN algorithm [20] utilizes a downward translation of the Bernstein over-approximation of the ReLU σ activation function to achieve its under-approximation (see the green curves in Fig. 2). While the over-approximation is optimal [20]—i.e., produces the tightest over-approximation of the ReLU function in the ℓ_2 sense—this under-approximation may not always be optimal, especially when the negative side of the preinput bounds significantly outweighs the positive side. To address this issue, we confine our attention to the use of quadratic polynomials, and we formulate the "optimal ReLU under-approximation problem" as an optimization problem defined as follows:

$$\begin{aligned} \underset{a,b,c}{\text{minimize}} \;\; & A(a,b,c) = \int_{\underline{d}}^{\overline{d}} \Big(\sigma(x) - \big(ax^2+bx+c\big)\Big)dx \\ \text{subject to} \;\; & ax^2+bx+c \le \sigma(x) \\ & x \in [\underline{d}, \overline{d}] \end{aligned} \tag{4}$$

where $\sigma(x)$ is a ReLU function defined on an interval $[\underline{d}, \overline{d}]$. The optimization problem presented above addresses the problem of finding the coefficients a, b, and c for a quadratic polynomial $q(x) = ax^2 + bx + c$, which provides a tight under-approximation of a given ReLU $\sigma(x)$. The goal is to minimize the area $A(a, b, c)$ between the ReLU curve and the under-approximation curve over the interval $[\underline{d}, \overline{d}]$, where $\underline{d}$ and $\overline{d}$ are the lower and upper bounds of the ReLU's domain, respectively. By solving this optimization problem, we can obtain a quadratic under-approximation of the ReLU function that accurately captures its behavior over the specified domain.

To ensure the soundness of the quadratic polynomial under-approximation over the entire interval $[\underline{d}, \overline{d}]$, the optimization problem in (4) includes the constraint $q(x) = ax^2 + bx + c \le \sigma(x)\ \forall x \in [\underline{d}, \overline{d}]$. This constraint guarantees that the under-approximation curve always lies below the ReLU curve. This quadratic constraint depends on the variable x, making it harder to solve the optimization problem. In the following proposition, we rewrite this constraint and remove this dependency so that it depends only on the polynomial coefficients a, b, and c.

Proposition 2: Assume that $\underline{d}$ is negative and $\overline{d}$ is positive, i.e., $\underline{d} < 0 < \overline{d}$. The following conditions are sufficient to ensure that the quadratic polynomial $q(x) = ax^2 + bx + c$ is an under approximation of the ReLU function—i.e., $q(x) \le \sigma(x)$:

$$\begin{aligned} & q(\underline{d}) = a\underline{d}^2 + b\underline{d} + c \le 0, \qquad q(0) = c \le 0 \\ & q(\overline{d}) = a\overline{d}^2 + b\overline{d} + c \le \overline{d}, \qquad 0 \le a. \end{aligned} \tag{5}$$

Proof: Our goal is to show that $q(x) \le \sigma(x)$ for all $x \in [\underline{d}, \overline{d}]$, subject to the given conditions. We divide the interval $[\underline{d}, \overline{d}]$ into two cases.

Case 1: For $x \in [\underline{d}, 0]$, we have $ax^2 \le a\underline{d}^2$. If $b \ge 0$, then $q(x) = ax^2 + bx + c \le q(\underline{d}) = a\underline{d}^2 + c \le a\underline{d}^2 + b\underline{d} + c \le 0$. If $b \le 0$, then $q(x) = ax^2 + bx + c \le q(\underline{d}) = a\underline{d}^2 + b\underline{d} + c \le 0$. In either case, we have $q(x) \le \sigma(x)$ for all $x \in [\underline{d}, 0]$.

Case 2: For $x \in [0, \overline{d}]$, we define $l(x) = m_1 x + m_2$, where $m_1 = [q(\overline{d}) - q(0)/\overline{d}]$ and $m_2 = q(0)$. It is clear that $l(0) = q(0)$ and $l(\overline{d}) = q(\overline{d})$. Furthermore, the quadratic polynomial

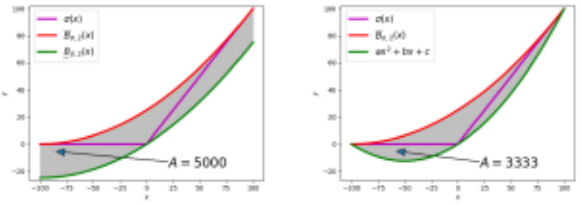

Fig. 2. (Left) State-of-the-art over- and under-approximations of ReLU functions $\sigma(x)$ using high-order polynomials. (Right) Proposed optimal over- and under-approximation of ReLU functions $\sigma(x)$. The figure shows the area between the two curves A, indicating the approximation error.

q is convex because $a \ge 0$. Therefore, $q(x) \le l(x)$ for all $x \in [0, \overline{d}]$. To complete the proof, we must show that $l(x) \le x$ for all $x \in [0, \overline{d}]$.

We define $l_{\text{diff}}(x) = l(x) - x = ([q(\overline{d}) - q(0) - \overline{d}]/\overline{d})x + q(0)$. Then, we have $l_{\text{diff}}(0) = q(0) \le 0$ and $l_{\text{diff}}(\overline{d}) = q(\overline{d}) - \overline{d} \le 0$. Since l_{diff} is a line defined over $[0, \overline{d}]$ and $l_{\text{diff}}(0) \le 0$ and $l_{\text{diff}}(\overline{d}) \le 0$, we conclude that $l_{\text{diff}}(x) \le 0$ for all $x \in [0, \overline{d}]$. Therefore, $l(x) \le x$ for all $x \in [0, \overline{d}]$. Consequently, we have $q(x) \le \sigma(x)$ for all $x \in [0, \overline{d}]$. ∎

Remark 1: It is crucial to note that the polynomial under-approximation is needed only whenever $\underline{d}$ is negative and $\overline{d}$ is positive, i.e., $\underline{d} < 0 < \overline{d}$. In the cases when $\underline{d}$ is positive or $\overline{d}$ is negative, the ReLU function $\sigma(x)$ is linear (either $\sigma(x) = 0$ or $\sigma(x) = x$) and polynomial approximation is not needed. Hence, the proof of Proposition 2 focuses only on the case when the condition $\underline{d} < 0 < \overline{d}$ is satisfied.

Proposition 2 enables us to transform the optimization problem's nonlinear constraint into four linear constraints. We reformulate the optimization problem (4) as

$$\begin{aligned} \underset{a,b,c}{\text{minimize}} \;\; & A(a,b,c) = \int_{\underline{d}}^{\overline{d}} \Big(\sigma(x) - \big(ax^2+bx+c\big)\Big)dx \\ \text{subject to} \;\; & a\underline{d}^2 + b\underline{d} + c \le 0, \qquad c \le 0, \\ & a\overline{d}^2 + b\overline{d} + c \le \overline{d}, \qquad 0 \le a. \end{aligned} \tag{6}$$

The objective function $A(x)$ of the optimization problem can be expressed in the following form:

$$\begin{aligned} A(a,b,c) &= \int_{\underline{d}}^{\overline{d}} \Big(\sigma(x) - \big(ax^2+bx+c\big)\Big) \\ &= \int_{\underline{d}}^{\overline{d}} \sigma(x)dx - \int_{\underline{d}}^{\overline{d}} \big(ax^2+bx+c\big)dx \\ &= \frac{\overline{d}}{2} - \left(\frac{a\left(\overline{d}^3 - \underline{d}^3\right)}{3} + \frac{b\left(\overline{d}^2 - \underline{d}^2\right)}{2} + c\left(\overline{d} - \underline{d}\right)\right). \end{aligned} \tag{7}$$

By examining (7), it becomes apparent that the objective function is linear with respect to the coefficients a, b, and c. Hence, we can transform the optimization problem in (4) into a linear programming (LP) problem

$$\begin{aligned} \underset{a,b,c}{\text{maximize}} \;\; & a\left(\frac{\overline{d}^3 - \underline{d}^3}{3}\right) + b\left(\frac{\overline{d}^2 - \underline{d}^2}{2}\right) + c\left(\overline{d} - \underline{d}\right) \\ \text{subject to} \;\; & a\underline{d}^2 + b\underline{d} + c \le 0, \qquad c \le 0, \\ & a\overline{d}^2 + b\overline{d} + c \le \overline{d}, \qquad 0 \le a. \end{aligned} \tag{8}$$

We now focus on optimizing the problem further. A critical observation based on the inequalities $([\overline{d}^3 - \underline{d}^3]/3) \geq 0$, $([\overline{d}^2 - \underline{d}^2]/2) \geq 0$, $\overline{d} - \underline{d} \geq 0$, and $c \leq 0$, allows us to consider the following inequality:

$\forall a, b,$ and $c \in \mathbb{R}$:

$$a\left(\frac{\overline{d}^3 - \underline{d}^3}{3}\right) + b\left(\frac{\overline{d}^2 - \underline{d}^2}{2}\right) + c(\overline{d} - \underline{d}) \leq a\left(\frac{\overline{d}^3 - \underline{d}^3}{3}\right) + b\left(\frac{\overline{d}^2 - \underline{d}^2}{2}\right).$$

The right side of the inequality is attained when $c = 0$. This adjustment simplifies our linear program (LP) by effectively reducing the number of variables, thereby transforming it into a more manageable form. The revised LP, with $c = 0$, is

$$\begin{aligned} \underset{a,b}{\text{maximize}} \quad & a\left(\frac{\overline{d}^3 - \underline{d}^3}{3}\right) + b\left(\frac{\overline{d}^2 - \underline{d}^2}{2}\right) \\ \text{subject to} \quad & a\underline{d}^2 + b\underline{d} \leq 0 \\ & a\overline{d}^2 + b\overline{d} \leq \overline{d}, \qquad 0 \leq a. \end{aligned} \tag{9}$$

This step is crucial as it reduces the LP's dimensionality from three to two, making the problem less complex and more approachable. The number of constraints also drops from four to three, further simplifying the solution process. These reductions are strategic, streamlining the problem without sacrificing the integrity of the solution space.

Finally, we concentrate on solving the LP problem delineated in (9). For simplification, let us define the function $f(a, b)$ representing our objective

$$f(a, b) = a\left(\frac{\overline{d}^3 - \underline{d}^3}{3}\right) + b\left(\frac{\overline{d}^2 - \underline{d}^2}{2}\right).$$

The problem's feasible region is notably a triangle, defined by three vertices: $v_1 = (0, 0)$, $v_2 = ([1/\overline{d} - \underline{d}], [-\underline{d}/(\overline{d} - \underline{d})])$, and $v_3 = (0, 1)$. Our goal is to identify the maximum value of $f(a, b)$ at these specific points, which correspond to potential solutions of the LP. To that end, we proceed by evaluating f at each vertex and comparing these values. The optimal solution is determined based on which vertex (v_1, v_2, v_3) yields the maximum value as explained in Algorithm 1.

This approach simplifies the solution process by reducing the problem to comparisons of function values at specific points rather than requiring a more extensive search through a continuous space. The solutions adapt based on the vertex yielding the highest function value, guiding the parameters a, b, and c accordingly. Fig. 2 vividly illustrates that the approximation error area from the quadratic polynomial is significantly smaller than that of the original under-approximation [20].

Algorithm 1 Quadratic Coefficients for ReLU Under-Approximation

Input: $I = [\underline{d}, \overline{d}]$
Output: Coefficients a, b, c for the quadratic polynomial under-approximating ReLU

function get_quad_coeffs_under($I = [\underline{d}, \overline{d}]$) **do**
  $f_1 \leftarrow 0$
  $f_2 \leftarrow \frac{2\times\overline{d}^2 - \overline{d}\times\underline{d} - \underline{d}^2}{6.0}$
  $f_3 \leftarrow \frac{\overline{d}^2 - \underline{d}^2}{2.0}$
  if $\max(f_1, f_2, f_3) == f_1$ **then**
    return $a = b = c = 0$
  else if $\max(f_1, f_2, f_3) == f_2$ **then**
    return $a = \frac{1}{\overline{d} - \underline{d}}$, $b = \frac{-\underline{d}}{\overline{d} - \underline{d}}$, $c = 0$
  else
    return $a = 0$, $b = 1$, $c = 0$
  end if
end function

IV. Memory Efficiency Using Implicit Representation Of Bernstein Polynomials

In this section, we present our second contribution to reduce the memory requirements of the dense tensor representation used for bound propagation. Recall that Proposition 1 shows that the memory requirements of the n dimensional tensors grow exponentially with the input dimensions n and the NN depth l. Our approach hinges on the so-called "implicit representation" of Bernstein polynomials. We propose a novel tensor representation of Bernstein polynomials that leads to better memory and computational efficiency, which is particularly beneficial for high-dimensional scenarios.

A. 2-D Tensor Encoding of High-Dimensional Polynomials

Given a multivariate polynomial p of order $L = (l_1, \ldots, l_n)$, that consists of t terms, as follows:

$$p(x_1, \ldots, x_n) = \sum_{K \leq L} a_K x^K \tag{10}$$

where $K \in \{K_1, \ldots, K_t\}$, and $0 \leq K_j \leq L \; \forall j \in \{1, \ldots, t\}$. Now, the polynomial p consists of t terms: $a_{K_j} x^{K_j} = a_{K_j} x_1^{k_j^1} \cdots x_n^{k_j^n}$, where $K_j = (k_j^1, \ldots, k_j^n)$. Let us denote by $\text{term}(K_j) = a_{K_j} x_1^{k_j^1} \cdots x_n^{k_j^n}$, the jth term of p. Let us denote by $\text{var}(k_j^i) = x_i^{k_j^i}$, $1 \leq i \leq n$, the ith variable in the jth term. We represent all the Bernstein coefficients for $\text{var}(k_j^i)$ as $\texttt{Imp}(B_{\text{var}(k_j^i), l_i})$ which is shown as follows:

$$\begin{aligned} \text{var}\left(k_j^i\right) = x_i^{k_j^i} \xrightarrow{\text{is encoded as}} \\ \texttt{Imp}\left(B_{\text{var}\left(k_j^i\right), l_i}\right) = \left[b_{0,l_i}^{\text{var}\left(k_j^i\right)}, \ldots, b_{l_i,l_i}^{\text{var}\left(k_j^i\right)}\right]. \end{aligned} \tag{11}$$

We call $\texttt{Imp}(B_{\text{var}(k_j^i), l_i})$ in (11) the implicit form representation (IBF) of one single variable $\text{var}(k_j^i)$ which is the Bernstein coefficients for that variable. Because the order of this $\text{var}(k_j^i)$ is l_i, we can have up to $l_i + 1$ Bernstein coefficients.

Now, computing all the Bernstein coefficients of the jth term, $\text{term}(K_j)$, is equal to the Cartesian product of the Bernstein coefficients of every single variable $\texttt{Imp}(B_{\text{var}(k_j^i), l_i})$ and multiply the resultant multidimensional tensor by the coefficient a_{K_j}. However, this process is not memory efficient

because the Cartesian product will result in a multidimensional tensor, which is the drawback of the dense representation. Instead, we compute the IBF of the jth term, term(K_j), by stacking the IBF of every single variable $\texttt{Imp}(B_{\text{var}(k_j^i),l_i})$ row-wise. After that, we multiply the coefficient a_{K_j} by just the first row. All this is summarized in the following equation:

$$\text{term}(K_j) = a_{K_j}x^{K_j} \xrightarrow{\text{is encoded as}} \texttt{Imp}\left(B_{\text{term}(K_j),L}\right) = \begin{bmatrix} a_{K_j}\texttt{Imp}\left(B_{\text{var}\left(k_j^1\right),l_1}\right) \\ \vdots \\ \texttt{Imp}\left(B_{\text{var}\left(k_j^n\right),l_n}\right) \end{bmatrix}. \quad (12)$$

The length of the ith row in $\texttt{Imp}(B_{\text{term}(K_j),L})$ is equal to $l_i + 1$, $1 \le i \le n$. We denote by $l_{\max} = \max_{1\le i\le n} l_i$. The size of the IBF of the jth term, term(K_j), $\texttt{Imp}(B_{\text{term}(K_j),L})$, is equal to $n \times (l_{\max} + 1)$, where we pad the rows of lengths $l_i < l_{\max}$ with $l_{\max} - l_i + 1$ zeros at the right side.

Now, the total Bernstein coefficients of the whole polynomial p is the summation of Bernstein coefficients of every term term(K_j), $1 \le j \le t$. This translates to the IBF of the whole polynomial p is by stacking the IBF of every term as follows:

$$p(x_1, \ldots, x_n) = \sum_{K\le L} a_K x^K \xrightarrow{\text{is encoded as}} \texttt{Imp}\left(B_{p,L}\right) = \begin{bmatrix} \texttt{Imp}\left(B_{\text{term}(K_1),L}\right) \\ \vdots \\ \texttt{Imp}\left(B_{\text{term}(K_n),L}\right) \end{bmatrix}. \quad (13)$$

Now, the total size of $\texttt{Imp}(B_{p,L})$ is equal to $nt \times (l_{\max} + 1)$. Below is an illustrative example.

Example 2: Consider the multivariate polynomial $p(x_1, x_2) = x_1^3x_2^2 - 30x_1x_2$, defined over the domain $I_2 = [1, 2] \times [2, 4]$, with $L = (3, 2)$. The IBF of x_1^3, x_2^2, x_1, and x_2 is as follows:

$$\begin{aligned} \texttt{Imp}\left(B_{\text{var}(k_1^1),3}\right) &= \left[1, 2, 4, 8\right] \\ \texttt{Imp}\left(B_{\text{var}(k_1^2),2}\right) &= \left[4, 8, 16\right] \\ \texttt{Imp}\left(B_{\text{var}(k_2^1),3}\right) &= \left[-1, -4/3, -5/3, -2\right] \\ \texttt{Imp}\left(B_{\text{var}(k_2^2),2}\right) &= \left[2, 3, 4\right]. \end{aligned} \quad (14)$$

Using (12) and (13), the IBF of the polynomial p is written as follows:

$$\texttt{Imp}\left(B_{p,L}\right) = \begin{bmatrix} 1 & 2 & 4 & 8 \\ 4 & 8 & 16 & 0 \\ -30 & -40 & -50 & -60 \\ 2 & 3 & 4 & 0 \end{bmatrix}. \quad (15)$$

This example illustrates that if a polynomial p comprises t terms, and each term is represented by n rows, the total number of rows in the implicit representation amounts to nt. The length of each row is given by $l_{\max}+1 = \max_{1\le i\le n}(l_i)+1$. Therefore, the overall size of the implicit representation matrix is determined by the dimensions $nt \times (l_{\max} + 1)$.

B. Efficient Multiplication of Implicit Bernstein Polynomials

Given the memory-efficient encoding in the previous section, we focus on how to implement an efficient procedure to compute the product of two Bernstein polynomials encoded using the 2-D tensors discussed above.

1) Monomial Bernstein Polynomial Multiplication: We focus initially on polynomials comprising a single term. Consider two monomials $q_1(x) = a_1x^{k_1}$ and $q_2(x) = a_2x^{k_2}$, with $k_1 \le L_1$ and $k_2 \le L_2$. The implicit representations of their Bernstein polynomials, $\texttt{Imp}(B_{q_1,L_1})$ and $\texttt{Imp}(B_{q_2,L_2})$. The implicit representation of the product of these polynomials, $\texttt{Imp}(B_{q_1,L}B_{q_2,L})$, is computed using the following procedure:

$$\texttt{Imp}\left(\tilde{B}_{q_1,L_1}\right) = \texttt{Imp}\left(B_{q_1,L_1}\right) * C_{L_1} \quad (16)$$

$$\texttt{Imp}\left(\tilde{B}_{q_2,L_2}\right) = \texttt{Imp}\left(B_{q_2,L_2}\right) * C_{L_2} \quad (17)$$

$$\texttt{Imp}\left(B_{q_1,L_1}B_{q_2,L_2}\right) = \frac{1}{C_{L_1+L_2}} * \mathbf{Conv}\left(\texttt{Imp}\left(\tilde{B}_{q_1,L_1}\right), \texttt{Imp}\left(\tilde{B}_{q_2,L_2}\right)\right) \quad (18)$$

where C_L denotes the multidimensional binomial tensor, with its Kth component in the ith row defined as $(C_L)^i_K = \binom{l_i}{K}$. With some abuse of notation, we use $1/C_{l_i}$ to denote the multidimensional binomial tensor where its Kth component in the ith row is equal to $(1/\binom{l_i}{K})$. The notation $*$ represents element-wise multiplication, while $\mathbf{Conv}(A, B)$ denotes the row-wise convolution between matrices A and B. The above formulation efficiently generalizes the concept of scaled Bernstein polynomials [26] to n-dimensional inputs. Efficiently implementing these operations on GPUs—by leveraging element-wise and convolution operations—ensures high computational performance. We denote the process in (16)–(18) as $\mathbf{Prod\_Bern\_Imp}(B_{p_1,L_1}, B_{p_2,L_2})$.

2) Multivariate Bernstein Polynomial Multiplication: The method can be extended to handle the multiplication of implicit representations of Bernstein polynomials consisting of multiple terms. Consider two polynomials, $p_1 = \sum_{K\le L_1} a_K^1 x^K$ and $p_2 = \sum_{K\le L_2} a_K^2 x^K$, with t_1 and t_2 terms, respectively. Their implicit representations are denoted as $\texttt{Imp}(B_{p_1,L_1})$ and $\texttt{Imp}(B_{p_2,L_2})$. The multiplication of these polynomials in their IBF is given by

$$\begin{aligned} &\texttt{Imp}\left(B_{p_1,L_1}B_{p_2,L_2}\right) \\ &= \begin{bmatrix} \mathbf{Prod\_Bern\_Impl}\left(\texttt{Imp}^1\left(B_{p_1,L_1}\right), \texttt{Imp}^1\left(B_{p_2,L_2}\right)\right) \\ \vdots \\ \mathbf{Prod\_Bern\_Impl}\left(\texttt{Imp}^i\left(B_{p_1,L_1}\right), \texttt{Imp}^j\left(B_{p_2,L_2}\right)\right) \\ \vdots \\ \mathbf{Prod\_Bern\_Impl}\left(\texttt{Imp}^{t_1}\left(B_{p_1,L_1}\right), \texttt{Imp}^{t_2}\left(B_{p_2,L_2}\right)\right) \end{bmatrix} \\ &1 \le i \le t_1, 1 \le j \le t_2. \end{aligned} \quad (19)$$

Here, $\texttt{Imp}^i(B_{p_1,L_1})$ and $\texttt{Imp}^j(B_{p_2,L_2})$ represent the ith and jth submatrices (terms) in $\texttt{Imp}(B_{p_1,L_1})$ and $\texttt{Imp}(B_{p_2,L_2})$, respectively. These submatrices are obtained by segmenting the original implicit representations into t_1 and t_2 submatrices along their rows. This formulation reveals that multiplying multivariate polynomials in IBF effectively boils down to multiplying terms from one polynomial with those from another, which can be parallelized efficiently using GPUs.

C. Efficient Summation of Implicit Bernstein Polynomials

In this section, we exploit the 2-D tensor encoding of Bernstein polynomials to implement effective procedures to add two Bernstein polynomials.

1) Monomial Bernstein Polynomial Summation: Building upon the foundational work in [27], which addresses the addition of 1-D Bernstein polynomials over the unit interval, this section introduces an advanced and generalized approach for the summation of implicit Bernstein polynomials in a multivariate framework. Our extension caters to scenarios with n variables over any specified interval $I_n(\underline{d}, \overline{d})$. Initially, we focus on monomials of the form $q_1(x) = a_1 x^{k_1}$ and $q_2(x) = a_2 x^{k_2}$, where $k_1 \leq L_1$ and $k_2 \leq L_2$. Their respective implicit Bernstein polynomial representations, $\texttt{Imp}(B_{q_1,L_1})$ and $\texttt{Imp}(B_{q_2,L_2})$, are conceived as $n \times N$ blocks. The summation of these polynomials in their implicit form, $\texttt{Imp}(B_{q_1+q_2,L_{\text{sum}}})$, is delineated through the following proposition.

Proposition 3: Given two Bernstein polynomials $B_{q_1,L_1}(x)$ and $B_{q_2,L_2}(x)$ with two different orders $L_1 = (l_1^1, \ldots, l_n^1)$ and $L_2 = (l_1^2, \ldots, l_n^2)$, let $L_{\text{sum}} = \max(L_1, L_2)$, where the max operator is applied element-wise. The implicit tensor representation of $B_{q_1+q_2,L_{\text{sum}}}$ can be computed as

$$L_{\text{sum}} = \left(\max(l_1^1, l_1^2), \ldots, \max(l_n^1, l_n^2)\right) \tag{20}$$

$$\texttt{Imp}\left(B_{q_1,L_{\text{sum}}}\right) = \textbf{Prod\_Bern\_Imp}\left(\texttt{Imp}\left(B_{q_1,L_1}\right), 1_{L_{\text{sum}}-L_1+1}\right) \tag{21}$$

$$\texttt{Imp}\left(B_{q_2,L_{\text{sum}}}\right) = \textbf{Prod\_Bern\_Imp}\left(\texttt{Imp}\left(B_{q_2,L_2}\right), 1_{L_{\text{sum}}-L_2+1}\right) \tag{22}$$

$$\texttt{Imp}\left(B_{q_1+q_2,L_{sum}}\right) = \begin{bmatrix} \texttt{Imp}\left(B_{q_1,L_{\text{sum}}}\right) \\ \texttt{Imp}\left(B_{q_2,L_{\text{sum}}}\right) \\ \cdot \end{bmatrix} \tag{23}$$

Here, 1_{L_e-L+1} signifies a 2-D tensor with dimensions $n \times (L_e - L + 1)$, exclusively containing ones.

The proof, which extends the argument in [27], is not presented for brevity. In this context, operations (21) and (22) are recognized as *degree elevation* processes, where tensor dimensions are suitably altered. The final summation, defined by (23), is executed by vertically concatenating both tensors once they align dimensionally. We denote this procedure by **Sum_Bern_Imp**.

2) Multivariate Bernstein Polynomial Summation: We further extend the method to accommodate summation of implicit Bernstein polynomial representations containing multiple terms. Consider polynomials $p_1 = \sum_{K \leq L_1} a_K^1 x^K$ and $p_2 = \sum_{K \leq L_2} a_K^2 x^K$, with t_1 and t_2 terms, respectively, denoted as $\texttt{Imp}(B_{p_1,L_1})$ and $\texttt{Imp}(B_{p_2,L_2})$. To sum these polynomials in their IBF, we first dissect $\texttt{Imp}(B_{p_1,L_1})$ and $\texttt{Imp}(B_{p_2,L_2})$ along their primary dimension into t_1 and t_2 submatrices, respectively, as $\texttt{Imp}^i(B_{p_1,L_1})$ and $Imp^j(B_{p_2,L_2})$. Subsequent to applying degree elevation via (21) and (22) to each submatrix, we amalgamate the resulting matrices vertically in accordance with (23). This efficient and elegant approach encapsulates the core principle of multivariate Bernstein polynomial summation in a multiterm context.

D. Memory and Computational Complexity of Implicit Bernstein Polynomial Representations

The following result summarizes the benefits of the proposed representation of the implicit Bernstein polynomial representation.

Proposition 4: Given n-dimensional Bernstein polynomial p that comprises of t terms and of order $L = (l_1, \ldots, l_n)$, $B_{p,L}(x)$. Its implicit representation $\texttt{Imp}(B_{p,L})$ is 2-D tensor of size $nt \times (l_{\max} + 1)$, i.e., the memory complexity of the implicit representation is $O(ntl_{\max})$. Moreover, the computational complexity of **Sum_Bern_Imp** and **Prod_Bern_Imp** is $O(t_{\max}^2 n l_{\max})$, where $l_{\max} = \max_{1 \leq i \leq n} l_i$ is the maximum order of the polynomials obtained during the bound propagation and $t_{\max}$ is the maximum number of terms in the polynomials obtained during the bound propagation.

Comparing Propositions 1 and 4, we can notice that the implicit representation reduced the scaling—with respect to the dimension n—from exponential scaling in the dense representation to linear scaling on the expense of depending on the number of terms t. In the worst case, when the number of terms t grows exponentially, the implicit form may result in exponential memory usage. Nevertheless, in practice—as we show in Section VI—the number of terms does not grow exponentially, which makes the implicit representation ideal for higher dimensions. In addition, the IBF representation is always a 2-D tensor no matter the dimension n or the polynomial order compared to its counterpart— the dense representation—which represents the polynomial as an n-dimensional tensor.

V. GPU Algorithms for Bernstein Polynomial Extrema

Recall that the polynomial-based interval bound propagation—described in Section II-C—requires three procedures, namely, **Sum_Bern**, **Prod_Bern**, and **Min/Max_Bern**. While Section IV showed how to address the memory and computational challenges with **Sum_Bern** and **Prod_Bern** using the 2-D tensor encoding, computing the minimum and maximum coefficients (the **Min/Max_Bern** procedure) cannot be carried over using the 2-D tensor encoding since it requires access to the dense n-dimensional tensor representation.

A direct approach is to convert the implicit representation $\texttt{Imp}(B_{p,L})$ into the corresponding dense representation $\texttt{Den}(B_{p,L})$ followed by finding the minimum and maximum coefficients. This conversion can be computationally demanding and memory-intensive, especially for large degrees d and input dimensions (number of variables) n. To address potential inefficiencies, we aim to avoid materializing the tensor explicitly, yet we aim to find the minimum and maximum values within the n-dimensional dense tensor $\texttt{Den}(B_{p,L})$ while avoiding storing the entire tensor in memory.

A. Implicit Form Min–Max Computation

To find the minimum and maximum values within the tensor $T_{\texttt{Den}} = \texttt{Den}(B_{p,L})$ without explicitly computing and storing it in the memory, we design a customized CUDA kernel. Given a 2-D implicit tensor $T_{\texttt{Imp}} = \texttt{Imp}(B_{p,L})$, we can index into the

tensor $T_{\texttt{Den}}(i, j, k, \ldots)$ by directly accessing memory locations offset by the axis sizes

$$T_{\texttt{Den}}(i, j, k, \ldots) = T_{\texttt{Imp}}[id^{n-1} + jd^{n-2} + kd^{n-3} + \ldots].$$

Using this connection between the dense and implicit representation, our CUDA kernel uses multiple threads in parallel to compute the local minimum/maximum within partitions of the dense form. Each CUDA thread is associated with an ID, `ebf_id` in Algorithm 2, that is mapped to a unique set of indices $\{i, j, k, \ldots\}$ to access and compute elements of the explicit Bernstein form. As t (number of terms), d (number of columns in the implicit form tensor), and n (input dimension) are known, this mapping is achieved through a set of iterative equations

$$\begin{aligned} i &= \lfloor t/d^{n-1} \rfloor \\ j &= \lfloor t/d^{n-2} \rfloor - id \\ k &= \lfloor t/d^{n-3} \rfloor - id^2 - jd \\ &\ldots \end{aligned} \tag{24}$$

The indices are computed on-the-fly within the algorithm, eliminating the need for storage. This iteration corresponds to lines 12–14 of Algorithm 2. The variable `index` corresponds to the sequence of indices and the variable `tracker` corresponds to the subtracted portion. Precomputed powers d^{r-1} for $r \in \{1, \ldots, n\}$ are stored in constant global memory and all the threads in a warp compute the same power in each iteration, which ensures low-latency access. Algorithm 2 returns the extrema for the portion of the tensor $T_{\texttt{Imp}}$ that is covered by a CUDA block. Then, we apply an additional reduction to compute the global extrema.

B. Quadrant-Constrained Min–Max Computation

In models where the input domain I_n is constrained to a single quadrant, specifically when all the variables are positive, we can significantly streamline the computation for both the min and max values. This constraint allows for a simplified kernel, where the computation narrows down to evaluating only two points in the dense n-dimensional tensor form. This simplification effectively eliminates the need for a loop over the entire dense tensor, reducing the computational complexity to a single iteration over the terms.

C. Distribution Strategy and Challenges

Distributing Bern-NN-IBF across multiple GPUs is essential for handling large models that may not fit within the memory constraints of a single GPU. Our approach involves having each GPU compute bounds for a batch of nodes in a layer, followed by an *allgather* operation to ensure that all GPUs have the input bounds necessary for processing the subsequent layer.

We leverage PyTorch Distributed with the NCCL backend [28] for efficient GPU communication. For the allgather operation, we use `torch.distributed.all_gather_object`. This choice is motivated by the need to communicate Python objects, specifically tensors of different shapes, as part of the bounding process. However, this flexibility comes at a cost of communication inefficiencies because it involves transferring tensors from the GPU to the CPU during the pickling process (i.e., serialization of Python objects into a byte stream that operates on CPU memory). This additional data transfer can be an overhead and impact performance, particularly when working with large polynomials and frequent communication between GPUs. Since `all_gather_object` cannot efficiently communicate objects over NVLink for direct GPU-to-GPU communication, we are likely to encounter a bottleneck that saturates the available communication bandwidth. Thus, our implementation provides an upper bound on the expected strong scaling.

Algorithm 2 Computing the Extrema of a Bernstein Polynomial in Implicit Form

Input: $T_{\texttt{Imp}} = \texttt{Imp}(B_{p,L})$, a 2D tensor representing a Bernstein polynomial. E, the number of elements in the explicit (or dense) Bernstein form. `nterms`, `nvars`, d respectively denote the number of terms, variables, and columns used in the implicit Bernstein form.

Output: $\min(B_{p,L})$, $\max(B_{p,L})$ for each CUDA block

1: **function ibf-extrema**($T_{\texttt{Imp}} = \texttt{Imp}(B_{p,L})$) **then**
2: `block_sum` ← zeros(E, `gridDim.y`)
3: `ebf_id` ← global thread id
4: **while** `ebf_id` < E **do**
5: `tsum` ← 0
6: `term_id` ← `blockIdx.y`
7: **while** `term_id` < `nterms` **do**
8: `acc` ← 1
9: `tracker` ← 0
10: **for** $v \in \{1, \ldots,$ `nvars`$\}$ **do**
11: $p \leftarrow$ lookup $d^{\texttt{nvars}-v-1}$
12: `index` ← $\lfloor$`ebf_id`$/p\rfloor$ − `tracker`
13: `acc` ← `acc` × $T_{\texttt{Imp}}$[`term_id`][v][`index`]
14: `tracker` ← (`tracker` + `index`) × d
15: **end for**
16: `tsum` ← `tsum` + `acc`
17: `term_id` ← `term_id` + `gridDim.y`
18: **end while**
19: `block_sum`[`ebf_id`][`blockIdx.y`] ← `tsum`
20: `ebf_id` ← `ebf_id` + `gridDim.x`
21: **end while**
22: **return** `block_sum`
23: **end function**

VI. Numerical Studies

In this section, we perform a series of numerical experiments to evaluate the scalability and effectiveness of our tool. First, we conduct an ablation study to check the effect of varying different parameters (e.g., NN width, NN depth, and ReLU approximation order) on the performance of our tool. We utilize two metrics.

1) *Execution Time:* which measures the time (in seconds) needed to compute the final Bernstein polynomials. Indeed, smaller values indicate better performance.
2) *Relative Volume of the Output Set:* this metric measures the "tightness" of the produced over- and under-approximation polynomials. Without loss of generality, we focus on NNs with one output z, and we compute this metric as

$$\text{Vol\_relative} = \frac{\text{Vol\_Output}}{\text{Vol\_Input}} = \frac{\overline{z} - \underline{z}}{\prod_{i=1}^{n} \left(\overline{d}_i - \underline{d}_i\right)} \quad (25)$$

where $\overline{z}$ and $\underline{z}$ are the upper and lower bounds on the NN's output z obtained by the end of the interval-bound propagation process. Indeed, smaller values of this metric indicate tighter approximations of the output set.

After the ablation study, we compare our tool with a set of state-of-the-art bound computation tools—including the winner of the last 2023 verification of NN (VNN) competition—to study the relative performance.

A. Ablation Studies

We compare the performance and bounds attained by the original BERN-NN [20] and the proposed BERN-NN-IBF for the ablation studies. For these comparisons, we attempt to push the approaches to their limits. We use randomly generated and fully connected NNs with a single output. We change the input dimension and the number of neurons in the hidden layers across the experiments. We run each experiment multiple times and report the execution time across all the experiments using box-and-whisker plots. All our experiments were performed using 8 Nvidia A100 GPU.

Experiment 1 (The Effect of Increasing the Hidden Dimension): First, we consider a four-layer NN and keep the input dimension fixed to 2 and the output size to 1. Each trial varies the dimensions of all the hidden layers. Fig. 3 reports the results of this experiment. The figure shows that the performance scaling is favorable for BERN-NN-IBF. Moreover, BERN-NN is designed to use only one 1 GPU while BERN-NN-IBF benefits significantly from the parallelization provided by the developed CUDA kernels and the ability to parallelize the algorithm across multiple GPUs. Moreover, we notice the volume of BERN-NN-IBF is 2× smaller than BERN-NN thanks to the optimal under-approximation (Algorithm 1) resulting in enhanced bounds.

Experiment 2 (The Effect of Increasing the Total Number of Layers): To study the effect of increasing the number of layers, we keep the input dimension fixed to two with a variable number of layers each with a hidden size of 5 neurons. Results are shown in Figs. 4. Again, we can see that BERN-NN-IBF achieves better scaling than the original BERN-NN. As we increase the number of layers, we see a more obvious performance win for BERN-NN-IBF. Even with this narrow network, we are able to achieve reasonable speedup by distributing over multiple GPUs. Also, similar

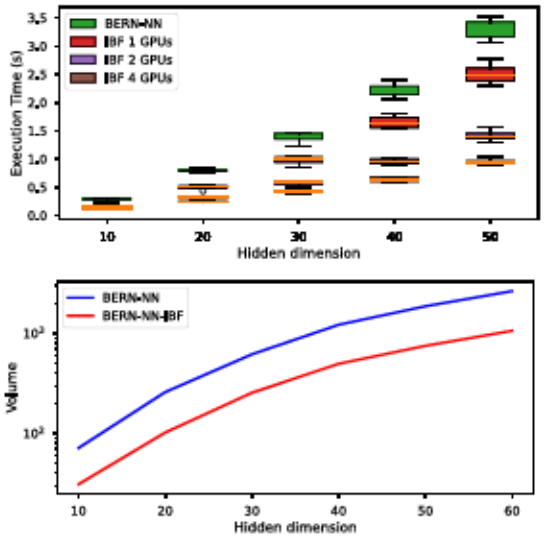

Fig. 3. Results of Experiment 1. (Top) Execution time versus hidden dimension. (Bottom) Relative volume versus hidden dimension.

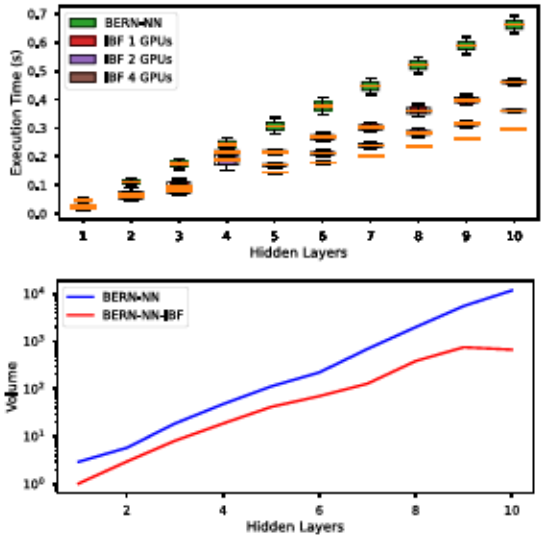

Fig. 4. Results of Experiment 2. (Top) Execution time versus the total number of layers. (Bottom) Relative volume versus the total number of layers.

to Experiment 1, we notice the effect of the optimal under-approximation (Algorithm 1) in enhancing the accuracy of the interval-bound propagation, leading to better relative volume.

Experiment 3 (Increasing the Total Number of Neurons): In Fig. 5, we compute the bounds for progressively larger models to compare the performance of the original BERN-NN against the developed BERN-NN-IBF. Again, we observe better scaling with BERN-NN-IBF even when using only 1 GPU for both BERN-NN and BERN-NN-IBF.

Experiment 4 (Assuming Positive Input Domain I_n): It is common that input data can be normalized to fall in the positive orthant. For instance, in computer vision applications, pixel values may be normalized to [0, 1]. Assuming that data falls into the positive orthant greatly simplifies finding the minimum and maximum of Bernstein polynomials (Algorithm 2), as we no longer need to convert to the explicit form. In this experiment, we use a network with an input dimension of 10 and a hidden layer of varying dimensions. Results are shown in

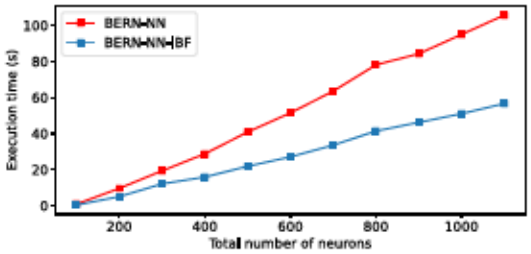

Fig. 5. Results of Experiment 3. Execution time versus the total number of neurons. Each layer has 100 neurons, and we successively add one layer.

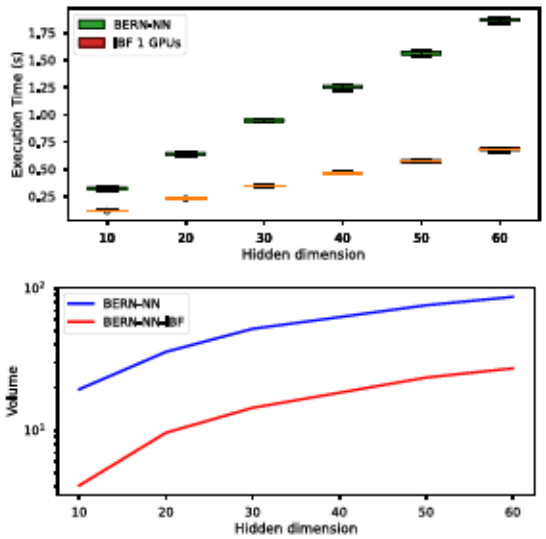

Fig. 6. Results of Experiment 4. (Top) Execution time versus the total number of layers. (Bottom) Relative volume versus the total number of layers.

Fig. 6. We notice that making the positive orthant assumption results in a massive performance boost, with relatively little effect on the resulting bounds in this case.

B. Comparison Against State-of-the-Art Tools

In this experiment, we compare the performance of our tool in terms of execution time and the output set's relative volume compared to bound propagation tools, such as symbolic interval analysis (SIA) [29] and part of alpha-CROWN [30]. We note that alpha-CROWN [30] won the 2023 VNN competition. We compare Bern-NN against the bound propagation algorithm used within alpha-CROWN as a representative tool for all the bound propagation techniques. Moreover, alpha-CROWN is also designed to harness the computational powers of GPUs.

Experiment 5 (Random Neural Networks): We compare the performance of our tool to SIA, alpha-CROWN, and BERN-NN for random NNs with varying numbers of neurons in the hidden layers (Fig. 7). We compare the execution time and relative volume as a function of the model's hidden dimension. The time and volume reported are the averages of ten trials on randomized models. We also compare the performance as the input dimension of the network increases (Fig. 8).

The results show that SIA is the fastest in terms of execution time for all different input hyperrectangles due to the simplicity of its computations. However, its relative volume is the highest. On the other hand, BERN-NN-IBF's relative volume is the smallest for all input spaces, thanks to its

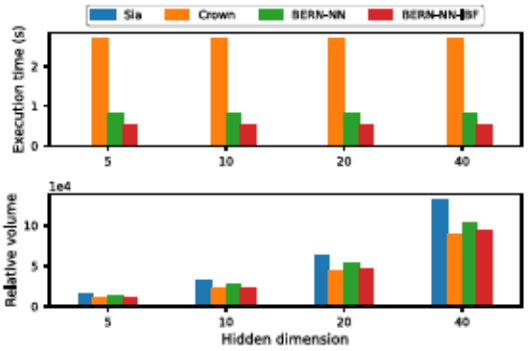

Fig. 7. Results of Experiment 5 for varying the number of neurons in each hidden layer.

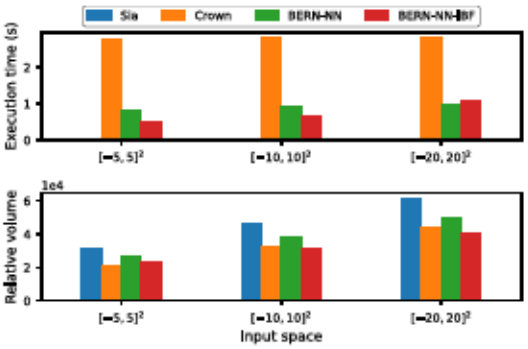

Fig. 8. Results of Experiment 5 for varying the input volume.

tight higher-order ReLU approximations. Compared to alpha-CROWN (which also runs on GPUs), BERN-NN-IBF is both faster and produces tighter bounds, leading to an average of 5× execution time speedup while achieving the same or better in the relative volume metric. We conclude that BERN-NN-IBF generally strikes a good balance between performance and tightness.

Experiment 6 (Case Study for Control Benchmarks): In this experiment, we comprehensively assessed various tools applied to benchmarks derived from NN controllers [21] to determine the precision of their estimated bounds. The architecture of the networks employed in each benchmark can be found in [21]. Table I encapsulates the performance metrics of these tools, focusing on average execution time and average relative volume across six control benchmarks. Notably, BERN-NN-IBF consistently emerged as the second quickest tool, yet it invariably provided the most accurate bounds. This accuracy is particularly significant for control applications, where the specification of interest often spans a time horizon and necessitates multistep reachability analysis; therefore, achieving finer bounds at each stage is imperative. Moreover, BERN-NN-IBF outperformed CROWN in terms of speed across all benchmarks, with the exception of Benchmark 5. While SIA exhibited faster performance than BERN-NN-IBF, it compromised on the precision of bound estimates. Each benchmark was subjected to tests with five distinct hyperrectangles, all centered at zero, with radii varying within $1, 1.5, 2, 2.5, 3$, to ensure a robust evaluation. This rigorous testing methodology underscores the effectiveness of BERN-NN-IBF in delivering precise and computationally efficient solutions for control applications, highlighting its

TABLE I
RESULTS OF EXPERIMENT 6: EXECUTION TIME AND VOLUME FOR SIA, ALPHA-CROWN, BERN-NN, AND BERN-NN-IBF

	Benchmark 1		Benchmark 2		Benchmark 3		Benchmark 4		Benchmark 5		Benchmark 6	
	Time (s)	Vol	Time	Vol	Time	Vol	Time	Vol	Time	Vol	Time	Vol
Sia	**0.013**	2.549	**0.010**	9.790	**0.010**	1.032	**0.010**	8.549	**0.014**	53.391	**0.012**	2.798
Crown	2.389	3.641	2.860	**8.574**	2.875	**0.735**	2.674	15.869	2.912	77.734	3.077	3.264
BERN-NN	0.551	**1.442**	0.846	9.629	0.844	1.106	0.916	8.921	18.367	54.810	1.338	3.454
BERN-NN-IBF	0.218	1.719	0.475	9.003	0.357	0.835	0.304	**7.161**	7.664	**42.747**	0.540	**2.141**

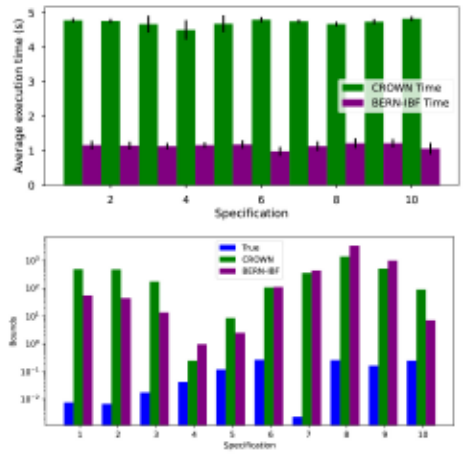

Fig. 9. Results of Experiment 7. (Top) Average execution time of Crown and BERN-NN-IBF on the ACAS XU benchmark. Error bars represent the standard deviation. (Bottom) Average bound of five neurons across ten specifications from the ACAS XU benchmark. True bounds (blue) were obtained by evaluating the NN with 100 000 samples from the input domain.

superiority in optimizing both speed and accuracy in bound estimation.

Experiment 7 (Case Study for ACAS Xu Benchmark): In this comprehensive experiment, we assess the efficacy of BERN-NN-IBF in contrast to CROWN within the context of the unmanned Airborne Collision Avoidance System (ACAS Xu) benchmark, as detailed in [31]. This benchmark encompasses ten distinct properties across 45 NNs that are instrumental in generating turn advisories for aircraft to avert collisions. Each network comprises 300 neurons distributed over six layers, utilizing ReLU activation functions. The networks are designed with five inputs that represent the states of the aircraft and produce five outputs, with the system adopting the minimum output value as the turn advisory. Further insights into this benchmark are elaborated in [31].

Empirical data presented in Fig. 9 underscore the superior performance of BERN-NN-IBF over CROWN, both in terms of computational speed and the precision of the volume estimations across all ten specifications. This enhancement in performance is pivotal, particularly in the high-stakes domain of collision avoidance, where the rapid and accurate computation of turn advisories is critical for ensuring the safety of the airspace. BERN-NN-IBF's ability to outperform CROWN in these key areas demonstrates its potential to significantly improve the reliability and efficiency of NN-based decision-making systems in safety-critical applications.

C. Scaling Bern-NN-IBF Across Multiple GPUs

Experiment 8 (Strong Scaling Experiments): To demonstrate the scalability of our approach, we perform strong scaling

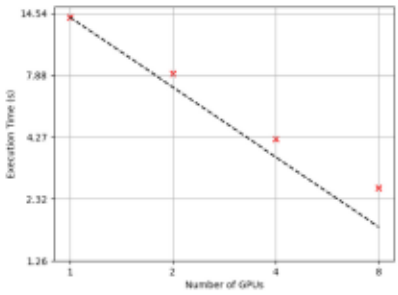

Fig. 10. Strong scaling up to eight NVIDIA A100 GPUs. We use a fixed model with an input dimension of 5, two hidden layers with 100 neurons each, and an output dimension of 1. The dashed line represents the ideal strong scaling. The red crosses are the average runtime of 20 trials with the corresponding GPU count. We found that the 95% confidence intervals of the mean runtimes are all within 5% of the mean, so we excluded them from the plot.

experiments on eight NVIDIA A100 GPUs. Results are shown in Fig. 10. The batch-parallel computation of node bounds across GPUs accelerates BERN-NN-IBF while preserving the bounds accuracy. This enables bounding larger models, where intermediate computations do not fit in a single GPU's memory.

Despite the limitations of *allgather with pickling* package, we expect strong scaling due to the computational complexity of computing bounds for each layer. In future work, exploring alternative communication strategies or optimizations tailored for large polynomial data transfers may be essential to further enhance the scalability of distributed BERN-NN-IBF.

Experiment 9 (Memory Scalability Experiments): Finally, we compare the memory scalability of BERN-NN-IBF against BERN-NN. In this experiment, we perform memory scalability over a single NVIDIA A100 GPU while limiting the GPU memory to 10 GB. We use NNs with random weights, 5 layers and 25 neurons per hidden layer. We iteratively increase the input dimension n and we record the input dimension at which the GPU will run out of memory. Our experiments show that when $n = 6$, the dense representation used in BERN-NN consumes the entire 10 GB of memory and the BERN-NN can no longer finish the bound propagation procedure. On the other side, and thanks to the 2-D tensor representation used in BERN-NN-IBF, we can scale up to $n = 25$. Repeating the same experiment while limiting the GPU memory to 20 GB results in a maximum $n = 8$ for BERN-NN and a maximum $n = 47$ for BERN-NN-IBF which reflects the linear scalability with the input dimension n in Proposition 4.

VII. CONCLUSION

This article introduces BERN-NN-IBF, which significantly enhances the BERN-NN framework by implementing the IBF

to improve memory efficiency. Key innovations include the design of IBF-specific operations and the implementation of specialized CUDA routines, improving the speed and accuracy of essential functions like summations and multiplications. Additionally, we developed a novel optimization technique for determining the coefficients of a quadratic polynomial under-approximation of the ReLU function, resulting in tighter output bounds. Empirical evaluations demonstrate that BERN-NN-IBF outperforms not only its predecessor, BERN-NN, but also state-of-the-art tools. These advancements position BERN-NN-IBF as a highly efficient tool for NN bound propagation, offering significant improvements in memory efficiency, computational speed, and precision.

REFERENCES

[1] X. Sun, H. Khedr, and Y. Shoukry, "Formal verification of neural network controlled autonomous systems," in *Proc. 22nd ACM Int. Conf. Hybrid Syst., Comput. Control*, 2019, pp. 147–156.

[2] D. J. Fremont, J. Chiu, D. D. Margineantu, D. Osipychev, and S. A. Seshia, "Formal analysis and redesign of a neural network-based aircraft taxiing system with VerifAI," in *Proc. Int. Conf. Comput. Aided Verif.*, 2020, pp. 122–134.

[3] U. Santa Cruz and Y. Shoukry, "NNLander-VeriF: A neural network formal verification framework for vision-based autonomous aircraft landing," in *Proc. NASA Formal Methods Symp.*, 2022, pp. 213–230.

[4] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *Proc. Int. Conf. Comput. Aided Verif.*, 2017, pp. 97–117.

[5] S. Dutta, X. Chen, S. Jha, S. Sankaranarayanan, and A. Tiwari, "Sherlock-a tool for verification of neural network feedback systems: Demo abstract," in *Proc. 22nd ACM Int. Conf. Hybrid Syst., Comput. Control*, 2019, pp. 262–263.

[6] E. Botoeva, P. Kouvaros, J. Kronqvist, A. Lomuscio, and R. Misener, "Efficient verification of relu-based neural networks via dependency analysis," in *Proc. AAAI Conf. Artif. Intell.*, vol. 34, 2020, pp. 3291–3299.

[7] V. Tjeng, K. Y. Xiao, and R. Tedrake, "Evaluating robustness of neural networks with mixed integer programming," in *Proc. Int. Conf. Learn. Represent.*, 2019, pp. 1–21.

[8] O. Bastani, Y. Ioannou, L. Lampropoulos, D. Vytiniotis, A. Nori, and A. Criminisi, "Measuring neural net robustness with constraints," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 29, 2016, pp. 2613–2621.

[9] R. Bunel, J. Lu, I. Turkaslan, P. Kohli, P. Torr, and P. Mudigonda, "Branch and bound for piecewise linear neural network verification," *J. Mach. Learn. Res.*, vol. 21, no. 42, pp. 1–39, 2020.

[10] M. Fischetti and J. Jo, "Deep neural networks and mixed integer linear optimization," *Constraints*, vol. 23, no. 3, pp. 296–309, 2018.

[11] R. Anderson, J. Huchette, W. Ma, C. Tjandraatmadja, and J. P. Vielma, "Strong mixed-integer programming formulations for trained neural networks," *Math. Program.*, vol. 183, no. 1, pp. 3–39, 2020.

[12] C.-H. Cheng, G. Nührenberg, and H. Ruess, "Maximum resilience of artificial neural networks," in *Proc. Int. Symp. Autom. Technol. Verif. Anal.*, 2017, pp. 251–268.

[13] W. Xiang, H.-D. Tran, J. A. Rosenfeld, and T. T. Johnson, "Reachable set estimation and safety verification for piecewise linear systems with neural network controllers," in *Proc. Annu. Amer. Control Conf. (ACC)*, 2018, pp. 1574–1579.

[14] W. Xiang, H.-D. Tran, and T. T. Johnson, "Output reachable set estimation and verification for multilayer neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 11, pp. 5777–5783, Nov. 2018.

[15] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "AI2: Safety and robustness certification of neural networks with abstract interpretation," in *Proc. IEEE Symp. Security Privacy (SP)*, 2018, pp. 3–18.

[16] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal security analysis of neural networks using symbolic intervals," in *Proc. 27th USENIX Conf. Security Symp.*, 2018, pp. 1599–1614.

[17] H.-D. Tran et al., "NNV: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems," in *Proc. Int. Conf. Comput. Aided Verif.*, 2020, pp. 3–17.

[18] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Verisig: Verifying safety properties of hybrid systems with neural network controllers," in *Proc. 22nd ACM Int. Conf. Hybrid Syst., Comput. Control*, 2019, pp. 169–178.

[19] M. Fazlyab, A. Robey, H. Hassani, M. Morari, and G. Pappas, "Efficient and accurate estimation of Lipschitz constants for deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 11423–11434.

[20] W. Fatnassi, H. Khedr, V. Yamamoto, and Y. Shoukry, "BERN-NN: Tight bound propagation for neural networks using bernstein polynomial interval arithmetic," in *Proc. 26th ACM Int. Conf. Hybrid Syst., Comput. Control*, 2023, pp. 1–11.

[21] C. Huang, J. Fan, X. Chen, W. Li, and Q. Zhu, "POLAR: A polynomial arithmetic framework for verifying neural-network controlled systems," in *Proc. Int. Symp. Autom. Technol. Verif. Anal.*, 2022, pp. 414–430.

[22] J. Ferlez, H. Khedr, and Y. Shoukry, "Fast BATLLNN: Fast box analysis of two-level lattice neural networks," in *Proc. 25th ACM Int. Conf. Hybrid Syst., Comput. Control*, 2022, pp. 1–11.

[23] H. Khedr and Y. Shoukry, "DeepBern-Nets: Taming the complexity of certifying neural networks using Bernstein polynomial activations and precise bound propagation," in *Proc. AAAI Conf. Artif. Intell.*, vol. 38, no. 19, 2024, pp. 21232–21240.

[24] V. Lin, R. Ivanov, J. Weimer, O. Sokolsky, and I. Lee, "T4V: Exploring neural network architectures that improve the scalability of neural network verification," in *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*. Cham, Switzerland: Springer, 2022, pp. 585–603.

[25] S. Wang et al., "Beta-CROWN: Efficient bound propagation with per-neuron split constraints for neural network robustness verification," in *Proc. Adv. Neural Inf. Process. Syst.*, 2021, pp. 1–27.

[26] J. Sánchez-Reyes, "Algebraic manipulation in the Bernstein form made simple via convolutions," *Comput.-Aided Design*, vol. 35, no. 10, pp. 959–967, 2003.

[27] R. T. Farouki and V. Rajan, "Algorithms for polynomials in Bernstein form," *Comput. Aided Geom. Design*, vol. 5, no. 1, pp. 1–26, 1988.

[28] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[29] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Efficient formal safety analysis of neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 31, 2018, pp. 6367–6377.

[30] K. Xu et al., "Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers," in *Proc. ICLR*, 2021, pp. 1–15.

[31] D. M. Lopez, T. T. Johnson, S. Bak, H.-D. Tran, and K. Hobbs, "Evaluation of neural network verification methods for air to air collision avoidance," *AIAA J. Air Transp.*, vol. 31, no. 1, pp. 1–7, 2022.